Logic Synthesis and Verification

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Fall 2024

SOPs and Incompletely Specified Functions

Reading:

Logic Synthesis in a Nutshell

Section 2

most of the following slides are by courtesy of Andreas Kuehlmann

Boolean Function Representation

Sum of Products

- A function can be represented by a sum of cubes (products):
  - E.g., f = ab + ac + bc

    Since each cube is a product of literals, this is a "sum of products" (SOP) representation

- An SOP can be thought of as a set of cubes F
  - E.g., F = {ab, ac, bc}

- A set of cubes that represents f is called a cover of f
  - E.g.,

    F_1={ab, ac, bc} and F_2={abc, abc', ab'c, a'bc} are covers of f = ab + ac + bc.

List of Cubes (Cover Matrix)

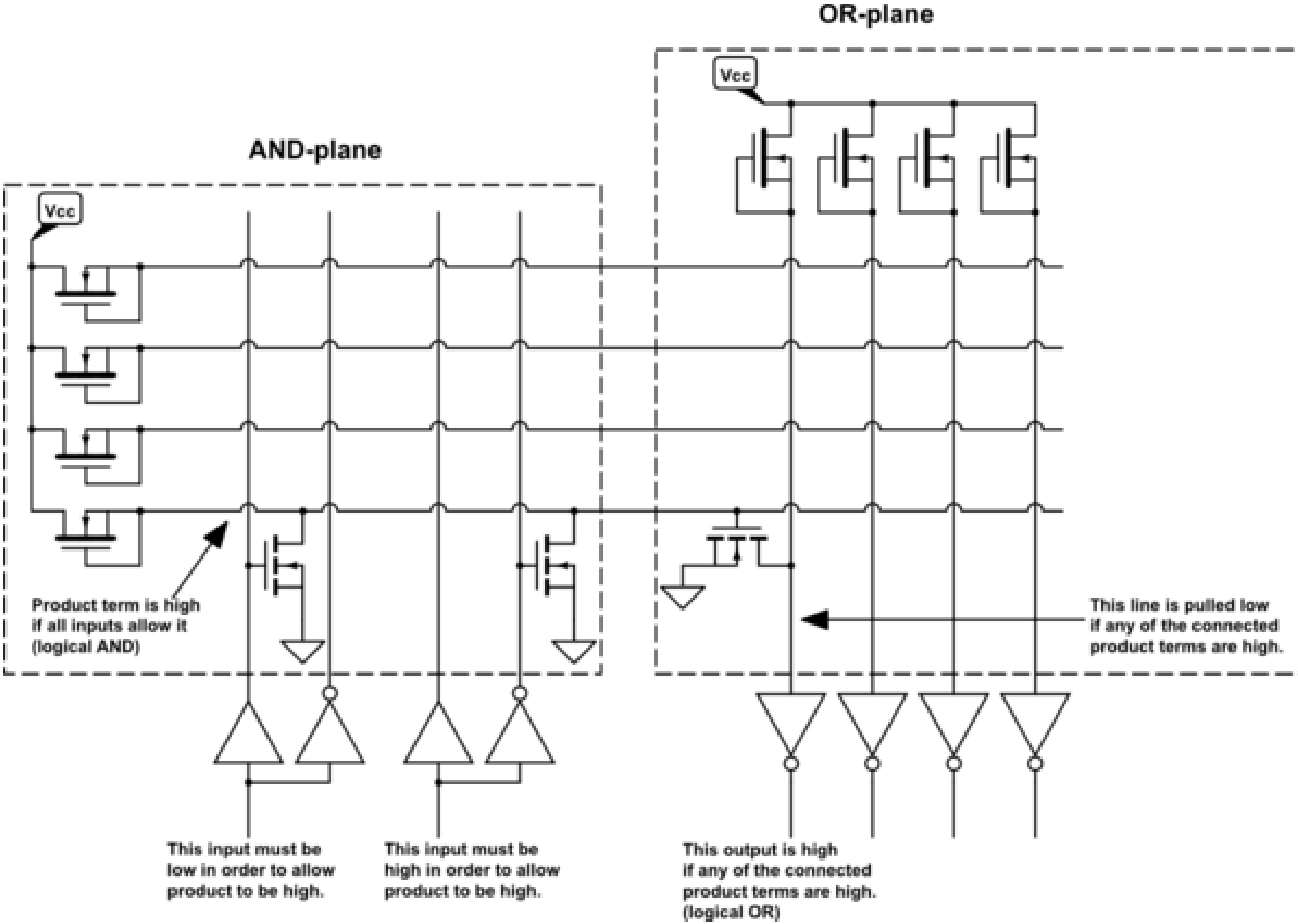
- We often use a matrix notation to represent a cover:
  - Example

    F = ac + c'd =

		a b c d		a b c d
a c	$\rightarrow$	1 2 1 2	or	1 - 1 -
c'd	$\rightarrow$	2 2 0 1		- - 0 1

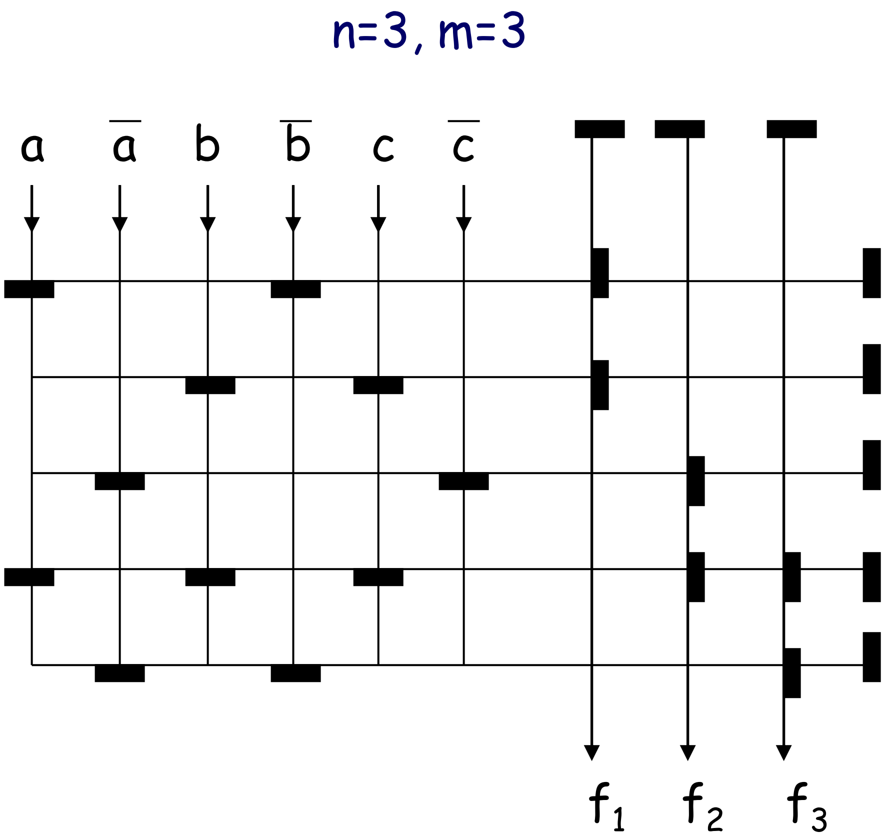
- Each row represents a cube
- 1 means that the positive literal appears in the cube
- 0 means that the negative literal appears in the cube
- 2 (or -) means that the variable does not appear in the cube. It implicitly represents both 0 and 1 values.

