095946- ADVANCED ALGORITHMS AND PARALLEL PROGRAMMING

Fabrizio Ferrandi

a.a. 2021-2022

□ **Dynamic Programming**
 • BDDs

ROBDD's (The Canonical Side)

- General idea:
  - instead of: exploring sub-cases by enumerating them in time
  - store sub-cases in memory
  - KEY: two hashing mechanisms:
    - unique table: find identical sub-cases and avoid replication
    - computed table: reduce redundant computation of sub-cases

- Representation of a logic function as graph (DAG):
  - many logic functions can be represented compactly - usually better than SOP's

- Can be made canonical !!
  - Shift the effort in a Boolean reasoning engine from SAT algorithm to representation

- Many logic operations can be performed efficiently on BDD's:
  - usually linear in size of result - tautology and complement are constant time

- Size of BDD critically dependent on variable ordering

ROBDD's

- 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)

$$f = vf_v + v'f_{v'}$$

  - Reduced:
    - any node with two identical children is removed
    - two nodes with isomorphic BDD's are merged
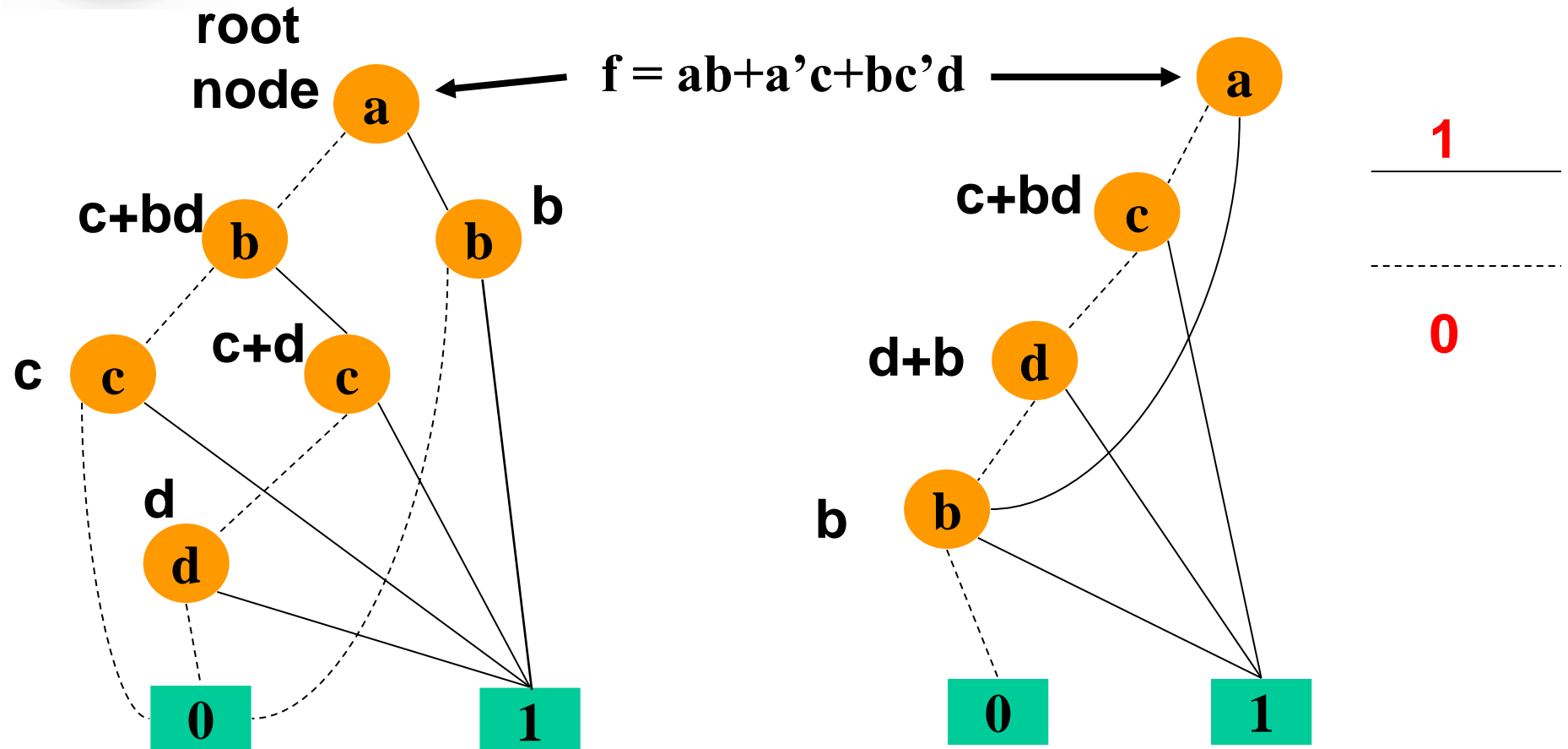 - Ordered:
    - Co-factoring variables (splitting variables) always follow the same order along all paths

