

Logic Design

Binary Decision Diagrams

*Courtesy RK Brayton (UCB)
and A Kuehlmann (Cadence)*

Representing Boolean functions

- Fundamental trade-off
 - canonical data structures
 - data structure uniquely represents function
 - Tautology decision procedure is trivial (e.g., just pointer comparison)
 - example: truth tables, Binary Decision Diagrams
 - size of data structure is in general exponential
 - noncanonical data structures
 - covers, POS, formulas, logic circuits
 - systematic search for satisfying assignment
 - size of data structure is often small
 - Tautology checking computationally expensive

ROBDDs

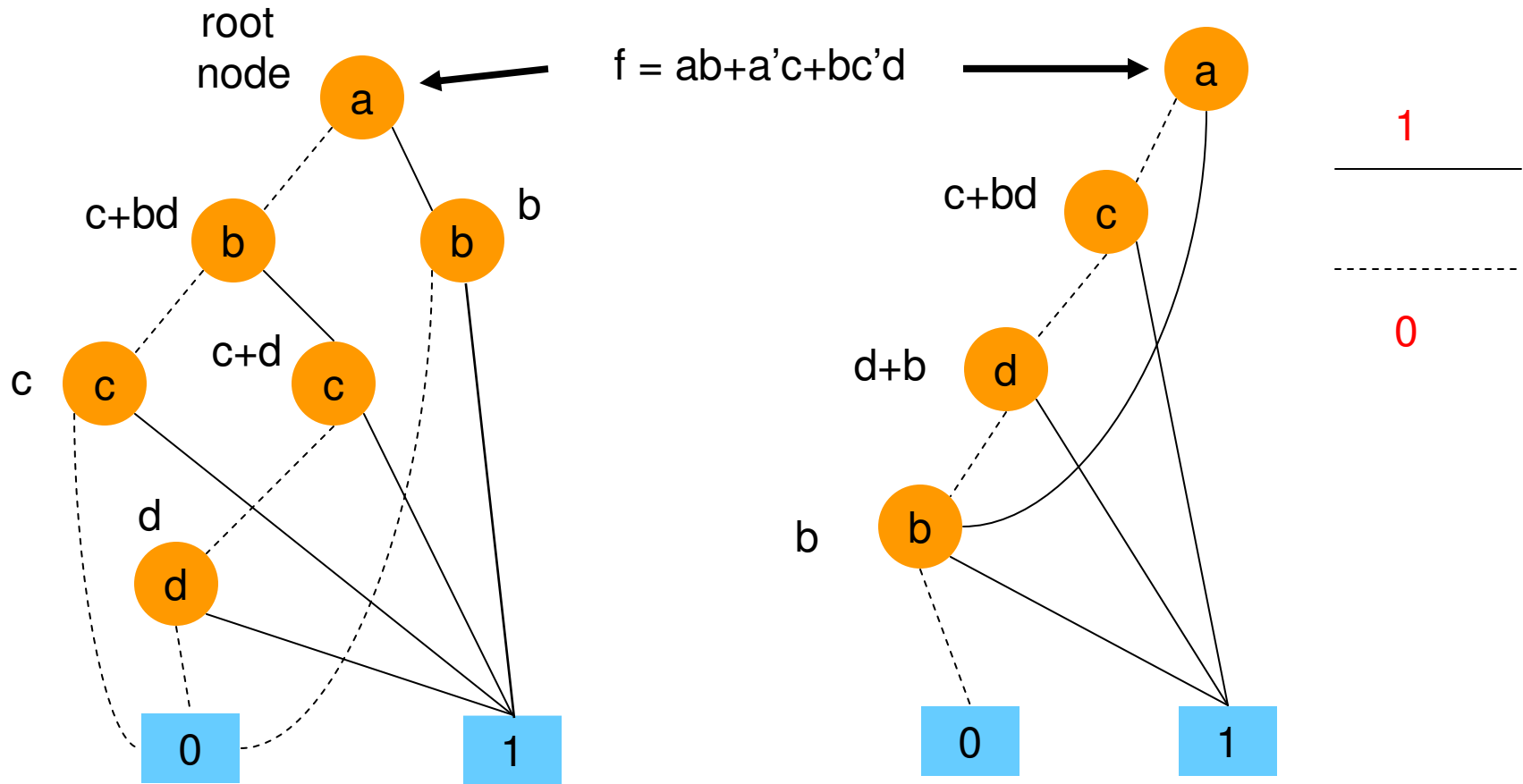
- **General idea:** Representation of a logic function as graph (DAG)
 - use Shannon decomposition to build a decision tree representation
 - Similar to what we saw in 2-level minimization
 - difference: instead of exploring sub-cases by enumerating them in time store sub-cases in memory
 - **Key to making efficient** : two hashing mechanisms:
 - unique table: find identical sub-cases and avoid replication
 - computed table: reduce redundant computation of sub-cases
- Represent of a logic function as graph
 - many logic functions can be represented compactly - usually better than SOPs
- Many logic operations can be performed efficiently on BDDs
 - usually linear in size of result - tautology and complement are constant time
- Size of BDD critically dependent on variable ordering