Programmable Logic Array (PLA)



PLA

- A PLA implements a (multiple-output) function $f : B^n \rightarrow B^m$ represented in SOP form



cover matrix

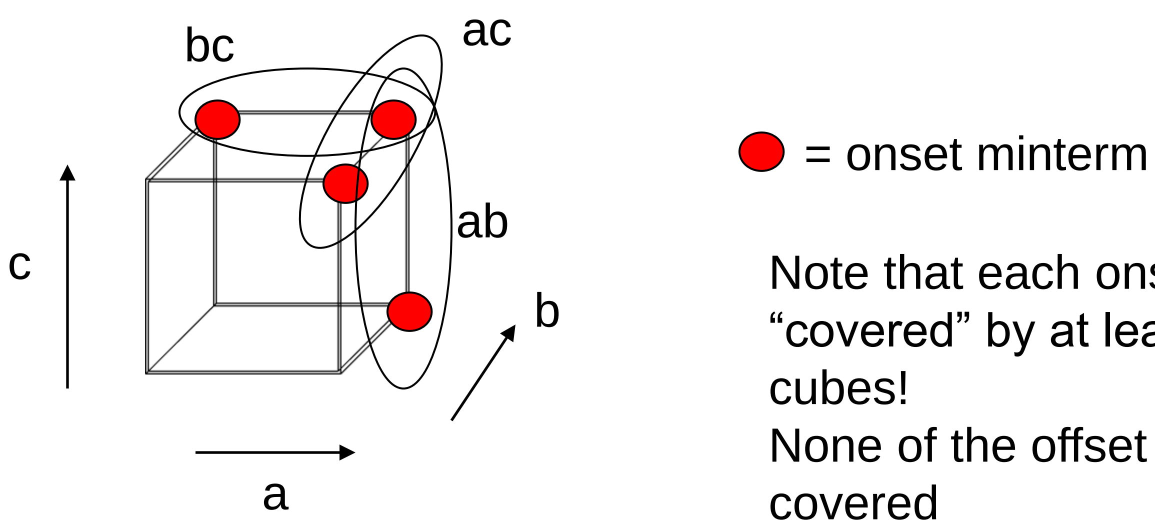
abc	$f_1 f_2 f_3$
10-	1 - -
-11	1 - -
0-0	- 1 -
111	- 1 1
00-	- - 1

PLA

- Each distinct cube appears just once in the AND-plane, and can be shared by (multiple) outputs in the OR-plane, e.g., cube (abc) in the previous slide
- Extensions from single-output to multiple-output minimization theory are straightforward

SOP

- The cover (set of cubes) can efficiently represent many practical logic functions (i.e., for many practical functions, there exist small covers)
- Two-level minimization seeks the cover of minimum size (least number of cubes)



Note that each onset minterm is "covered" by at least one of the cubes!
None of the offset minterms is covered

Irredundant Cube

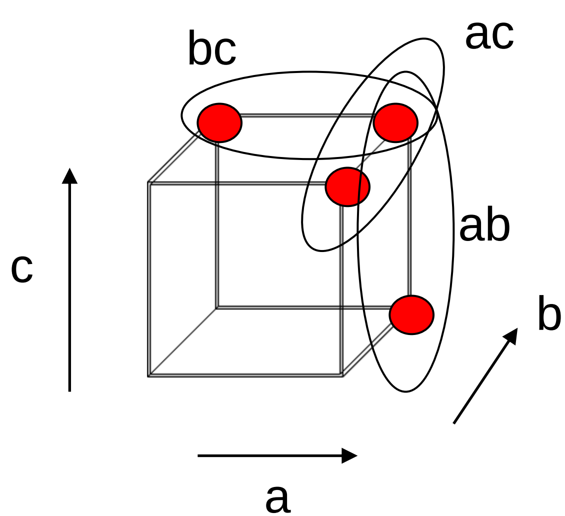
- Let $F = \{c_1, c_2, \ldots, c_k\}$ be a cover for f, i.e.,

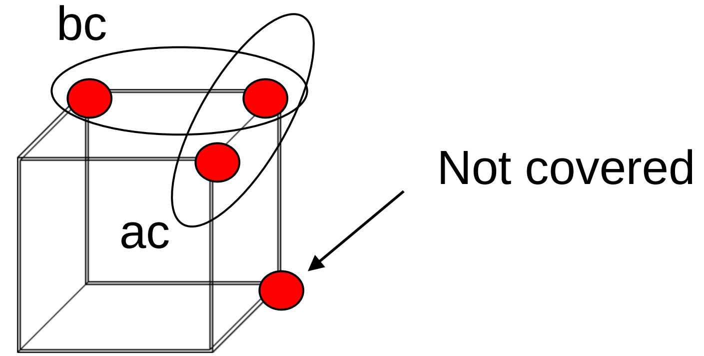
  $f = \sum_{i=1}^{k} c_i$

  A cube $c_i \in F$ is irredundant if $F \setminus \{c_i\} \neq f$

  - Example

    $f = ab + ac + bc$





$F \setminus \{ab\} \neq f$

Prime Cube

- A literal x (a variable or its negation) of cube $c \in F$ (cover of f) is prime if $(F \setminus \{c\}) \cup \{c_x\} \neq f$,
  where c_x (cofactor w.r.t. x) is c with literal x of c deleted

- A cube of F is prime if all its literals are prime

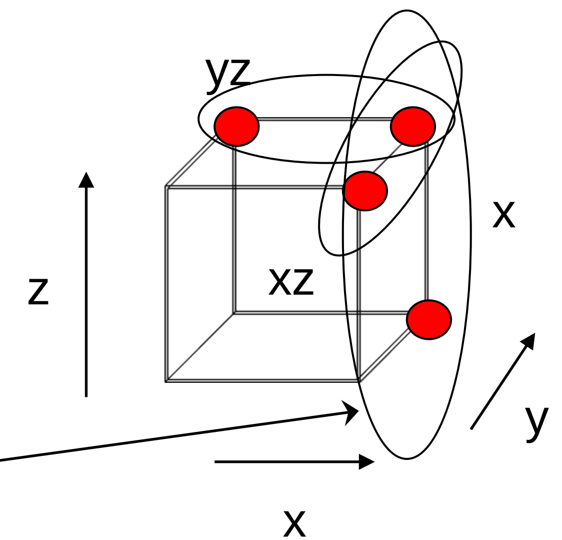
- Example

$$f = xy + xz + yz$$

$$c = xy;\ c_y = x \text{ (literal y deleted)}$$

$$(F \setminus \{c\}) \cup \{c_y\} = x + xz + yz$$

inequivalent to f since
offset vertex is covered



Prime and Irredundant Cover

- Definition 1. A cover is prime (resp. irredundant) if all its cubes are prime (resp. irredundant)

- Definition 2. A prime (cube) of f is essential (essential prime) if there is a onset minterm (essential vertex) in that prime but not in any other prime

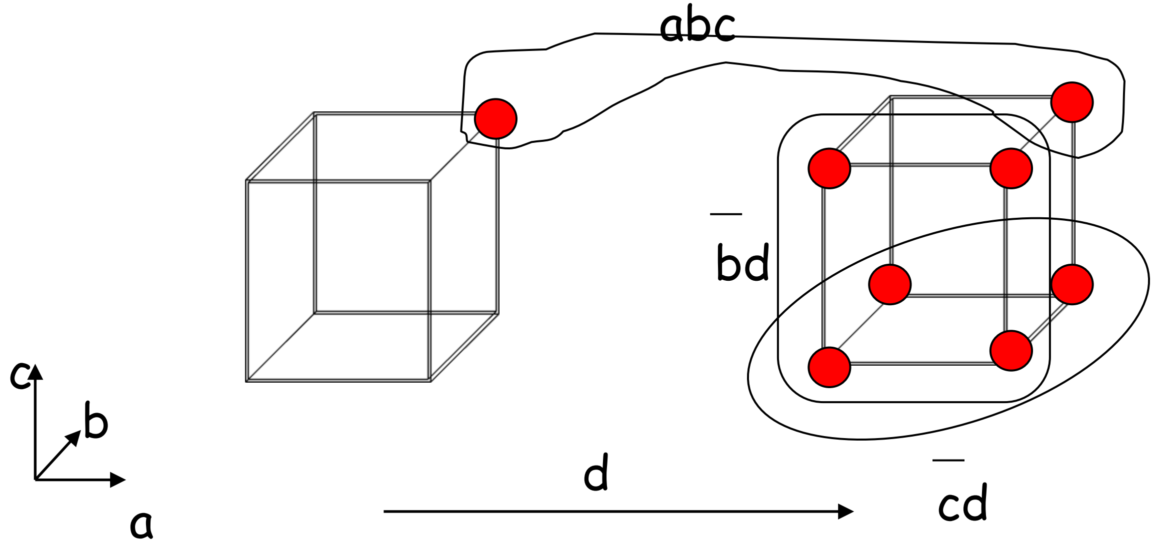
- Definition 3. Two cubes are orthogonal if they do not have any minterm in common
  - E.g. $c_1 = xy$ $c_2 = y'z$ are orthogonal
    $c_1 = x'y$ $c_2 = yz$ are not orthogonal

Prime and Irredundant Cover

- Example

  f = abc + b'd + c'd is prime and irredundant.

  abc is essential since abcd'∈abc, but not in b'd or c'd or ad



Why is abcd not an essential vertex of abc?
What is an essential vertex of abc?
What other cube is essential? What prime is not essential?