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

root node

a

$$f = ab+a'c+bc'd$$

a

1

c+bd **b**

b **b**

c+bd **c**

0

c **c**

c+d **c**

d+b **d**

d **d**
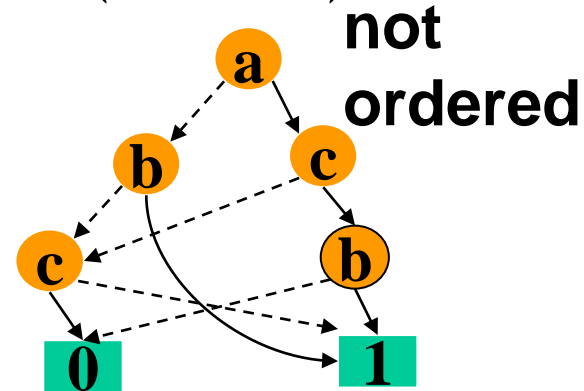
b **b**
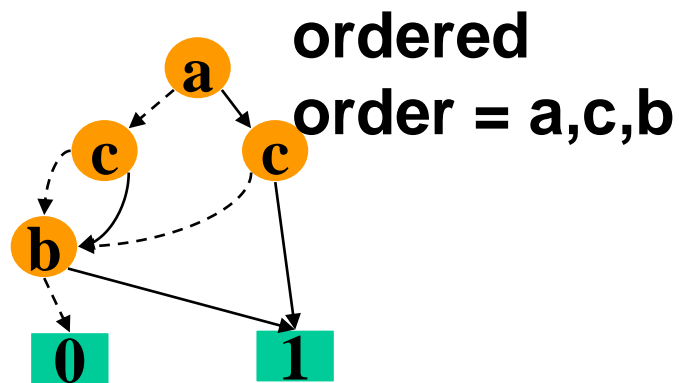
0

1

0

1

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.
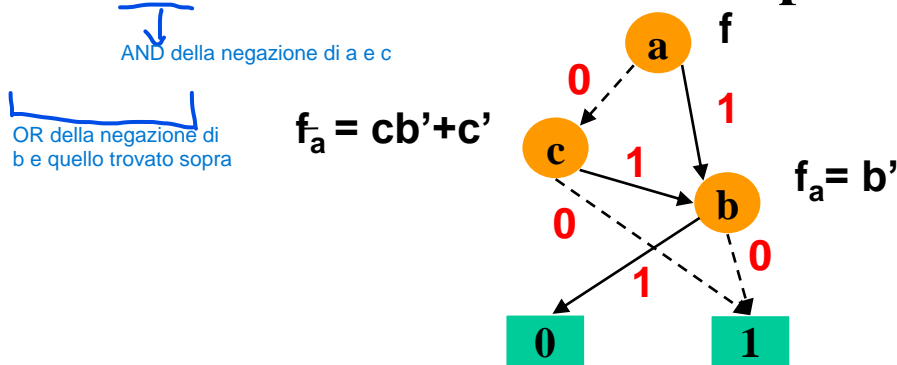
ordered

order = a,c,b

not

ordered

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. **if two nodes have isomorphic graphs, they are replaced by one of them**
These two rules make it so that each node represents a distinct logic function.

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

AND della negazione di a e c

OR della negazione di b e quello trovato sopra

$f_a = cb'+c'$

$f_a = b'$

a f

0

1

c

1

0

b

0

1

0   1

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 measures 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
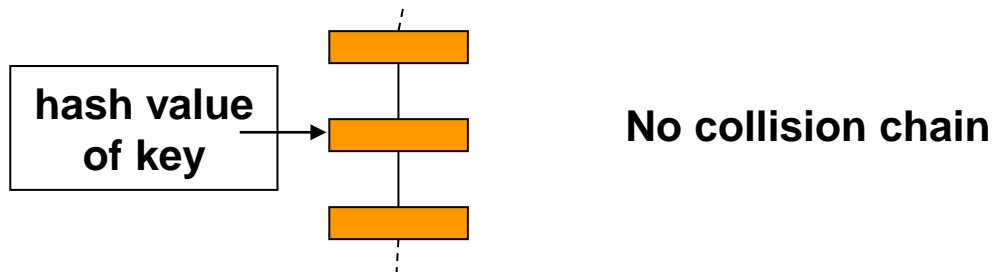
Efficient Implementation of BDD's

Unique Table:

- ❏ avoids duplication of existing nodes
   - – Hash-Table: hash-function(key) = value
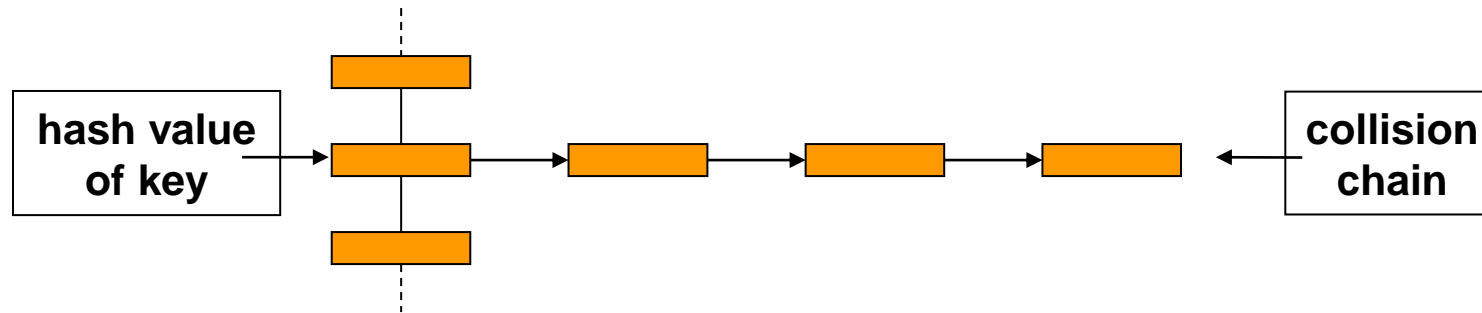
Computed Table:

- ❏ 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:
    - $(node(f_v) = node(g_v)) \wedge (node(f_{\bar{v}}) = node(g_{\bar{v}})) \Rightarrow node(f) = node(g)$
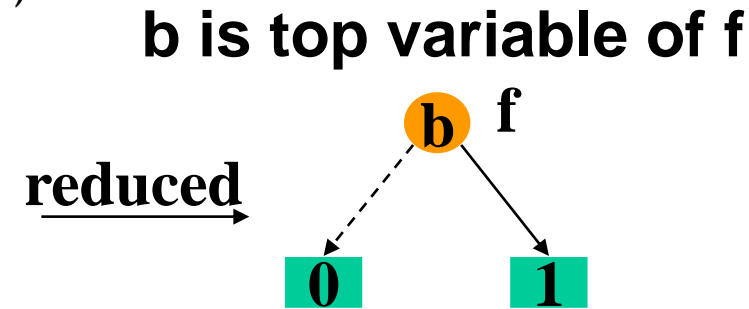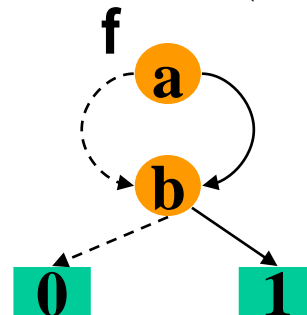 - provides recursive argument that node(f) is unique when using the unique hash-table
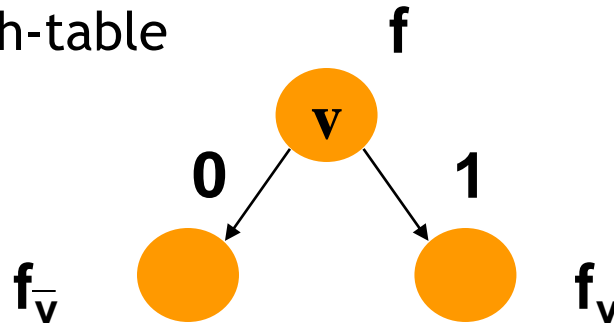
Implementation

- 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.
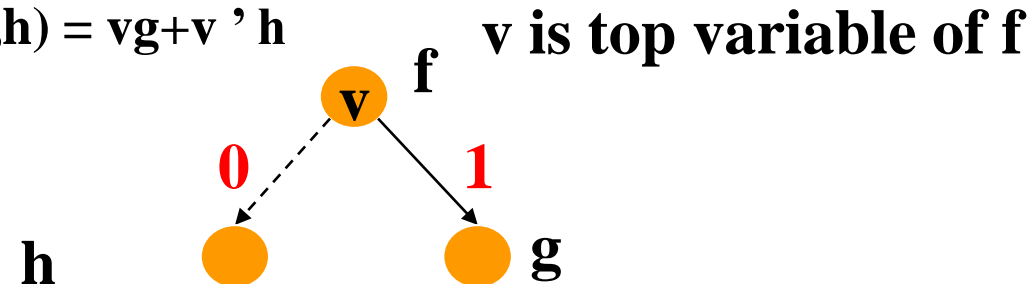- Example: f = ab + a'bc + a'bc'. Order is (a < b < c).
  - fa = b, f $\bar{a}$ = b
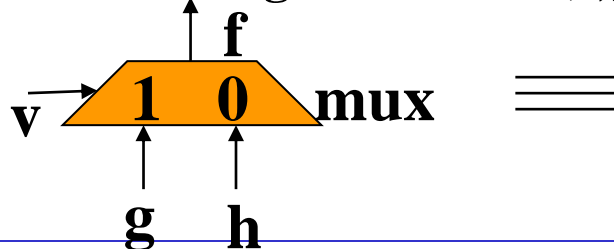
b is top variable of f

f does not depend on a,
since $f_a = f_{\bar{a}}$.

