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

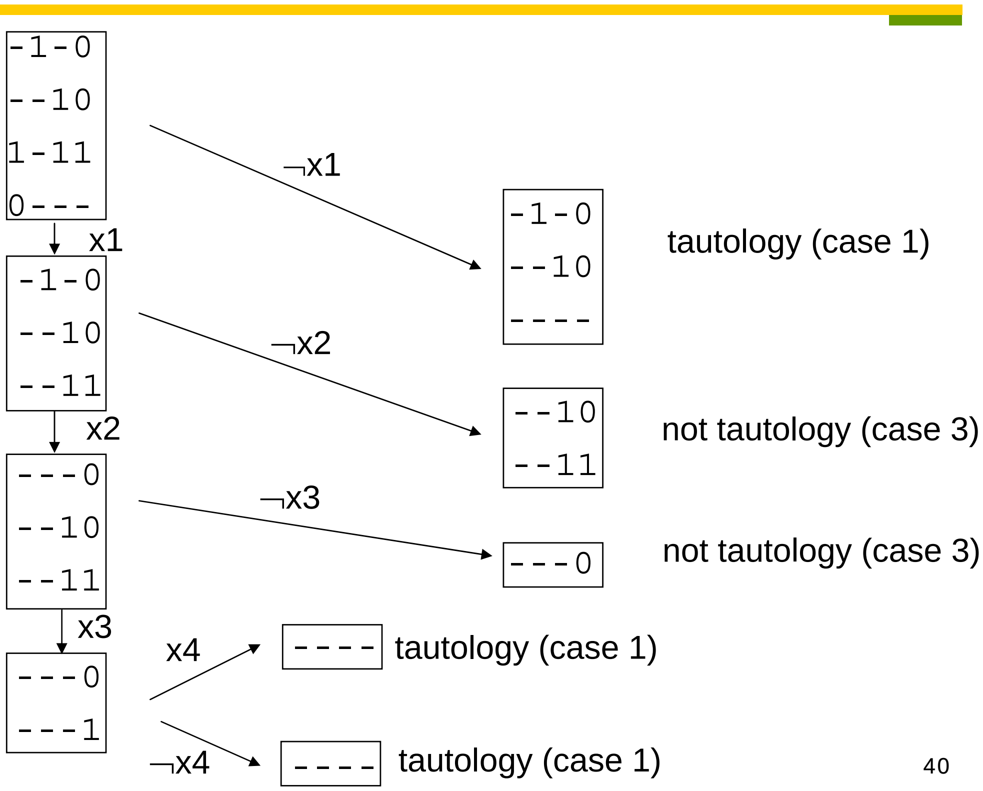
```
Algorithm CHECK_TAUTOLOGY(List_of_Cubes C) {
  if(C == ∅)      return FALSE;
  if(C == {-...-})return TRUE; // cube with all '-'
  xi = SELECT_VARIABLE(C)
  C0 = COFACTOR(C,¬Xi)
  if(CHECK_TAUTOLOGY(C0) == FALSE) {
     print xi = 0
     return FALSE;
  }
  C1 = COFACTOR(C,Xi)
  if(CHECK_TAUTOLOGY(C1)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

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

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

Special Functions

- **Definition.** A function $f : B^n \rightarrow B$ 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

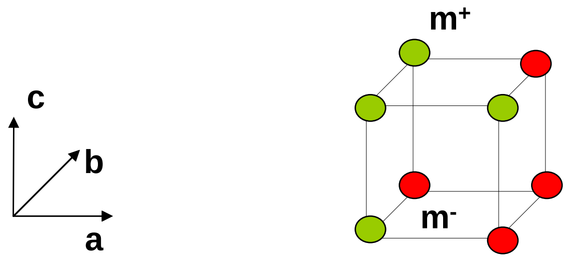
- Definition. A cover F is positive unate in x_i iff $\bar{x}_i \notin c_j$ for all cubes $c_j \in F$

- 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

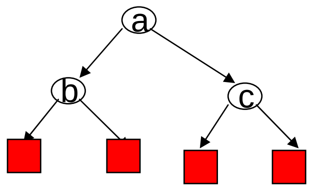
- Key pruning technique is based on exploiting the properties of unate functions
  - based on the fact that unate leaf cases can be solved efficiently
- New case splitting heuristic
  - splitting variable is chosen so that the functions at lower nodes of the recursion tree become unate

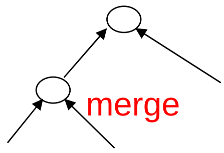
Unate Recursive Paradigm

- 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 the 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 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

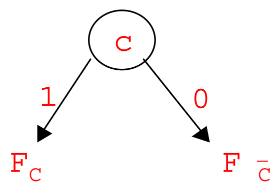
$\Rightarrow$ Choose c for splitting!