Incompletely Specified Function

- Let $F = (f, d, r) : B^n \rightarrow \{0, 1, *\}$, where * represents “don’t care”
  - f = onset function $\quad f(x)=1 \leftrightarrow F(x)=1$
  - r = offset function $\quad r(x)=1 \leftrightarrow F(x)=0$
  - d = don’t care function $\quad d(x)=1 \leftrightarrow F(x)=*$
- (f,d,r) forms a *partition* of B^n, i.e,
  - $f + d + r = B^n$
  - $(f \cdot d) = (f \cdot r) = (d \cdot r) = \varnothing$ (pairwise disjoint)

    (Here we don’t distinguish characteristic functions and the sets they represent)

Incompletely Specified Function

- A completely specified function g is a cover for F = (f,d,r) if

  $f \subseteq g \subseteq f+d$

  - $g \cdot r = \varnothing$
  - if $x \in d$ (i.e. $d(x)=1$), then $g(x)$ can be 0 or 1; if $x \in f$, then $g(x) = 1$; if $x \in r$, then $g(x) = 0$
    - We "don't care" which value g has at $x \in d$

Prime of Incompletely Specified Function

- Definition. A cube c is a prime of F = (f,d,r) if c $\subseteq$ f+d (an implicant of f+d), and no other implicant (of f+d) contains c (i.e., it is simply a prime of f+d)

- Definition. Cube c_j of cover G = $\{c_i\}$ of F = (f,d,r) is redundant if f $\subseteq$ G$\backslash\{c_j\}$; otherwise it is irredundant

- Note that c $\subseteq$ f+d $\leftrightarrow$ c $\cdot$ r = $\varnothing$

Prime of Incompletely Specified Function

- Example
  Consider logic minimization of $F(a,b,c)=(f,d,r)$ with $f=a'bc'+ab'c+abc$ and $d = abc'+ab'c'$

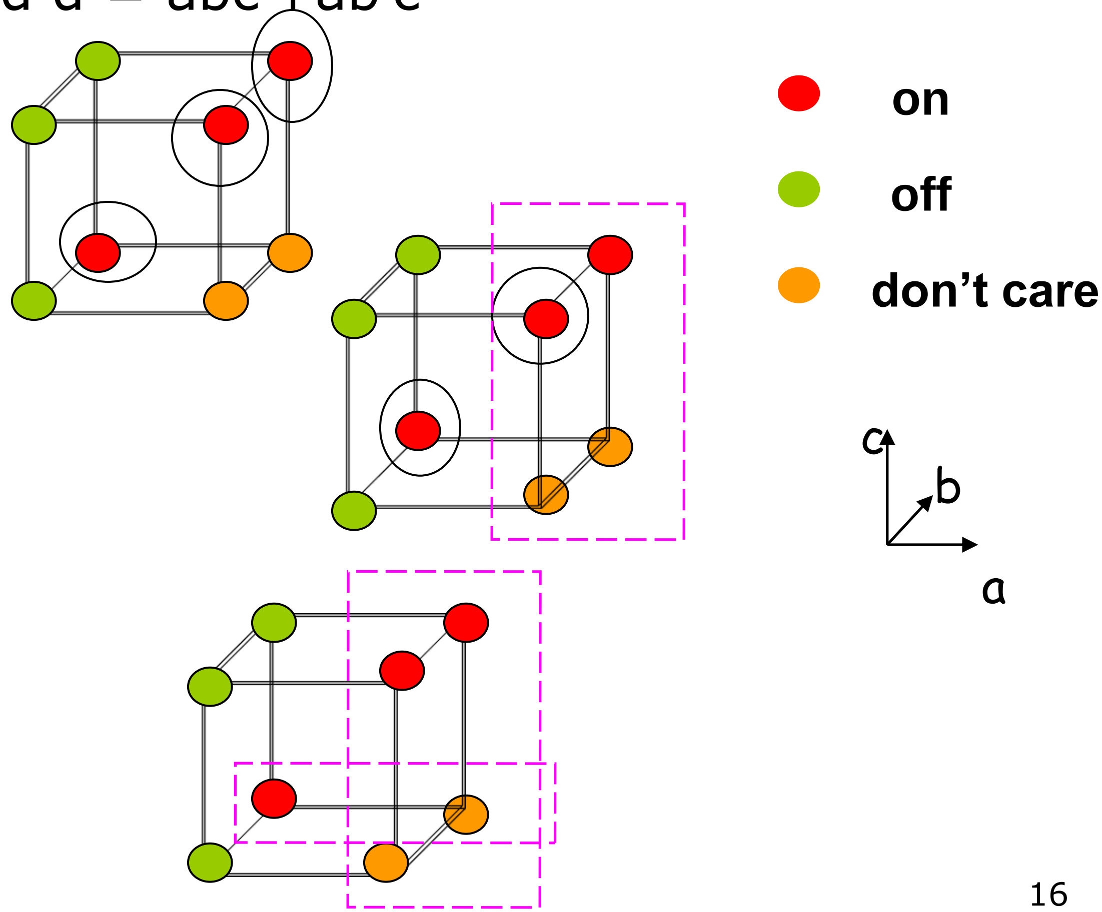
$F_1=\{a'bc', ab'c, abc\}$

Expand $abc \rightarrow a$

$F_2=\{a, a'bc', ab'c\}$

$ab'c$ is redundant
a is prime
$F_3= \{a, a'bc'\}$
Expand $a'bc' \rightarrow bc'$

$F_4= \{a, bc'\}$



Checking of Prime and Irredundancy

Let G be a cover of F = (f,d,r), and D be a cover for d

- $c_i \in G$ is redundant iff
  $c_i \subseteq (G\backslash\{c_i\}) \cup D$ (1)

  (Let $G^i \equiv G\backslash\{c_i\} \cup D$. Since $c_i \subseteq G^i$ and $f \subseteq G \subseteq f+d$, then $c_i \subseteq c_i f + c_i d$ and $c_i f \subseteq G\backslash\{c_i\}$. Thus $f \subseteq G\backslash\{c_i\}$.)

- A literal $l \in c_i$ is prime if $(c_i\backslash\{ l \})$ ($= (c_i)_l$) is not an implicant of F
- A cube c_i is a prime of F iff all literals $l \in c_i$ are prime
  Literal $l \in c_i$ is not prime $\Leftrightarrow (c_i)_l \subseteq f+d$ (2)

Note: Both tests (1) and (2) can be checked by tautology (to be explained):

- $(G^i)_{c_i} \equiv 1$ (implies c_i redundant)
- $(f \cup d)_{(c_i)_l} \equiv 1$ (implies l not prime)
  The above two cofactors are with respect to cubes instead of literals

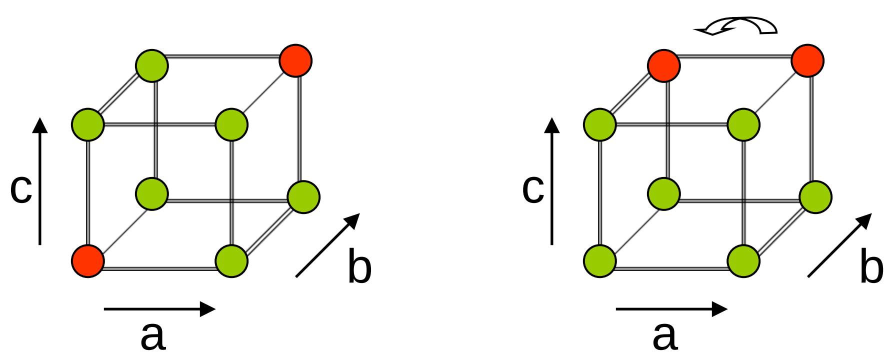
(Literal) Cofactor

- Let $f : B^n \rightarrow B$ be a Boolean function, and $x = (x_1, x_2, \ldots, x_n)$ the variables in the support of f; the cofactor f_a of f by a literal $a = x_i$ or $a = \neg x_i$ is
- $f_{x_i}(x_1, x_2, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$
- $f_{\neg x_i}(x_1, x_2, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$

The computation of the cofactor is a fundamental operation in Boolean reasoning!