ROBDDs

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

$$X_{i_1} < X_{i_2} < X_{i_3} < \dots < X_{i_n}$$

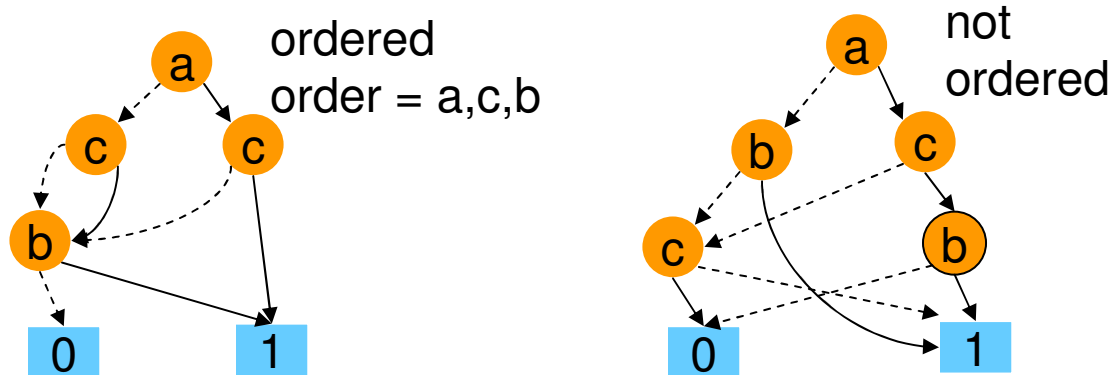
Example



Two different orderings, same function.

ROBDD

Ordered BDD (OBDD) Input variables are ordered - each path from root to sink visits nodes with labels (variables) in ascending order.



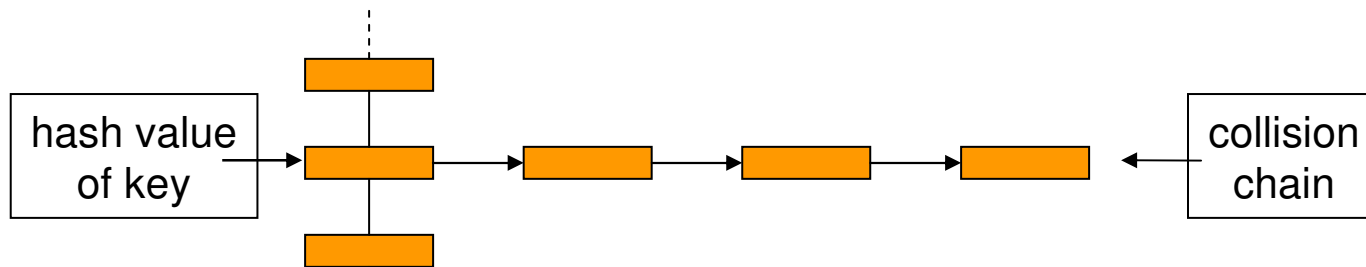
Reduced Ordered BDD (ROBDD) - reduction rules:

1. if the two children of a node are the **same**, the node is eliminated:
 $f = vf + \bar{v}f$
 2. two nodes have **isomorphic** graphs \Rightarrow replace by one of them
- These two rules make it so that each node represents a distinct logic function.

Efficient Implementation of BDD's

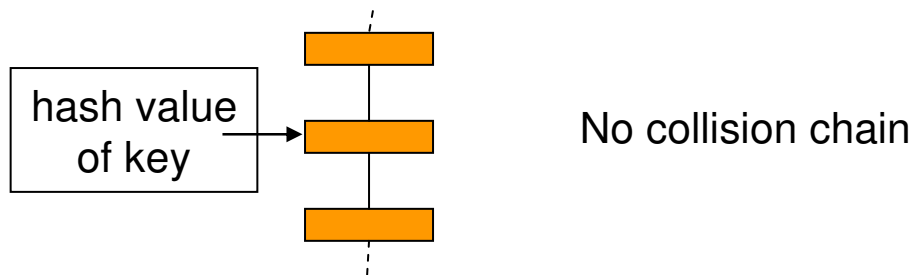
Unique Table:

- avoids duplication of existing nodes
 - Hash-Table: $\text{hash-function}(\text{key}) = \text{value}$
 - identical to the use of a hash-table in AND/INVERTER circuits



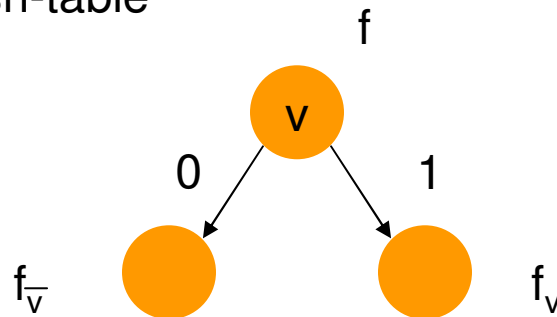
Computed Table:

- avoids re-computation of existing results



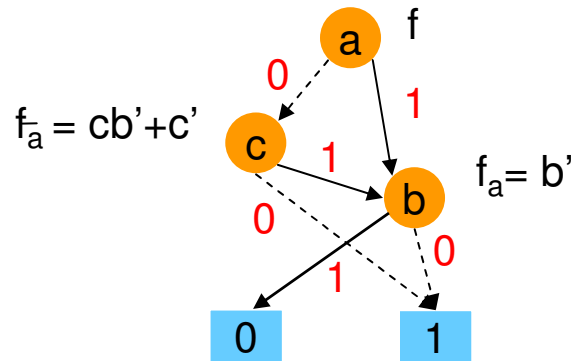
Efficient Implementation of BDD's

- BDDs is a compressed Shannon co-factoring tree:
 - $f = v f_v + \bar{v} f_{\bar{v}}$
 - leafs are constants “0” and “1”
- Three components make ROBDDs canonical (Proof: Bryant 1986):
 - unique nodes for constant “0” and “1”
 - identical order of case splitting variables along each paths
 - hash table that ensures:
 - $(\text{node}(f_v) = \text{node}(g_v)) \wedge (\text{node}(f_{\bar{v}}) = \text{node}(g_{\bar{v}})) \Rightarrow \text{node}(f) = \text{node}(g)$
 - provides recursive argument that node(f) is unique when using the unique hash-table



Onset is Given by all Paths to “1”

$F = b' + a'c' = ab' + a'cb' + a'c'$ all paths to the 1 node



Notes:

- By tracing paths to the 1 node, we get a **cover** of pair wise **disjoint** cubes.
- The power of the BDD representation is that it does **not** explicitly enumerate all paths; rather it represents paths by a graph whose size is measured by its nodes and not paths.
- A DAG can represent an **exponential number of paths** with a linear number of nodes.
- BDDs can be used to efficiently represent sets
 - interpret elements of the onset as elements of the set
 - f is called the **characteristic function** of that set

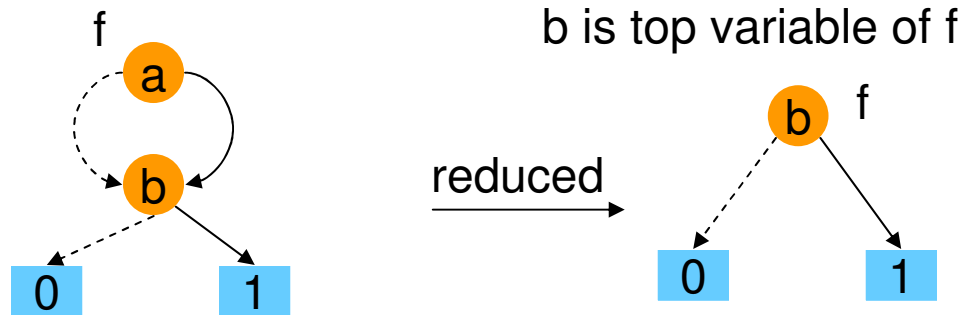
Implementation

Variables are **totally ordered**: If $v < w$ then v occurs “higher” up in the ROBDD
Top variable of a function f is a variable associated with its **root node**.