$$\text{reduced}$$

Each node is written as a **triple:** $f = (v,g,h)$ where $g = f_v$ and $h = f_{\bar{v}}$.
  We read this triple as:
    f = if v then g else h = **ite** (v,g,h) = vg+v'h

v is top variable of f

ITE Operator

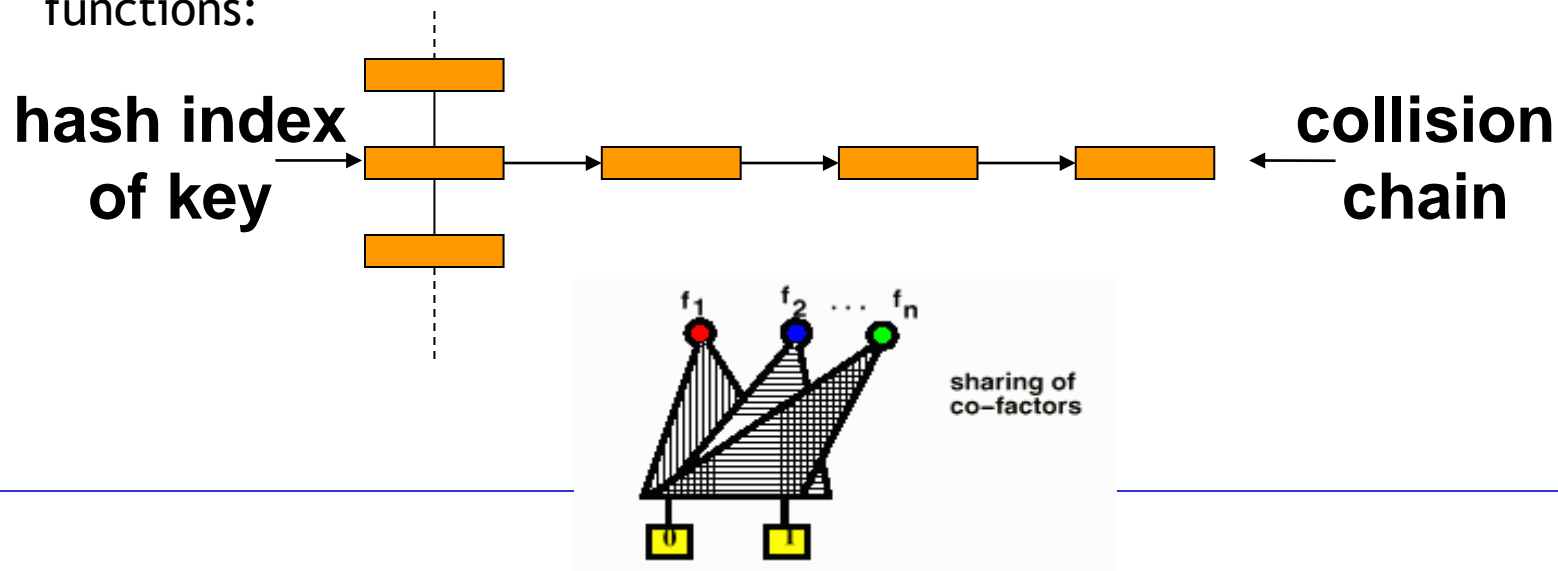$$\mathrm{ite}(f, g, h) = fg + \bar{f}h$$

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	ite(f, g, 0)
0010	f > g	f $\bar{g}$	ite(f, $\bar{g}$, 0)
0011	f	f	f
0100	f < g	$\bar{f}$g	ite(f, 0, g)
0101	g	g	g
0110	XOR(f, g)	f ⊕ g	ite(f, $\bar{g}$, g)
0111	OR(f, g)	f + g	ite(f, 1, g)
1000	NOR(f, g)	$\overline{f + g}$	ite(f, 0, $\bar{g}$)
1001	XNOR(f, g)	$\overline{f \oplus g}$	ite(f, g, $\bar{g}$)
1010	NOT(g)	$\bar{g}$	ite(g, 0, 1)
1011	f ≥ g	f + $\bar{g}$	ite(f, 1, $\bar{g}$)
1100	NOT(f)	$\bar{f}$	ite(f, 0, 1)
1101	f ≤ g	$\bar{f}$ + g	ite(f, g, 1)
1110	NAND(f, g)	$\overline{fg}$	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:

hash index of key → → **collision chain** ←

$f_1 \quad f_2 \quad \cdots \quad f_n$

sharing of co-factors

Recursive Formulation of ITE

v = top-most variable among the three BDD's *f, g, h*

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

$$= v(fg + \overline{f}h)_v + \overline{v}(fg + \overline{f}h)_{\overline{v}}$$

$$= v(f_v g_v + \overline{f_v}h_v) + \overline{v}(f_{\overline{v}} g_{\overline{v}} + \overline{f_{\overline{v}}}h_{\overline{v}})$$

$$= ite(v, ite(f_v, g_v, h_v), ite(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}}))$$

$$= (v, ite(f_v, g_v, h_v), ite(f_{\overline{v}}, g_{\overline{v}}, h_{\overline{v}}))$$