- Example

$f = \bar{a}\bar{b}\bar{c} + abc$ $\qquad$ $f_a = bc$



(Literal) Cofactor

- The cofactor C_{x_j} of a cube C (representing some Boolean function) with respect to a literal x_j is
  - C if x_j and x_j' do not appear in C
  - $C\backslash\{x_j\}$ if x_j appears positively in C, i.e., $x_j \in C$
  - $\varnothing$ if x_j appears negatively in C, i.e., $x_j' \in C$

- Example

$C = x_1 x_4' x_6$,

$C_{x_2} = C$ (x_2 and x_2' do not appear in C)

$C_{x_1} = x_4' x_6$ (x_1 appears positively in C)

$C_{x_4} = \varnothing$ (x_4 appears negatively in C)

(Literal) Cofactor

- Example

  $F = abc' + b'd + cd$

  $F_b = ac' + cd$

  (Just drop *b* everywhere and throw away cubes containing literal *b*')

  Cofactor and disjunction commute!

Shannon Expansion

Let $f : B^n \rightarrow B$

Shannon Expansion:

$$f = x_i \, f_{x_i} + x_i' f_{x_i'}$$

Theorem: F is a cover of f. Then

$$F = x_i \, F_{xi} + x_i' F_{x_i'}$$

We say that f and F are expanded about x_i, and x_i is called the splitting variable

Shannon Expansion

- Example

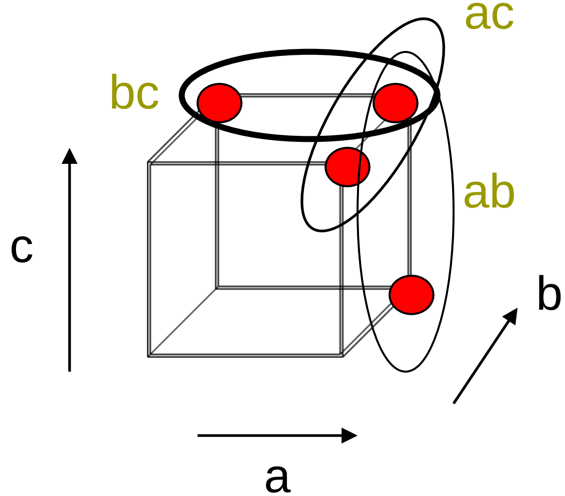
$F = ab + ac + bc$

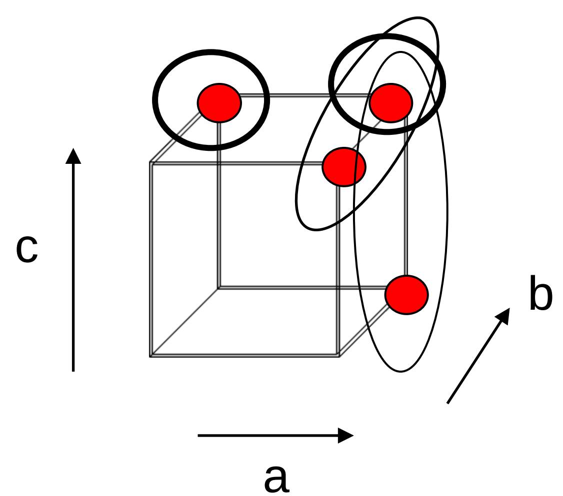
$F = a\,F_a + a'F_{a'}$

$= a\,(b+c+bc)+a'(bc)$

$= ab+ac+abc+a'bc$

Cube bc got split into two cubes





(Cube) Cofactor

- The cofactor f_C of f by a cube C is f with the fixed values indicated by the literals of C
  - E.g., if $C = x_i x_j'$, then $x_i = 1$ and $x_j = 0$
  - For $C = x_1 x_4' x_6$, f_C is just the function f restricted to the subspace where $x_1 = x_6 = 1$ and $x_4 = 0$
    - Note that f_C does not depend on x_1, x_4 or x_6 anymore

      (However, we still consider f_C as a function of all *n* variables, it just happens to be independent of x_1, x_4 and x_6)
  - $x_1 f \neq f_{x_1}$
    - E.g., for $f = ac + a'c$, $a \cdot f_a = a \cdot f = a \cdot c$ and $f_a = c$

(Cube) Cofactor

- The cofactor of the cover F of some function f is the sum of the cofactors of each of the cubes of F

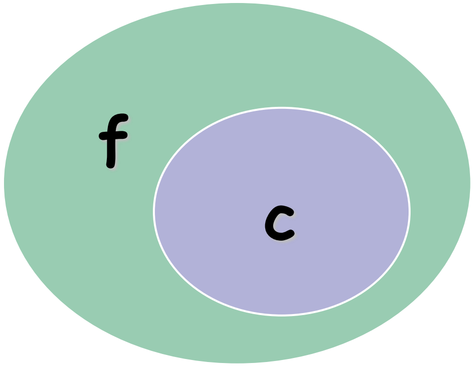
- If $F=\{c_1, c_2,\ldots, c_k\}$ is a cover of f, then $F_c= \{(c_1)_c, (c_2)_c,\ldots, (c_k)_c\}$ is a cover of f_c

Containment vs. Tautology

- A fundamental theorem that connects functional containment and tautology:

  Theorem. Let c be a cube and f a function. Then $c \subseteq f \Leftrightarrow f_c \equiv 1$.

  Proof.
  We use the fact that $xf_x = xf$, and f_x is independent of x.
  $(\Leftarrow)$
  Suppose $f_c \equiv 1$. Then $cf = f_c c = c$. Thus, $c \subseteq f$.
  $(\Rightarrow)$
  Suppose $c \subseteq f$. Then $f+c=f$. In addition, $f_c = (f+c)_c = f_c+1=1$. Thus, $f_c=1$.



Checking of Prime and Irredundancy (Revisited)

Let G be a cover of F = (f,d,r). Let D be a cover for d

- $c_i \in G$ is redundant iff
  $c_i \subseteq (G \backslash \{c_i\}) \cup D$ (1)

  (Let $G^i \equiv G \backslash \{c_i\} \cup D$. Since $c_i \subseteq G^i$ and $f \subseteq G \subseteq f+d$, then $c_i \subseteq c_i f + c_i d$ and $c_i f \subseteq G \backslash \{c_i\}$. Thus $f \subseteq G \backslash \{c_i\}$.)

- A literal $l \in c_i$ is prime if $(c_i \backslash \{ l \})$ ($= (c_i)_l$) is not an implicant of F
- A cube c_i is a prime of F iff all literals $l \in c_i$ are prime
  Literal $l \in c_i$ is not prime $\Leftrightarrow (c_i)_l \subseteq f+d$ (2)

Note: Both tests (1) and (2) can be checked by tautology (explained):

- $(G^i)_{c_i} \equiv 1$ (implies c_i redundant)
- $(f \cup d)_{(c_i)_l} \equiv 1$ (implies l not prime)
  The above two cofactors are with respect to cubes instead of literals

Generalized Cofactor

- Definition. Let *f, g* be completely specified functions. The generalized cofactor of *f* with respect to *g* is the incompletely specified function:

$$co(f,g) = (f \cdot g, \overline{g}, \overline{f} \cdot g)$$

- Definition. Let $\mathfrak{I}$ = (*f, d, r*) and *g* be given. Then

$$co(\mathfrak{I},g) = (f \cdot g, d + \overline{g}, r \cdot g)$$

Shannon vs. Generalized Cofactor

- Let $g = x_i$. Shannon cofactor is

$$f_{x_i}(x_1, x_2, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$$

- Generalized cofactor with respect to $g=x_i$ is

$$co(f, x_i) = (f \cdot x_i, \bar{x}_i, \bar{f} \cdot x_i)$$

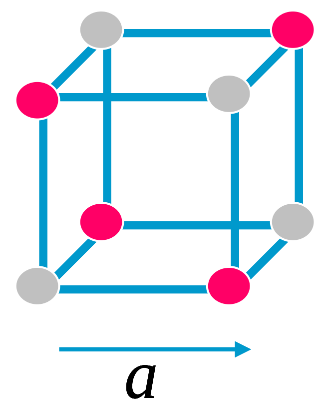
- Note that

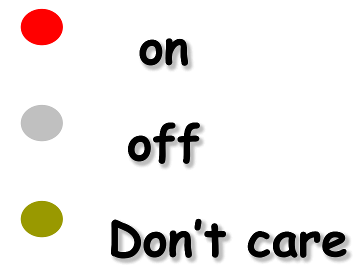
$$f \cdot x_i \subseteq f_{x_i} \subseteq f \cdot x_i + \bar{x}_i = f + \bar{x}_i$$

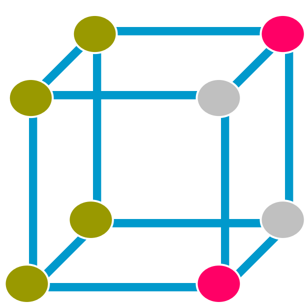
In fact f_{x_i} is the unique cover of $co(f, x_i)$ independent of the variable x_i .