  - 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

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



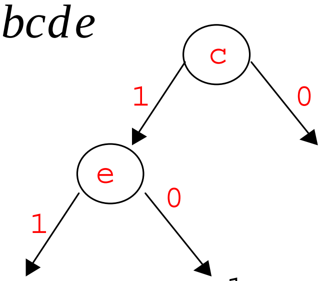
1---
-1-0
unate

---1
unate

F=

1	-	1	-
-	-	0	1
-	1	1	0

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



1---0
-1-01

---1-
unate

-1-0-
unate

1----
unate

F=

1	-	1	-	0
-	-	0	1	-
-	1	1	0	1

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

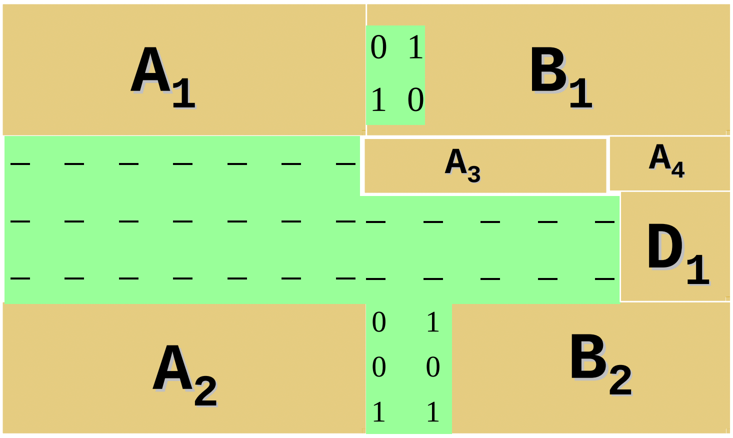
$$F = \left[\begin{array}{c:c} A & C \\ \hdashline T & F^* \end{array}\right]$$

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

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* ≡ 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 $m_1 m_2$ (in the full variable set) satisfies $F_{m_1 m_2} = 0$. Since $m_1 m_2$ is not covered by F, $F \neq 1$.

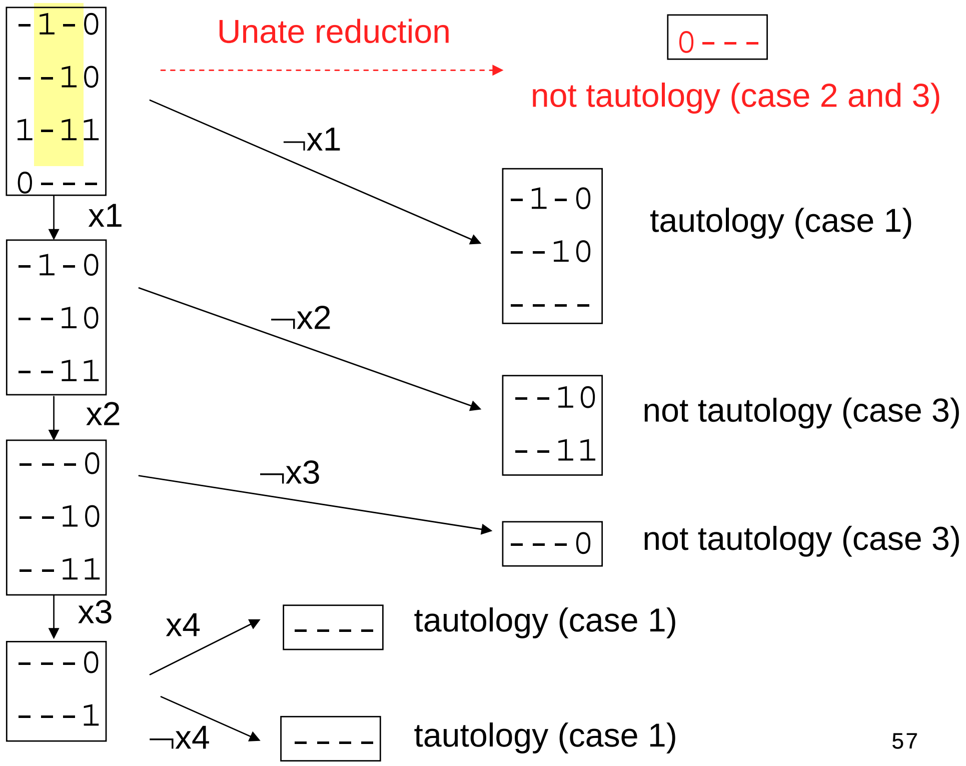
Unate Recursive Paradigm
Application – Tautology Checking

- 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

→

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

1	2	3	4	5
0	1	0	1	0
0	0	1	1	1
1	1	0	0	1
1	0	1	0	1

→

1
1
1
1

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

$\longrightarrow$

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 $\overline{f}$.

- 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, $(m_i)_j = x_j \Rightarrow c_j = \overline{x}_j$ and $(m_i)_j = \overline{x}_j \Rightarrow c_j = x_j$. Thus $m_i c = \emptyset$. Thus $c \subseteq \overline{f}$.

Unate Recursive Paradigm Application – Complement

- Proof (cont'd):
  We now show that any minterm $m \in \overline{f}$ 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