Example: $f = ab + a'bc + a'bc'$. Order is $(a < b < c)$.

$$f_a = b, f_{a^-} = b$$

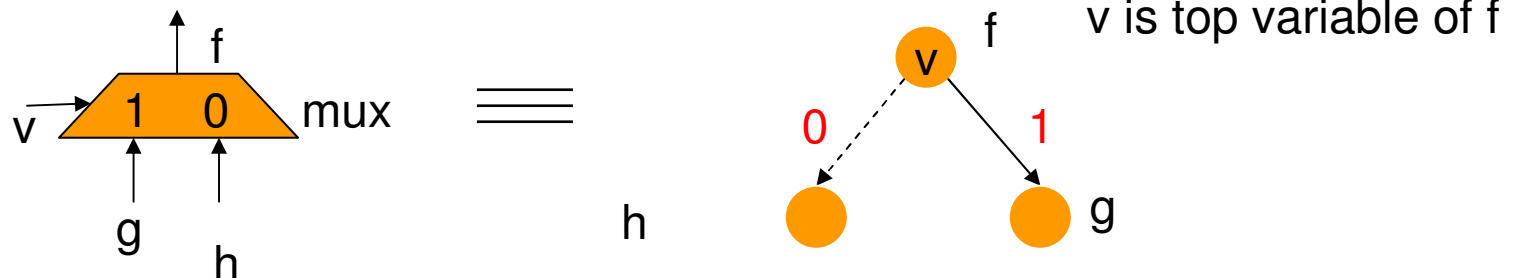
f does not depend on a ,
 since $f_a = f_{a^-}$.



Each node is written as a **triple**: $f = (v, g, h)$ where $g = f_v$ and $h = f_{v^-}$.

We read this triple as:

$$f = \text{if } v \text{ then } g \text{ else } h = \text{ite } (v, g, h) = vg + v' h$$



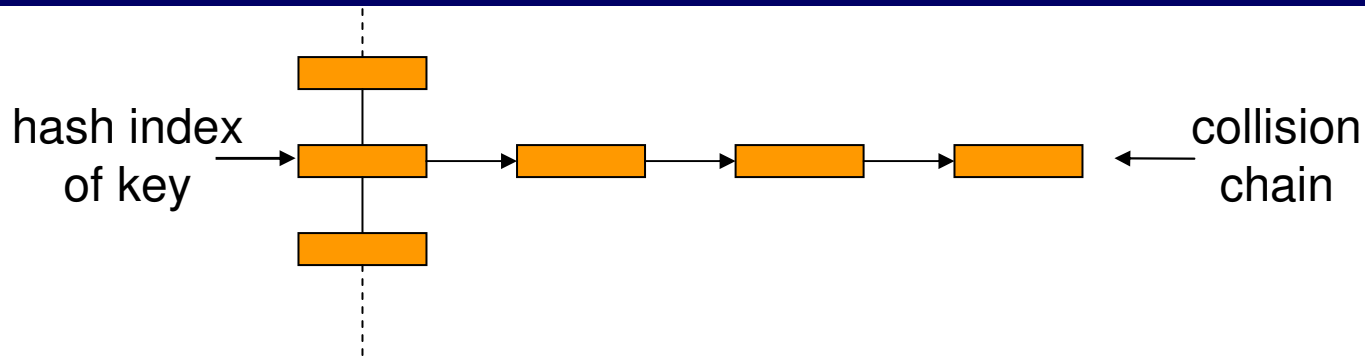
ITE Operator

$$\text{ite}(f, g, h) = fg + \bar{f}h$$

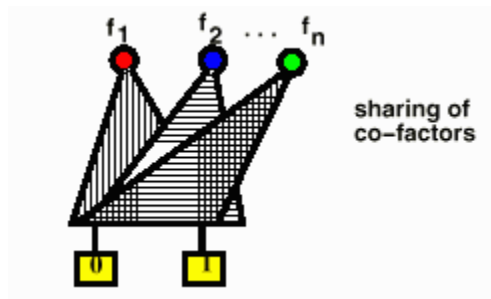
ITE operator can implement any two variable logic function. There are 16 such functions corresponding to all subsets of vertices of B^2 :

Table	Subset	Expression	Equivalent Form
0000	0	0	0
0001	AND(f, g)	fg	$\text{ite}(f, g, 0)$
0010	$f > g$	$f\bar{g}$	$\text{ite}(f, \bar{g}, 0)$
0011	f	f	f
0100	$f < g$	$\bar{f}g$	$\text{ite}(f, 0, g)$
0101	g	g	g
0110	XOR(f, g)	$f \oplus g$	$\text{ite}(f, \bar{g}, g)$
0111	OR(f, g)	$f + g$	$\text{ite}(f, 1, g)$
1000	NOR(f, g)	$\overline{f + g}$	$\text{ite}(f, 0, \bar{g})$
1001	XNOR(f, g)	$f \oplus \bar{g}$	$\text{ite}(f, g, \bar{g})$
1010	NOT(g)	\bar{g}	$\text{ite}(g, 0, 1)$
1011	$f \geq g$	$f + \bar{g}$	$\text{ite}(f, 1, \bar{g})$
1100	NOT(f)	\bar{f}	$\text{ite}(f, 0, 1)$
1101	$f \leq g$	$\bar{f} + g$	$\text{ite}(f, g, 1)$
1110	NAND(f, g)	\overline{fg}	$\text{ite}(f, \bar{g}, 1)$
1111	1	1	1