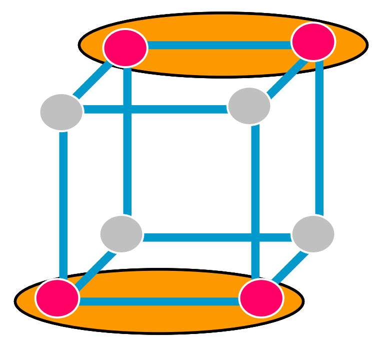
Shannon vs. Generalized Cofactor

$$f = abc + a\bar{b}\bar{c} + \bar{a}\bar{b}c + \bar{a}b\bar{c}$$





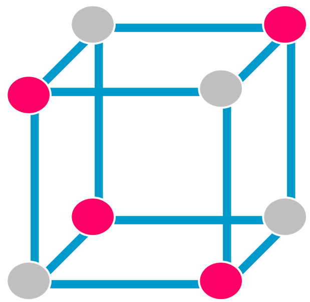




$$co(f, a) = (f \cdot a, \bar{a}, \bar{f} \cdot a)$$

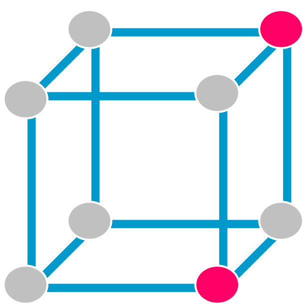
$$f_a = bc + \bar{b}\bar{c}$$

Shannon vs. Generalized Cofactor



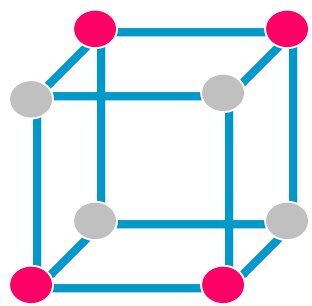
f

$$co(f,a) = (f \cdot a, \bar{a}, \bar{f} \cdot a)$$



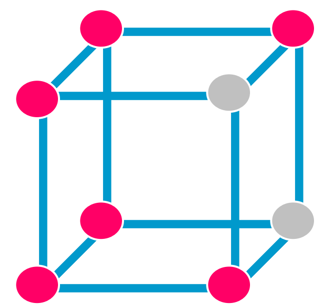
$f \cdot a$

$\subseteq$



f_a

$\subseteq$



$fa + \bar{a}$

So $\quad f \cdot a \subseteq f_a \subseteq f + \bar{a}$

Shannon vs. Generalized Cofactor

Shannon Cofactor

$$x \cdot f_x + \bar{x} \cdot f_{\bar{x}} = f$$

$$\left(f_x\right)_y = f_{xy}$$

$$\left(f \cdot g\right)_y = f_y \cdot g_y$$

$$\left(\bar{f}\right)_x = \overline{\left(f_x\right)}$$

Generalized Cofactor

$$f = g \cdot co(f, g) + \bar{g} \cdot co(f, \bar{g})$$

$$co(co(f, g), h) = co(f, gh)$$

$$co(f \cdot g, h) = co(f, h) \cdot co(g, h)$$

$$co(\bar{f}, g) = \overline{co(f, g)}$$

We will get back to the use of generalized cofactor later

Data Structure for SOP Manipulation

most of the following slides are by courtesy of Andreas Kuehlmann

Operation on Cube Lists

- AND operation:
  - take two lists of cubes
  - compute pair-wise AND between individual cubes and put result on new list
  - represent cubes in computer words
  - implement set operations as bit-vector operations

```
Algorithm AND(List_of_Cubes C1,List_of_Cubes C2) {
  C = ∅
  foreach c1 ∈ C1 {
    foreach c2 ∈ C2 {
      c = c1 & c2
      C = C ∪ c
    }
  }
  return C
}
```

Operation on Cube Lists

- OR operation:
  - take two lists of cubes
  - computes union of both lists
- Naive implementation:

```
Algorithm OR(List_of_Cubes C1, List_of_Cubes C2) {
  return C1 ∪ C2
}
```

- On-the-fly optimizations:
  - remove cubes that are completely covered by other cubes
    - complexity is $O(m^2)$; m is length of list
  - conjoin adjacent cubes (consensus operation)
  - remove redundant cubes?
    - coNP-complete
    - too expensive for non-orthogonal lists of cubes

Operation on Cube Lists

- Simple trick:
  - keep cubes in lists orthogonal
    - check for redundancy becomes $O(m^2)$
    - but lists become significantly larger (worst case: exponential)
  - Example

01-0 1-01	OR	0-1- 1-11	=	01-0 1-01 001- 0111 1-11

Operation on Cube Lists

- Adding cubes to orthogonal list:

```
Algorithm ADD_CUBE(List_of_Cubes C, Cube c) {
  if(C = ∅) return {c}
  c'  = TOP(C)
  Cres = c-c'  /* chopping off minterms may result in multiple cubes */
  foreach cres ∈ Cres {
    C = ADD_CUBE(C\{c'},cres) ∪ {c'}
  }
  return C
}
```

- How can the above procedure be further improved?
- What about the AND operation, does it gain from orthogonal cube lists?

Operation on Cube Lists

- Naive implementation of COMPLEMENT operation
  - apply De'Morgan's law to SOP
  - complement each cube and use AND operation

  Example

Input		non-orth.		orthogonal
01-10	=>	1----	=>	1----
		-0---		00---
		---0-		01-0-
		----1		01-11

- Naive implementation of TAUTOLOGY check
  - complement function using the COMPLEMENT operator and check for emptiness
- We will show that we can do better than that!

Tautology Checking

- Let A be an orthogonal cover matrix, and all cubes of A be pair-wise distinguished by at least two literals (this can be achieved by an on-the-fly merge of cube pairs that are distinguished by only one literal)

  Does the following conjecture hold?

  $A \equiv 1 \quad \Leftrightarrow \quad$ A has a row of all "-"s ?

  This would dramatically simplify the tautology check!

Tautology Checking

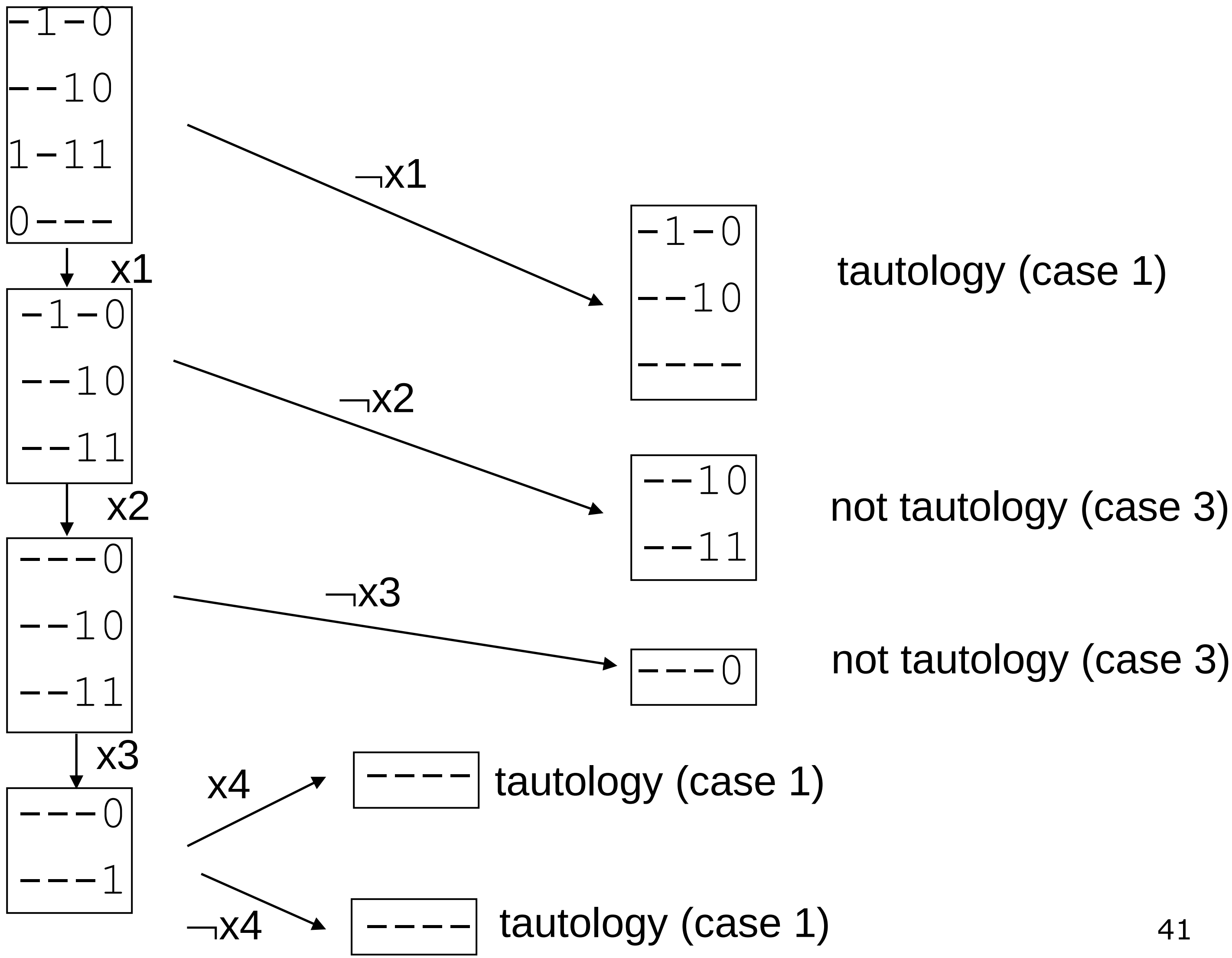
```
Algorithm CHECK_TAUTOLOGY(List_of_Cubes C) {
  if(C == ∅)        return FALSE;
  if(C == {-...-})return TRUE; // cube with all '-'
  xi = SELECT_VARIABLE(C)
  C0 = COFACTOR(C,¬Xi)
  if(CHECK_TAUTOLOGY(C0) == FALSE) {
     print xi = 0
     return FALSE;
  }
  C1 = COFACTOR(C,Xi)
  if(CHECK_TAUTOLOGY(C1) == FALSE) {
     print xi = 1
     return FALSE;
  }
  return TRUE;
}
```

