Logic Synthesis and Verification

Jie-Hong Roland Jiang
江介宏

Department of Electrical Engineering
National Taiwan University

Fall 2012

Boolean Function Representation

Reading:
Logic Synthesis in a Nutshell
Section 2

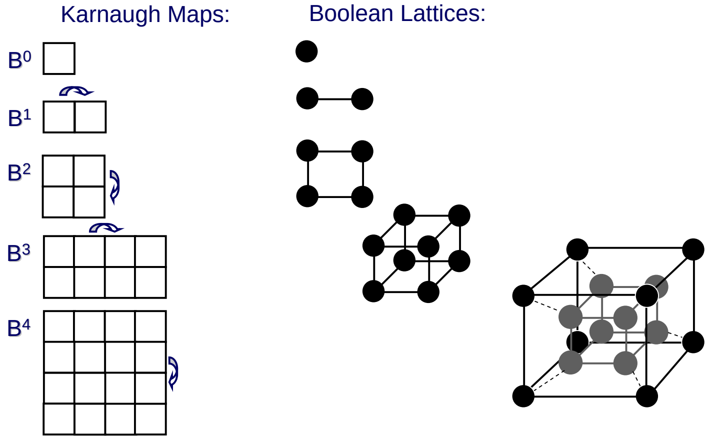
most of the following slides are by
courtesy of Andreas Kuehlmann

Assumption

- Unless otherwise said, from now on we are concerned with two-element Boolean algebra (i.e. **B** = {0,1})

Boolean Space

- B = {0,1}
- B^2 = {0,1}×{0,1} = {00, 01, 10, 11}

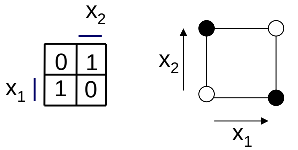


Boolean Function

- For **B** = {0,1}, a Boolean function f: $\mathbf{B}^n \to \mathbf{B}$ over variables $x_1,\ldots,x_n$ maps each Boolean valuation (truth assignment) in $\mathbf{B}^n$ to 0 or 1

Example

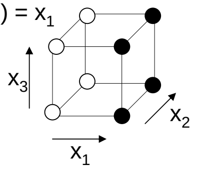
$f(x_1,x_2)$ with f(0,0) = 0, f(0,1) = 1, f(1,0) = 1, f(1,1) = 0



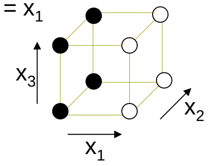
Boolean Function

- Onset of f, denoted as f^1, is $f^1 = \{v \in \mathbf{B}^n \mid f(v)=1\}$
  - If $f^1 = \mathbf{B}^n$, f is a tautology
- Offset of f, denoted as f^0, is $f^0 = \{v \in \mathbf{B}^n \mid f(v)=0\}$
  - If $f^0 = \mathbf{B}^n$, f is unsatisfiable. Otherwise, f is satisfiable.
- f^1 and f^0 are sets, not functions!
- Boolean functions f and g are equivalent if $\forall v \in \mathbf{B}^n.\ f(v) = g(v)$ where v is a truth assignment or Boolean valuation
- A literal is a Boolean variable *x* or its negation x′ (or x, ¬x) in a Boolean formula

$f(x_1, x_2, x_3) = x_1$

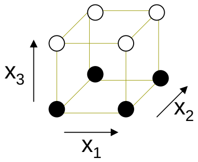


$f(x_1, x_2, x_3) = \overline{x}_1$



Boolean Function

- There are 2^n vertices in $\mathbf{B}^n$
- There are 2^{2^n} distinct Boolean functions
  - Each subset $f^1 \subseteq \mathbf{B}^n$ of vertices in $\mathbf{B}^n$ forms a distinct Boolean function f with onset f^1