Unique Table - Hash Table



- Before a node (v, g, h) is added to BDD data base, it is looked up in the “unique-table”. If it is there, then existing pointer to node is used to represent the logic function. Otherwise, a new node is added to the unique-table and the new pointer returned.
- Thus a **strong canonical form** is maintained. The node for $f = (v, g, h)$ exists **iff** (v, g, h) is in the unique-table. There is only one pointer for (v, g, h) and that is the address to the unique-table entry.
- Unique-table allows single multi-rooted DAG to represent all users’ functions:



Recursive Formulation of ITE

v = top-most variable among the three BDDs f, g, h

$$\begin{aligned} \text{ite}(f, g, h) &= fg + \bar{f}h \\ &= v(fg + \bar{f}h)_v + \bar{v}(fg + \bar{f}h)_{\bar{v}} \\ &= v(f_v g_v + \bar{f}_v h_v) + \bar{v}(f_{\bar{v}} g_{\bar{v}} + \bar{f}_{\bar{v}} h_{\bar{v}}) \\ &= \text{ite}(v, \text{ite}(f_v, g_v, h_v), \text{ite}(f_{\bar{v}}, g_{\bar{v}}, h_{\bar{v}})) \\ &= (v, \text{ite}(f_v, g_v, h_v), \text{ite}(f_{\bar{v}}, g_{\bar{v}}, h_{\bar{v}})) \\ &= (v, A, B) \end{aligned}$$

Where A, B are pointers to results of $\text{ite}(f_v, g_v, h_v)$ and $\text{ite}(f_{\bar{v}}, g_{\bar{v}}, h_{\bar{v}})$
- merged if equal

Recursive Formulation of ITE

Algorithm **ITE**(f, g, h)

if(f == 1) **return** g

if(f == 0) **return** h

if(g == h) **return** g

if((p = **HASH_LOOKUP_COMPUTED_TABLE**(f,g,h)) **return** p

v = **TOP_VARIABLE**(f, g, h) //top variable from f,g,h

fn = **ITE**(f_v, g_v, h_v) //recursive calls

gn = **ITE**(f_v, g_v, h_v) - -

if(fn == gn) **return** gn //reduction

if(!(p = **HASH_LOOKUP_UNIQUE_TABLE**(v,fn,gn)) {

 p = **CREATE_NODE**(v,fn,gn) // and insert into UNIQUE_TABLE

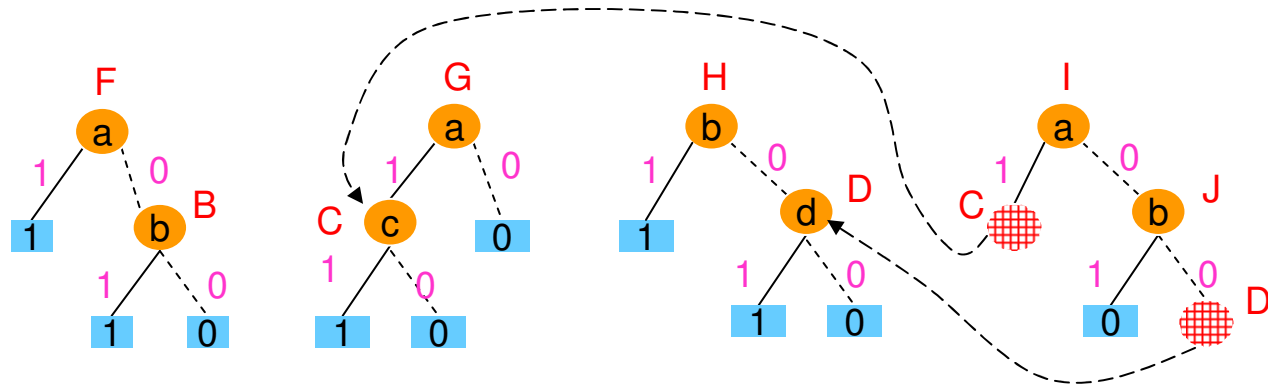
}

INSERT_COMPUTED_TABLE(p, **HASH_KEY**{f,g,h})

return p

}

Example



$$\begin{aligned}
 I &= \text{ite}(F, G, H) \\
 &= (a, \text{ite}(F_a, G_a, H_a), \text{ite}(F_{\bar{a}}, G_{\bar{a}}, H_{\bar{a}})) \\
 &= (a, \text{ite}(1, C, H), \text{ite}(B, 0, H)) \\
 &= (a, C, (b, \text{ite}(B_b, 0_b, H_b), \text{ite}(B_{\bar{b}}, 0_{\bar{b}}, H_{\bar{b}}))) \\
 &= (a, C, (b, \text{ite}(1, 0, 1), \text{ite}(0, 0, D))) \\
 &= (a, C, (b, 0, D)) \\
 &= (a, C, J)
 \end{aligned}$$