Tautology Checking

- Implementation tricks
  - Variable ordering:
    - pick variable that minimizes the two sub-cases (“-”s get replicated into both cases)
  - Quick decision at leaf:
    - return TRUE if C contains at least one complete “-” cube among others (case 1)
    - return FALSE if number of minterms in onset is $< 2^n$ (case 2)
    - return FALSE if C contains same literal in every cube (case 3)

Tautology Checking

Example



Special Functions

- **Definition.** A function $f : B^n \rightarrow B$ is symmetric with respect to variables x_i and x_j iff
  $f(x_1,\ldots,x_i,\ldots,x_j,\ldots,x_n) = f(x_1,\ldots,x_j,\ldots,x_i,\ldots,x_n)$

- **Definition.** A function $f : B^n \rightarrow B$ is totally symmetric iff any permutation of the variables in f does not change the function

Symmetry can be exploited in searching BDD since

$$f_{x_i \overline{x}_j} = f_{\overline{x}_i x_j}$$

- can skip one of four sub-cases
- used in automatic variable ordering for BDDs

Special Functions

- Definition. A function $f : B^n \to B$ is positive unate in variable x_i iff

$$f_{\bar{x}_i} \subseteq f_{x_i}$$

  - This is equivalent to monotone increasing in x_i:

$$f(m^-) \leq f(m^+)$$

    for all min-term pairs (m^-, m^+) where

$$m_j^- = m_j^+, j \neq i$$

$$m_i^- = 0$$

$$m_i^+ = 1$$

- Example
  (1001, 1011) with $i = 3$

Special Functions

- Similarly for negative unate $f_{x_i} \subseteq f_{\bar{x}_i}$

  monotone decreasing $f(m^-) \geq f(m^+)$

- A function is unate in x_i if it is positive unate or negative unate in x_i

- Definition. A function is unate if it is unate in each variable

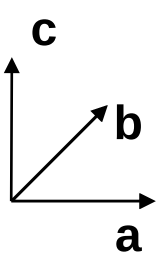
- Definition. A cover F is positive unate in x_i iff $\bar{x}_i \notin c_j$ for all cubes $c_j \in F$

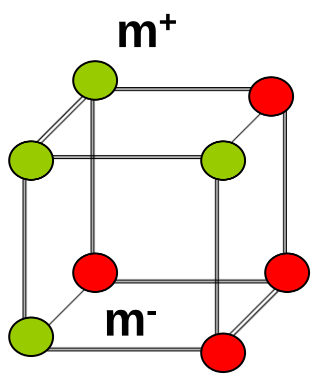
- Note that a cover of a unate function is not necessarily unate! (However, there exists a unate cover for a unate function.)

Special Functions

- Example

$$f = ab + b\bar{c} + a\bar{c}$$





positive unate in a,b
negative unate in c

$f(m^-)=1 \geq f(m^+)=0$

Unate Recursive Paradigm

- Key pruning technique is based on exploiting the properties of unate functions
  - based on the fact that unate leaf cases can be solved efficiently

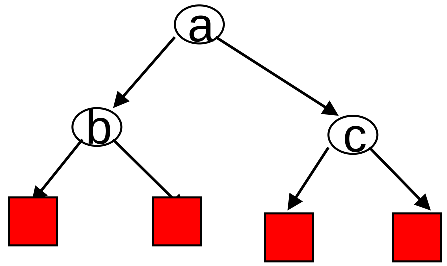
- New case splitting heuristic
  - splitting variable is chosen so that the functions at lower nodes of the recursion tree become unate

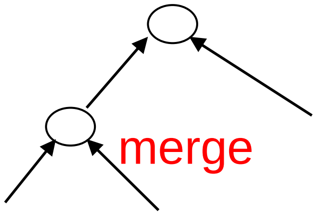
Unate Recursive Paradigm

- Unate covers F have many extraordinary properties:
  - If a **prime** cover F is minimal with respect to single-cube containment, all of its cubes are essential primes
    - In this case F is the unique minimum cube representation of its logic function
  - A unate cover represents a tautology iff it contains a cube with no literals, i.e., a single tautologous cube
- This type of implicit enumeration applies to many sub-problems (prime generation, reduction, complementation, etc.). Hence, we refer to it as the Unate Recursive Paradigm.

Unate Recursive Paradigm

1. Create cofactoring tree stopping at unate covers
   - choose, at each node, the "most binate" variable for splitting
   - iterate until no binate variable left (unate leaf)
2. "Operate" on the unate cover at each leaf to obtain the result for that leaf. Return the result
3. At each non-leaf node, merge (appropriately) the results of the two children.





- Main idea: "Operation" on unate leaf is computationally less complex
- Operations: complement, simplify, tautology, prime generation, ...

Unate Recursive Paradigm

- Binate select heuristic
  - Tautology checking and other programs based on the unate recursive paradigm use a heuristic called BINATE_SELECT to choose the splitting variable in recursive Shannon expansion
    - The idea is, for a given cover F, choose the variable which occurs, both positively and negatively, most often in the cubes of F

Unate Recursive Paradigm

- Binate select heuristic
  - Example

    Unate and non-unate covers:

    G = ac+cd'

a	b	c	d
1	–	1	–
–	–	1	0

    is unate

    F = ac+c'd+bcd'

a	b	c	d
1	–	1	–
–	–	0	1
–	1	1	0

    is not unate

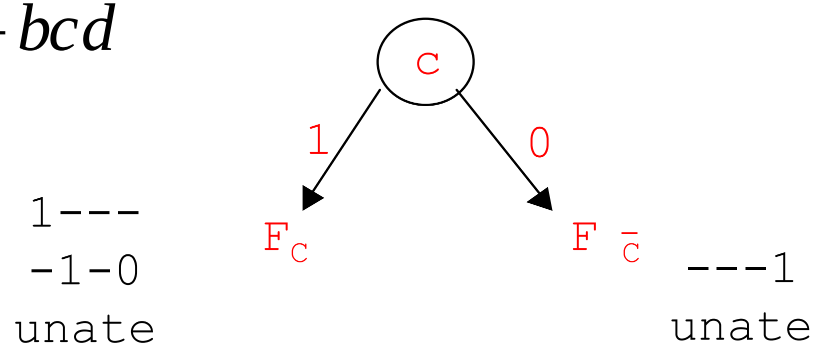
    $\Rightarrow$ Choose c for splitting!