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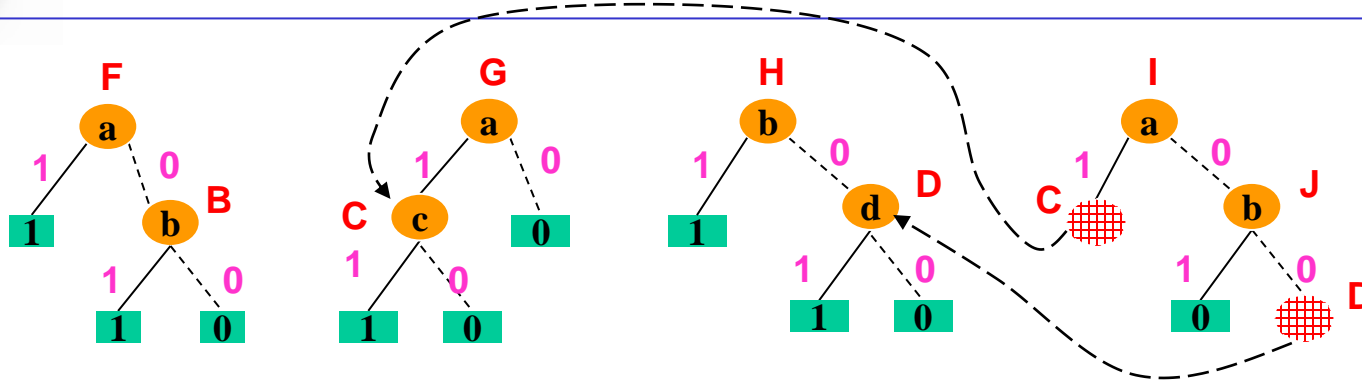
F,G,H,I,J,B,C,D
are pointers

I $= ite\,(F, G, H)$

$= (a,\ ite\,(F_a,\ G_a,\ H_a),\ ite\,(F_{\bar{a}},\ G_{\bar{a}},\ H_{\bar{a}}))$

$= (a,\ ite\,(1,\ C,\ H),\quad ite(B, 0, H))$

$= (a,\ C,\qquad\qquad (b,\ ite\,(B_b,\ 0_b,\ H_b),\ ite\,(B_{\bar{b}},\ 0_{\bar{b}},\ H_{\bar{b}}))$