F,G,H,I,J,B,C,D
are pointers

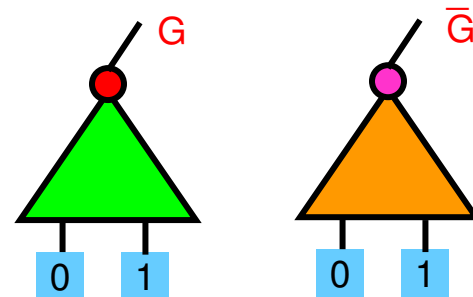
Check: $F = a + b$, $G = ac$, $H = b + d$

$$\text{ite}(F, G, H) = (a + b)(ac) + ab(b + d) = ac + abd$$

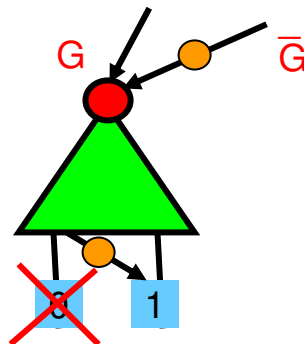
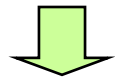
Extension - Complement Edges

Combine inverted functions by using complemented edge

- similar to circuit case
- reduces memory requirements
- BUT MORE IMPORTANT:
 - makes some operations more efficient (NOT, ITE)



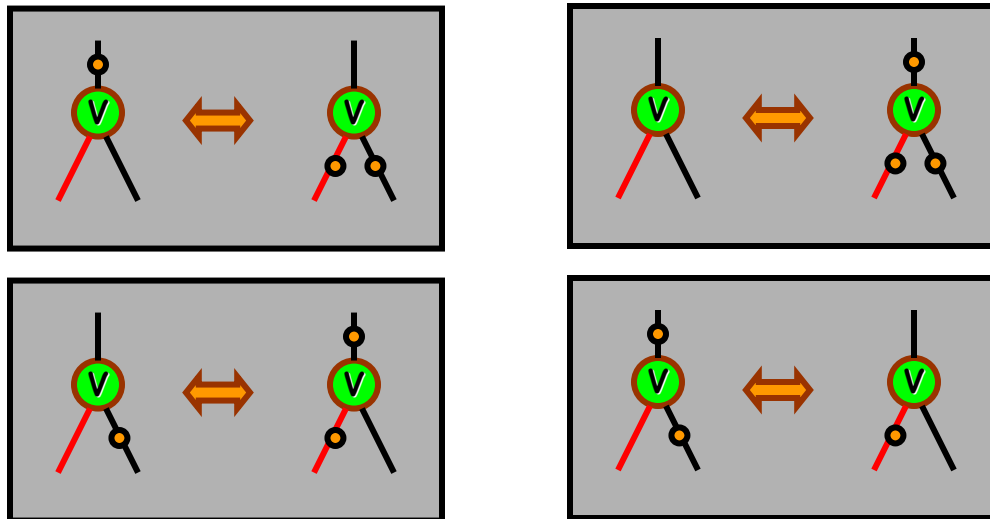
two different
DAGs



only one DAG
using complement
pointer

Extension - Complement Edges

To maintain strong canonical form, need to resolve 4 equivalences:



Solution: Always choose one on **left**, i.e. the “then” leg must have **no** complement edge.

Ambiguities in Computed Table

Standard Triples:

$$ite(F, F, G) \Rightarrow ite(F, 1, G)$$

$$ite(F, G, F) \Rightarrow ite(F, G, 0)$$

$$ite(F, G, \bar{F}) \Rightarrow ite(F, G, 1)$$

$$ite(F, \bar{F}, G) \Rightarrow ite(F, 0, G)$$

To resolve equivalences:

$$ite(F, 1, G) \equiv ite(G, 1, F)$$

$$ite(F, 0, G) \equiv ite(\bar{G}, 1, \bar{F})$$

$$ite(F, G, 0) \equiv ite(G, F, 0)$$

$$ite(F, G, 1) \equiv ite(\bar{G}, \bar{F}, 1)$$

$$ite(F, G, \bar{G}) \equiv ite(G, F, \bar{F})$$

To maximize matches on computed table:

1. First argument is chosen with smallest top variable.
2. Break ties with smallest address pointer. (breaks PORTABILITY!!!!!!!!!!!!)