  - Binate variables of a cover are those with both 1's and 0's in the corresponding column
  - In the unate recursive paradigm, the BINATE_SELECT heuristic chooses a (most) binate variable for splitting, which is thus eliminated from the sub-covers

Unate Recursive Paradigm

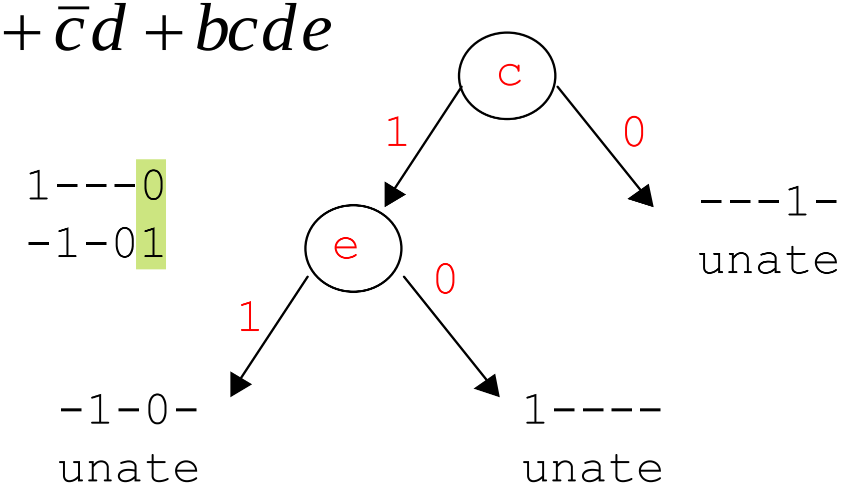
- Example

$$f = ac + \bar{c}d + bc\bar{d}$$



$$F = \begin{array}{cccc} 1 & - & 1 & - \\ - & - & 0 & 1 \\ - & 1 & 1 & 0 \end{array}$$

$$f = ac\bar{e} + \bar{c}d + bc\bar{d}e$$



$$F = \begin{array}{ccccc} 1 & - & 1 & - & 0 \\ - & - & 0 & 1 & - \\ - & 1 & 1 & 0 & 1 \end{array}$$

Unate Recursive Paradigm
Unate Reduction

- Let $F(x)$ be a cover. Let (a,c) be a *partition* of the variables x, and let

$$F = \left[\begin{array}{c:c} A & C \\ \hdashline T & F^* \end{array}\right]$$

where
1. the columns of A (a unate submatrix) correspond to variables a of x
2. T is a matrix of all “-”s

- Theorem. Assume $A \neq 1$. Then $F \equiv 1 \Leftrightarrow F^* \equiv 1$

Unate Recursive Paradigm
Unate Reduction

- Example

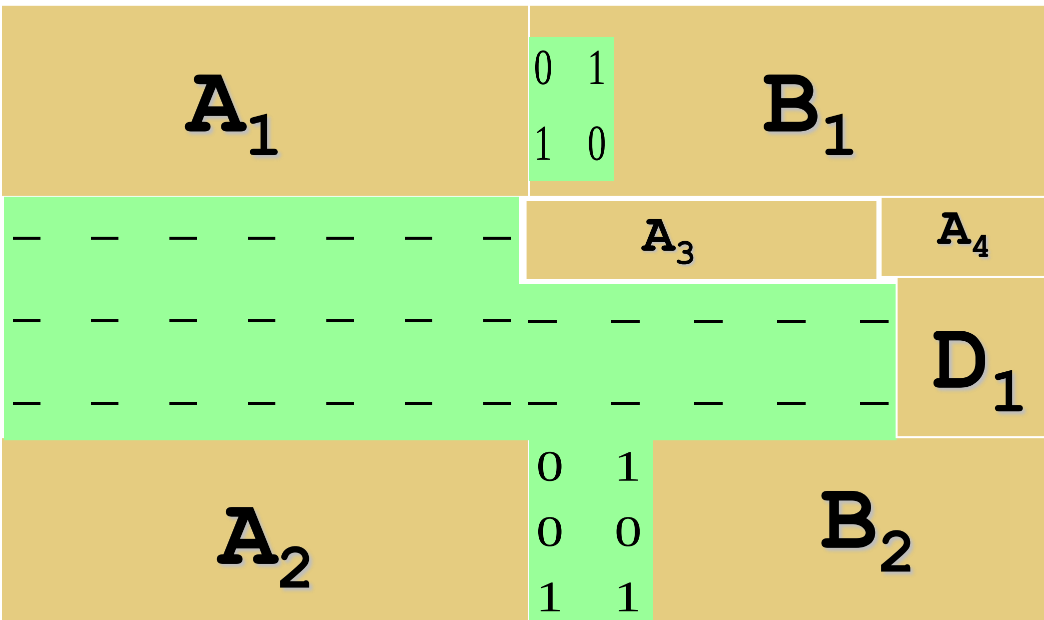
$$F = \left[\begin{array}{c:c} A & C \\ \hdashline T & F^{*} \end{array}\right]$$

$$\begin{array}{cccc} 1 & - & 0 & 1 \\ - & 1 & 1 & 0 \\ - & - & 1 & 1 \\ - & - & - & 0 \\ - & - & - & 1 \end{array} \longrightarrow \begin{array}{cc} 1 & 1 \\ - & 0 \\ - & 1 \end{array}$$

We pick for the partitioning unate variables because it is easy to decide that $A \neq 1$

Unate Recursive Paradigm
Unate Reduction

- Example



  - Assume A_1 and A_2 are unate and have no row of all “-”s.
  - Note that A_3 and A_4 are unate (single-row sub-matrices)
  - Consequently only have to look at D_1 to test if this is a tautology

Unate Recursive Paradigm
Unate Reduction

- Theorem:

$$F = \left[\begin{array}{c:c} A & C \\ \hdashline T & F^* \end{array}\right]$$

Let A be a non-tautological unate matrix ($A \neq 1$) and T is a matrix of all -'s. Then $F \equiv 1 \Leftrightarrow F^* \equiv 1$.

- Proof:

If part: Assume $F^* \equiv 1$. Then we can replace F^* by all -'s. Then last row of F becomes a row of all "-"s, so tautology.

Unate Recursive Paradigm
Unate Reduction

- Proof (cont'd):

  Only if part: Assume $F^* \neq 1$. Then there is a minterm m_2 (in c variables) such that $F^*_{m_2} = 0$ (cofactor in cube), i.e. m_2 is not covered by F^*. Similarly, m_1 (in a variables) exists where $A_{m_1} = 0$, i.e. m_1 is not covered by A. Now the minterm m_1m_2 (in the full variable set) satisfies $F_{m_1m_2} = 0$. Since m_1m_2 is not covered by F, $F \neq 1$.