$= (a,\ C,\qquad\qquad (b,\ ite\,(1,\ 0,\ 1),\ ite\,(0,\ 0,\ D)))$

$= (a,\ C,\qquad\qquad (b,\ 0,\ D))$

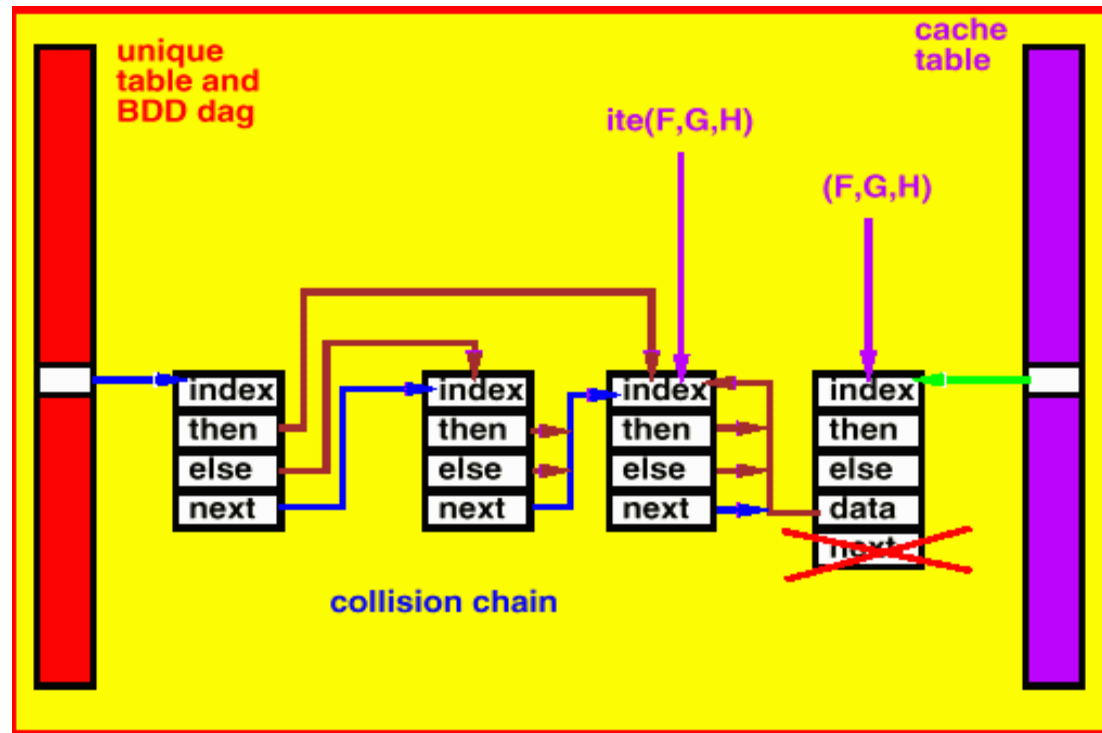$= (a,\ C,\qquad\qquad J)$

Check: $F = a + b$

$G = ac$

$H = b + d$

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

Computed Table

Keep a record of (*F, G, H*) triplets already computed by the ITE operator
- software cache (*"cache" table*)
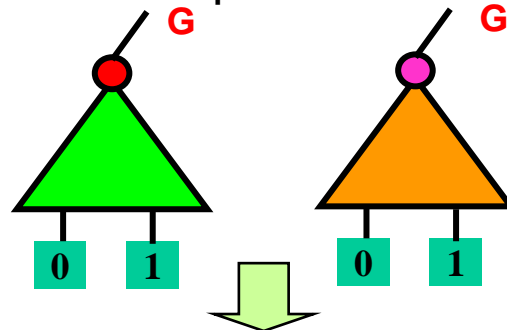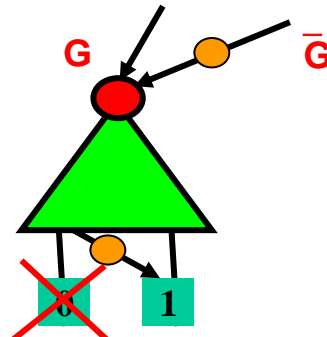- simply hash-table without collision chain (lossy cache)

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
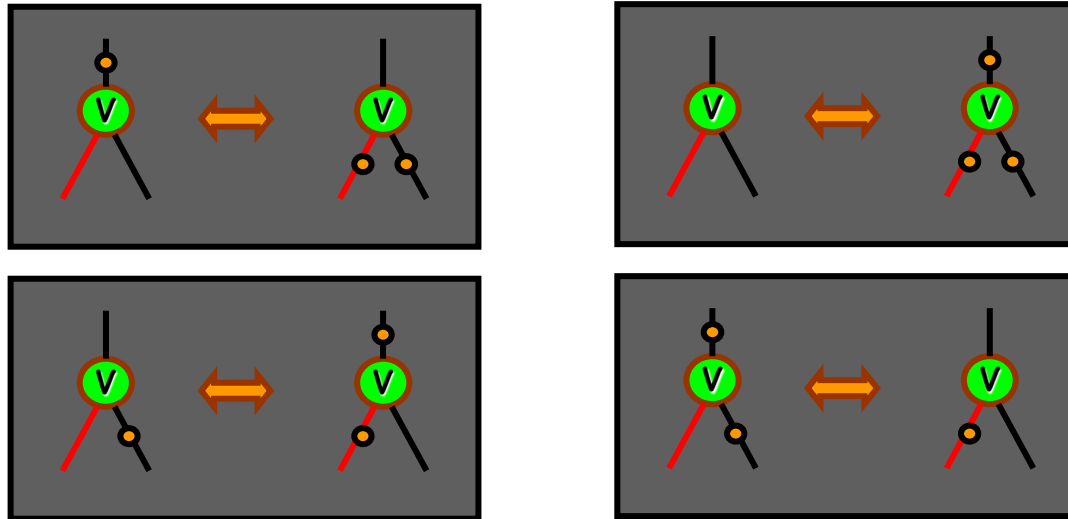
only one DAG using complement pointer

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:

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

To resolve equivalences:

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

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:

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

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;
}
```

Compose

Compose is an important operation, e.g. for building the BDD of a circuit backwards

Compose(*F, v, G*) : $F(v, x) \rightarrow F(G(x), x)$, means substitute $v = G(x)$