Triples:

$$ite(F, G, H) \equiv ite(\bar{F}, H, G) \equiv \overline{ite(F, \bar{G}, \bar{H})} \equiv \overline{ite(\bar{F}, \bar{H}, G)}$$

Choose the one such that the first and second argument of *ite* should not be complement edges (i.e. the first one above).

Use of Computed Table

- Often BDD packaged use optimized implementations for special operations
 - e.g. ITE_Constant (check whether the result would be a constant)
 - AND_Exist (AND operation with existential quantification)
- All operations need a cache for decent performance
 - local cache
 - for one operation only - cache will be thrown away after operation is finished (e.g. AND_Exist)
 - keep inter-operational (ITE, ...)
 - special cache for each operation
 - does not need to store operation type
 - shared cache for all operations
 - better memory handling
 - needs to store operation type

Example: Tautology Checking

```
Algorithm ITE_CONSTANT(f,g,h) { //returns 0,1, or NC
if(TRIVIAL_CASE(f,g,h) return result (0,1, or NC)
if((res = HASH_LOOKUP_COMPUTED_TABLE(f,g,h))) return res
v = TOP_VARIABLE(f,g,h)
i = ITE_CONSTANT(fv,gv,hv)
if(i == NC) {
  INSERT_COMPUTED_TABLE(NC, HASH_KEY{f,g,h}) // special table!!
  return NC
}
e = ITE_CONSTANT(fv,gv,hv)
if(e == NC) {
  INSERT_COMPUTED_TABLE(NC, HASH_KEY{f,g,h})
  return NC
}
if(e != i) {
  INSERT_COMPUTED_TABLE(NC, HASH_KEY{f,g,h})
  return NC
}
INSERT_COMPUTED_TABLE(e, HASH_KEY{f,g,h})
return i;
}
```

Variable Ordering

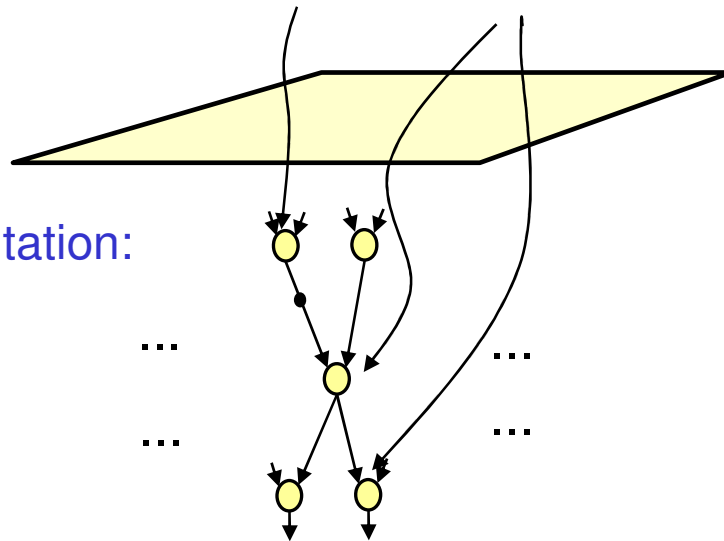
- Static variable ordering
 - variable ordering is computed up-front based on the problem structure
 - works very well for many combinational functions that come from circuits we actually build
 - general scheme: control variables first
 - DFS order is pretty good for most cases
 - work bad for unstructured problems
 - e.g., using BDDs to represent arbitrary sets
 - lots of research in ordering algorithms
 - simulated annealing, genetic algorithms
 - give better results but extremely costly

Dynamic Variable Ordering

- Changes the order in the middle of BDD applications
 - must keep same global order
- Problem: External pointers reference internal nodes!!!