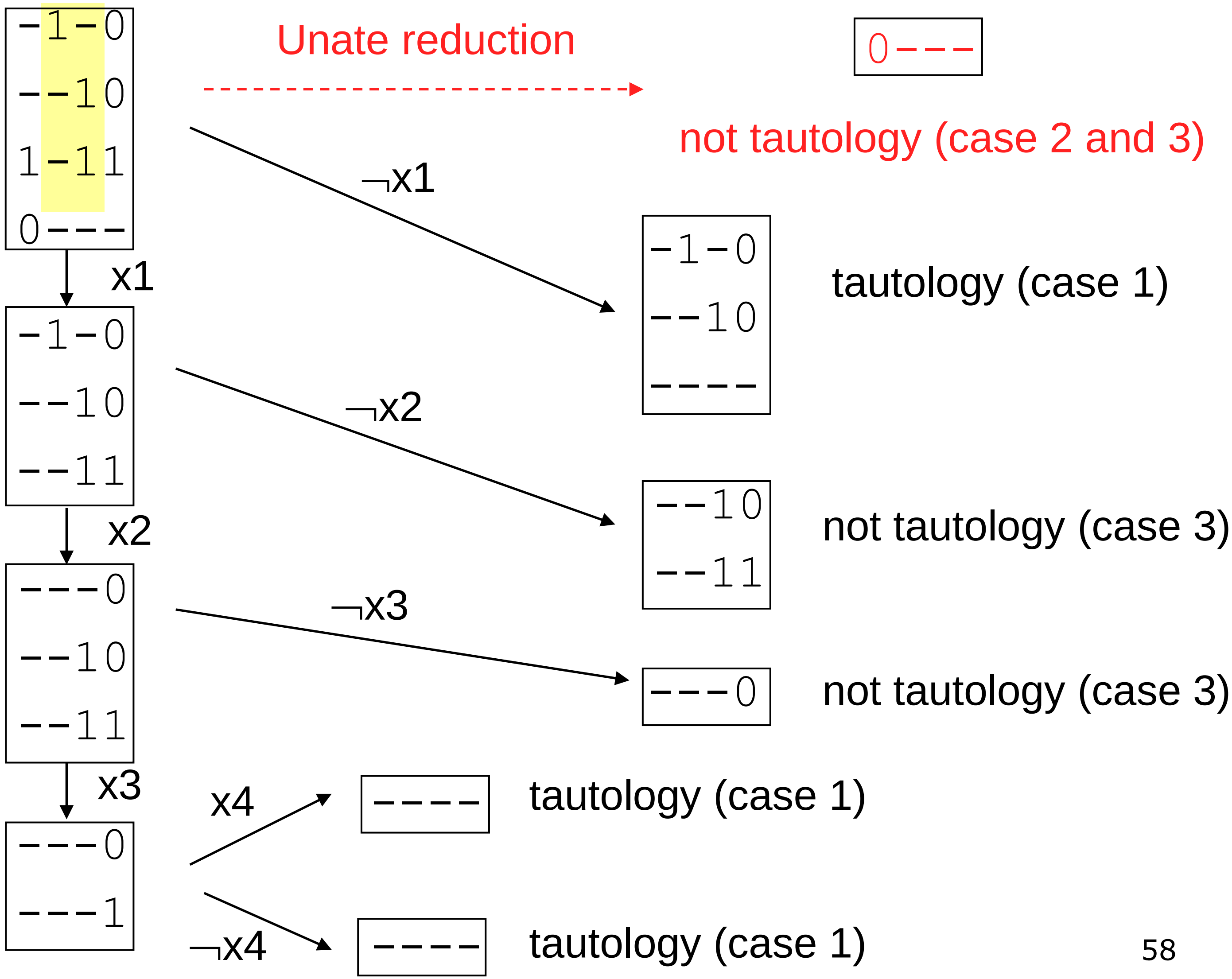
Unate Recursive Paradigm
Application – Tautology Checking

- Improved tautology check

```
Algorithm CHECK_TAUTOLOGY(List_of_Cubes C) {
  if(C == ∅)              return FALSE;
  if(C == {-...-})        return TRUE; // cube with all '-'
  C = UNATE_REDUCTION(C)
  xi = BINATE_SELECT(C)
  C0 = COFACTOR(C,¬xi)
  if(CHECK_TAUTOLOGY(C0) == FALSE) {
     return FALSE;
  }
  C1 = COFACTOR(C,xi)
  if(CHECK_TAUTOLOGY(C1) == FALSE) {
     return FALSE;
  }
  return TRUE;
}
```

Unate Recursive Paradigm
Application – Tautology Checking

- Example



Unate Recursive Paradigm Application – Complement

- We have shown how tautology check (SAT check) can be implemented recursively using the Binary Decision Tree
- Similarly, we can implement Boolean operations recursively, e.g. the COMPLEMENT operation:
- Theorem. $$\overline{f} = x \cdot \overline{f_x} + \overline{x} \cdot \overline{f_{\overline{x}}}$$
- Proof.

$$g = x \cdot \overline{f_x} + \overline{x} \cdot \overline{f_{\overline{x}}}$$

$$f = x \cdot f_x + \overline{x} \cdot f_{\overline{x}}$$

$$\left.\begin{aligned} f \cdot g &= 0 \\ f + g &= 1 \end{aligned}\right\} \Rightarrow g = \overline{f}$$

Unate Recursive Paradigm
Application – Complement

- Complement operation on cube list

```
Algorithm COMPLEMENT(List_of_Cubes C) {
  if(C contains single cube c) {
    Cres = complement_cube(c)   // generate one cube per
    return Cres                 // literal l in c with ¬l
  }
  else {
    xi = SELECT_VARIABLE(C)
    C0 = COMPLEMENT(COFACTOR(C,¬xi)) ∧ ¬xi
    C1 = COMPLEMENT(COFACTOR(C,xi))  ∧ xi
    return OR(C0,C1)
  }
}
```

Unate Recursive Paradigm Application – Complement

- Efficient complement of a unate cover
- Idea:
  - variables appear only in one polarity on the original cover
    (ab + bc + ac)' = (a'+b')(b'+c')(a'+c')
  - when multiplied out, a number of products are redundant
    a'b'a' + a'b'c' + a'c'a' + a'c'c'+ b'b'a' + b'b'c' + b'c'a' + b'c'c' =
    a'b' + a'c' + b'c'
  - we just need to look at the combinations for which the variables cover all original cubes (see the following example)
    - this works independent of the polarity of the variables because of symmetry to the (1,1,1,…,1) case (see the following example)

Unate Recursive Paradigm Application – Complement

- Map (unate) cover matrix F into Boolean matrix B

F

a	b	c	d	e
–	1	–	0	–
–	–	0	0	1
1	1	–	–	1
1	–	0	–	1

$\longrightarrow$

B

a	b	c	d	e
0	1	0	1	0
0	0	1	1	1
1	1	0	0	1
1	0	1	0	1

convert: "0","1" in F to "1" in B (literal is present)
"-" in F to "0" in B (literal is not present)

Unate Recursive Paradigm Application – Complement

- Find all minimal column covers of B.
  - A *column cover* is a set of columns J such that for each row i, $\exists j \in J$ such that $B_{ij} = 1$

- Example

  {1,4} is a *minimal column cover* for matrix B

$$\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \end{bmatrix} \end{array} \longrightarrow \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

All rows “covered” by at least one 1

Unate Recursive Paradigm Application – Complement

- For each minimal column cover create a cube with opposite column literal from F
- Example

  By selecting a column cover {1,4}, a'd is a cube of f'

1	2	3	4	5
a	b	c	d	e
–	1	–	0	–
–	–	0	0	1
1	1	–	–	1
1	–	0	–	1

→

1	2	3	4	5
a	b	c	d	e
0	1	0	1	0
0	0	1	1	1
1	1	0	0	1
1	0	1	0	1

Unate Recursive Paradigm
Application – Complement

- The set of all minimal column covers = cover of $\overline{f}$
- Example

a	b	c	d	e
–	1	–	0	–
–	–	0	0	1
1	1	–	–	1
1	–	0	–	1

$\longrightarrow$

a	b	c	d	e
0	1	0	1	0
0	0	1	1	1
1	1	0	0	1
1	0	1	0	1

  - {(1,4), (2,3), (2,5), (4,5)} is the set of all minimal covers. This translates into:

$$\overline{f} = \overline{a}d + \overline{b}c + \overline{b}\overline{e} + d\overline{e}$$

Unate Recursive Paradigm Application – Complement

- Theorem (unate complement theorem):
  Let F be a unate cover of f. The set of cubes associated with the minimal column covers of B is a cube cover of $\bar{f}$.

- Proof:
  We first show that any such cube c generated is in the offset of f, by showing that the cube c is orthogonal with any cube of F.
  - Note, the literals of c are the complemented literals of F.
    - Since F is a unate cover, the literals of F are just the union of the literals of each cube of F).
  - For each cube $m_i \in F$, $\exists j \in J$ such that $B_{ij}=1$.
    - J is the column cover associated with c.
  - Thus, $(m_i)_j = x_j \Rightarrow c_j = \bar{x}_j$ and $(m_i)_j = \bar{x}_j \Rightarrow c_j = x_j$. Thus $m_i c = \emptyset$. Thus $c \subseteq \bar{f}$.

Unate Recursive Paradigm Application – Complement

- Proof (cont'd):

  We now show that any minterm $m \in \bar{f}$ is contained in some cube c generated:
  - First, m must be orthogonal to each cube of F.
    - For each row of F, there is at least one literal of m that conflicts with that row.
  - The union of all columns (literals) where this happens is a column cover of B
  - Hence this union contains at least one minimal cover and the associated cube contains m.

Unate Recursive Paradigm Application – Complement

- The **unate covering problem** finds a minimum column cover for a given Boolean matrix B
  - Unate complementation is one application based on the unate covering problem

- Unate Covering Problem:
  Given a matrix B, with $B_{ij} \in \{0,1\}$, find x, with $x_i \in \{0,1\}$, such that $Bx \geq 1$ (componentwise inequality) and $\Sigma_j\ x_j$ is minimized

  - Sometimes we want to minimize

    $$\Sigma_j\ c_j x_j$$

    where c_j is a cost associated with column j