```
Algorithm COMPOSE(F,v,G) {
  if(TOP_VARIABLE(F) > v)  return F  // F does not depend on v
  if(TOP_VARIABLE(F) == v) return ITE(G,F₁,F₀)
  i = COMPOSE(F₁,v,G)
  e = COMPOSE(F₀,v,G)
  return ITE(TOP_VARIABLE(F),i,e)   // Why not CREATE_NODE...
}
```

Notes:
1. F_1 is the 1-child of *F*, F_0 the 0-child.
2. *G* , *i*, *e* are not functions of *v* .
3. If TOP_VARIABLE of *F* is *v*, then *ite* (*G* , *i*, *e*) does the replacement of *v* by *G*.

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
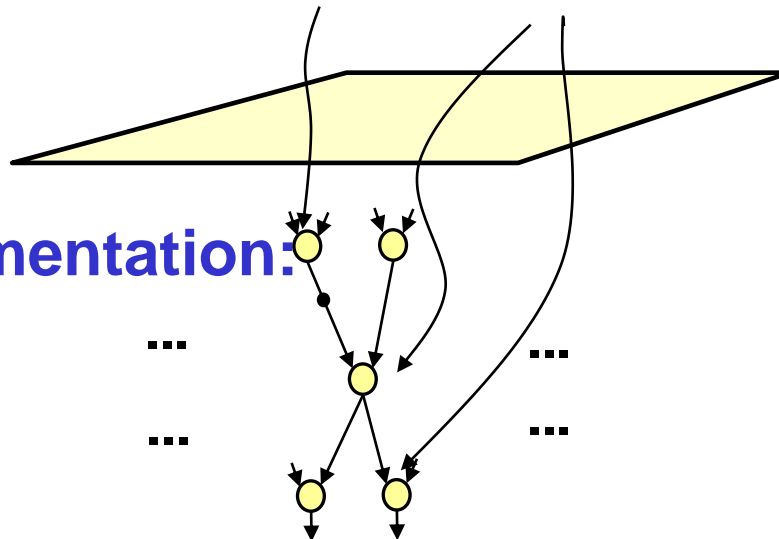   - • give better results but extremely costly

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
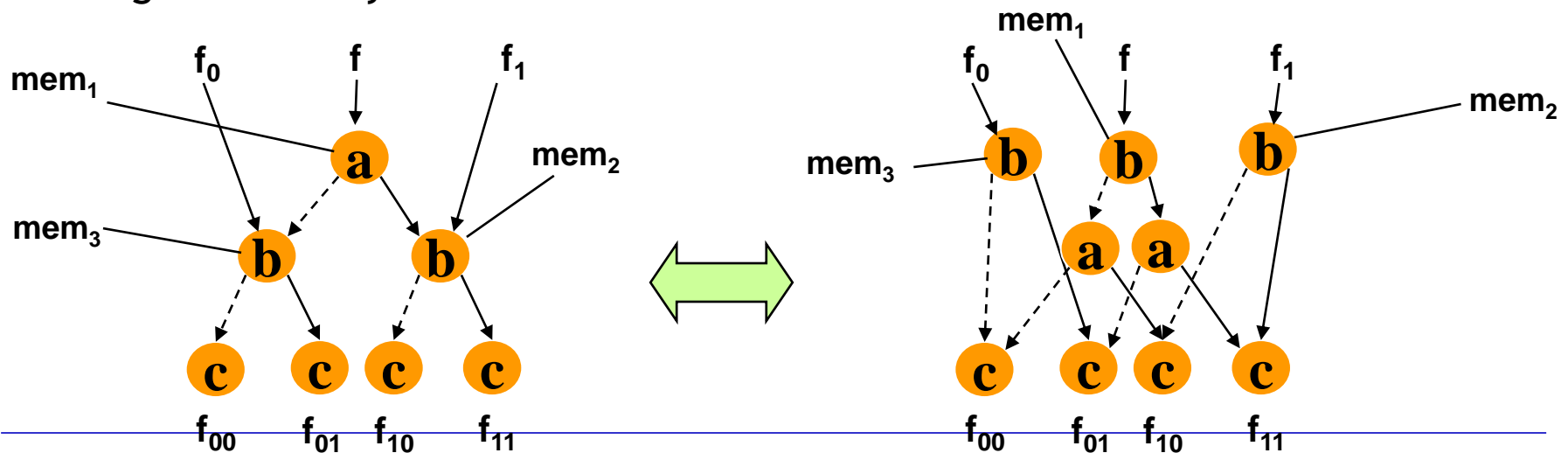
Dynamic Variable Ordering

- 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 then 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 ruse memory of unused BDD nodes
- explicitly free'd by an external `bdd_free` operation
- BDD nodes that were temporary created during BDD operations

□ Two mechanism 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 free'ed
        - 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 free'ed BDD nodes to

BDD Derivatives

- MDD: Multi-valued BDDs