External reference pointers attached to application data structures

BDD Implementation:



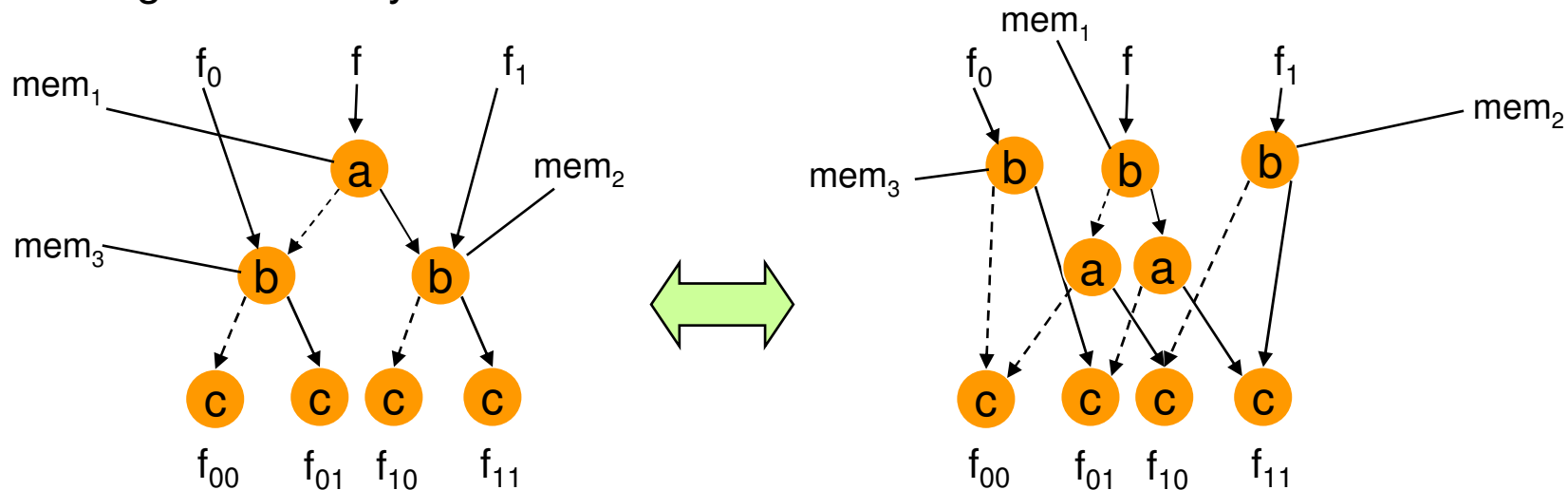
Dynamic Variable Ordering

Theorem (Friedman):

Permuting any top part of the variable order has no effect on the nodes labeled by variables in the bottom part.

Permuting any bottom part of the variable order has no effect on the nodes labeled by variables in the top part.

- Trick: Two adjacent variable layers can be exchanged by keeping the original memory locations for the nodes



Dynamic Variable Ordering

- BDD sifting:
 - shift each BDD variable to the top and then to the bottom and see which position had minimal number of BDD nodes
 - efficient if separate hash-table for each variable
 - can stop if lower bound on size is worse than the best found so far
 - shortcut:
 - two layers can be swapped very cheaply if there is no interaction between them
 - expensive operation, sophisticated trigger condition to invoke it
- grouping of BDD variables:
 - for many applications, pairing or grouping variables gives better ordering
 - e.g. current state and next state variables in state traversal
 - grouping them for sifting explores ordering that are otherwise skipped

Garbage Collection

- Very important to free and reuse memory of unused BDD nodes
 - explicitly freed by an external `bdd_free` operation
 - BDD nodes that were temporarily created during BDD operations
- Two mechanisms to check whether a BDD is not referenced:
 - **Reference counter** at each node
 - increment whenever node gets one more reference (incl. External)
 - decrement when node gets de-references (`bdd_free` from external, de-reference from internal)
 - counter-overflow -> freeze node
 - **Mark and Sweep** algorithm
 - does not need counter
 - first pass, mark all BDDs that are referenced
 - second pass, free the BDDs that are not marked
 - need additional handle layer for external references

Garbage Collection

- Timing is very crucial because garbage collection is expensive
 - immediately when node gets freed
 - bad because dead nodes get often reincarnated in next operation
 - regular garbage collections based on statistics collected during BDD operations
 - “death row” for nodes to keep them around for a bit longer
- Computed table must be cleared since not used in reference mechanism
- Improving memory locality and therefore cache behavior:
 - sort freed BDD nodes

BDD Derivatives

- MDD: Multi-valued BDDs
 - natural extension, have more than two branches
 - can be implemented using a regular BDD package with binary encoding
 - advantage that binary BDD variables for one MV variable do not have to stay together -> potentially better ordering
- ADDs: (Analog BDDs) MTBDDs
 - multi-terminal BDDs
 - decision tree is binary
 - multiple leafs, including real numbers, sets or arbitrary objects
 - efficient for matrix computations and other non-integer applications
- FDDs: Free BDDs
 - variable ordering differs
 - not canonical anymore
- and many more