    - natural extension, have more then 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 …..

Zero Suppressed BDD's - ZBDD's

ZBDD's were invented by Minato to efficiently represent sparse sets. They have turned out to be useful in implicit methods for representing primes (which usually are a sparse subset of all cubes).

Different reduction rules:

- BDD: eliminate all nodes where then edge and else edge point to the same node.
- ZBDD: eliminate all nodes where the then node points to 0. Connect incoming edges to else node.
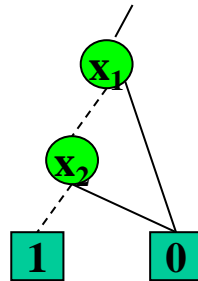- For both: share equivalent nodes.

BDD:
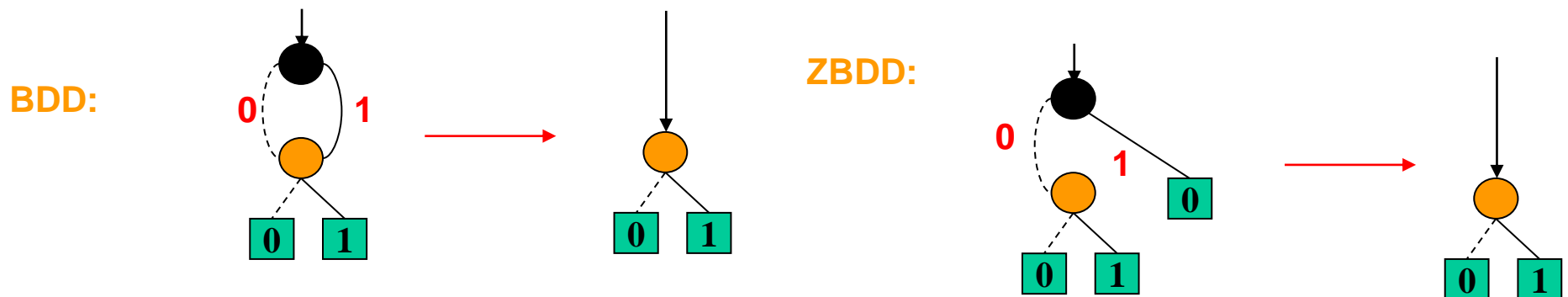
ZBDD:

Canonicity

Theorem: (Minato) ZBDD's are canonical given a variable ordering and the support set.
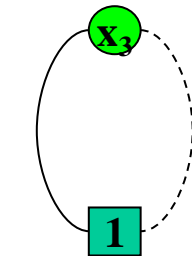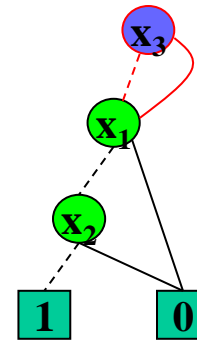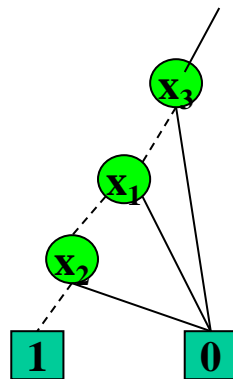
Example:

BDD

ZBDD if support is x_1, x_2

ZBDD if support is x_1, x_2, x_3

BDD

ZBDD if support is x_1, x_2, x_3
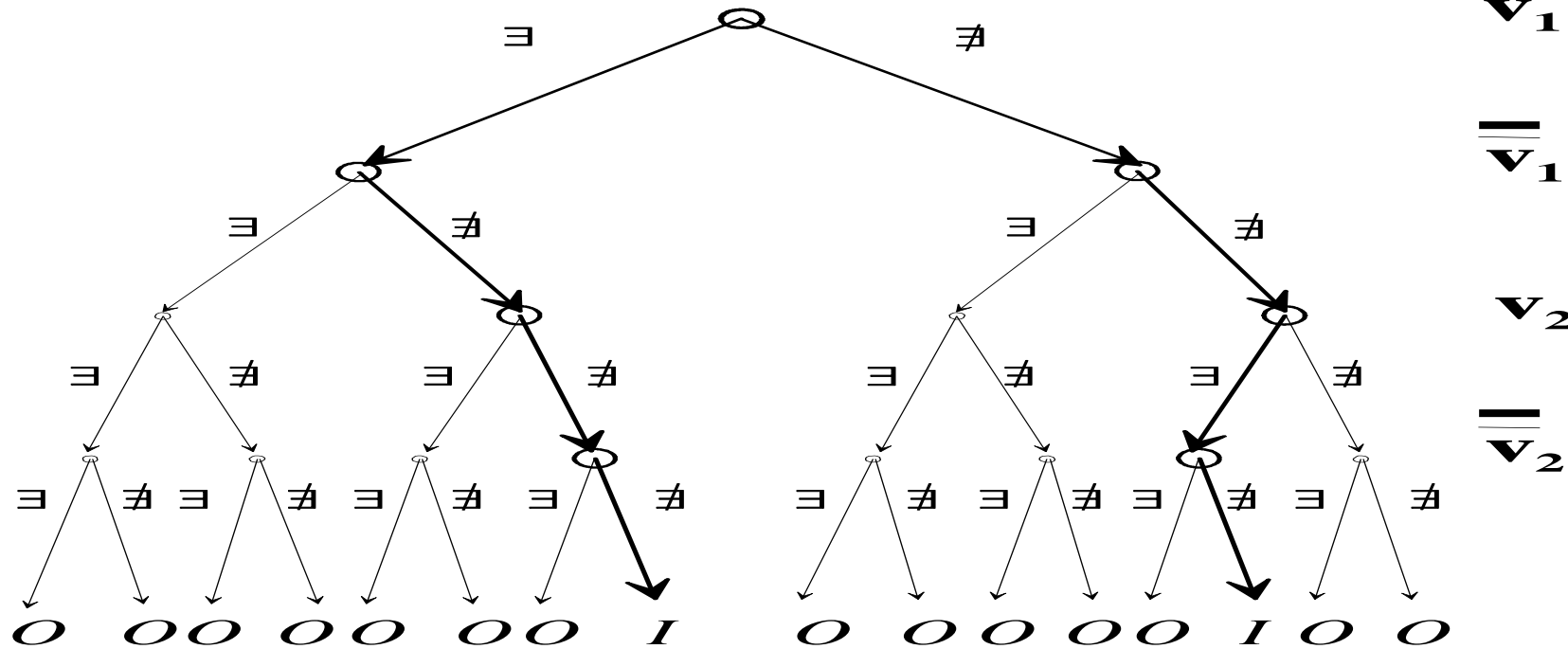
An algebraic approach

ZDDs

Combinational sets

$$P = \{v_1, v_2\}$$

Symbols

$I = inside$

$O = outside$

v_1

$\overline{v_1}$

v_2

$\overline{v_2}$

I

O