$x_1x_2x_3$	f
0 0 0	1
0 0 1	0
0 1 0	1
0 1 1	0
1 0 0	⇒ 1
1 0 1	0
1 1 0	1
1 1 1	0

Boolean Operations

Given two Boolean functions:

$f : \mathbf{B}^n \to \mathbf{B}$
$g : \mathbf{B}^n \to \mathbf{B}$

- h = f ∧ g from AND operation is defined as
  $h^1 = f^1 \cap g^1$; $h^0 = \mathbf{B}^n \setminus h^1$
- h = f ∨ g from OR operation is defined as
  $h^1 = f^1 \cup g^1$; $h^0 = \mathbf{B}^n \setminus h^1$
- h = ¬f from COMPLEMENT operation is defined as
  $h^1 = f^0$; $h^0 = f^1$

Cofactor and Quantification

Given a Boolean function:
f : $\mathbf{B}^n \rightarrow \mathbf{B}$, with the input variable $(x_1,x_2,\ldots,x_i,\ldots,x_n)$

- Positive cofactor on variable x_i
  $h = f_{x_i}$ is defined as $h = f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Negative cofactor on variable x_i
  $h = f_{\neg x_i}$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n)$
- Existential quantification over variable x_i
  $h = \exists x_i.\ f$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \vee f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Universal quantification over variable x_i
  $h = \forall x_i.\ f$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \wedge f(x_1,x_2,\ldots,1,\ldots,x_n)$
- Boolean difference over variable x_i
  $h = \partial f/\partial x_i$ is defined as $h = f(x_1,x_2,\ldots,0,\ldots,x_n) \oplus f(x_1,x_2,\ldots,1,\ldots,x_n)$

Representation of Boolean Function

- Represent Boolean functions for two reasons
  - to represent and manipulate the actual circuit we are implementing
  - to facilitate Boolean reasoning
- Data structures for representation
  - Truth table
  - Boolean formula
    - Sum of products (Disjunctive “normal” form, DNF)
    - Product of sums (Conjunctive “normal” form, CNF)
  - Boolean network
    - Circuit (network of Boolean primitives)
    - And-inverter graph (AIG)
  - Binary Decision Diagram (BDD)

Boolean Function Representation
Truth Table

- Truth table (function table for multi-valued functions):
  The truth table of a function f : $\mathbf{B}^n \rightarrow \mathbf{B}$ is a tabulation of its value at each of the 2^n vertices of $\mathbf{B}^n$.

  In other words the truth table lists all mintems
  Example: f = a′b′c′d + a′b′cd + a′bc′d +
  ab′c′d + ab′cd + abc′d +
  abcd′ + abcd

  The truth table representation is
  - impractical for large *n*
  - canonical
  If two functions are the same, then their canonical representations are isomorphic.

	abcd	f		abcd	f
0	0000	0	8	1000	0
1	0001	1	9	1001	1
2	0010	0	10	1010	0
3	0011	1	11	1011	1
4	0100	0	12	1100	0
5	0101	1	13	1101	1
6	0110	0	14	1110	1
7	0111	0	15	1111	1

Boolean Function Representation
Boolean Formula

- A Boolean formula is defined inductively as an expression with the following formation rules (syntax):

formula ::= ‘(‘ formula ‘)’
| Boolean constant (**true** or **false**)
| <Boolean variable>
| formula “+” formula (OR operator)
| formula “·” formula (AND operator)
| ¬ formula (complement)

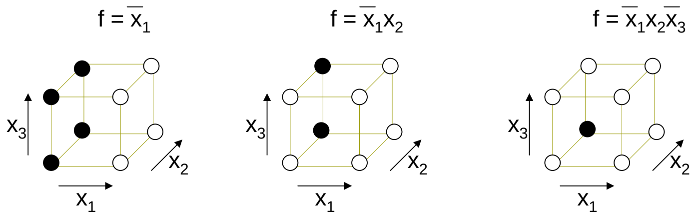
Example
$f = (x_1 \cdot x_2) + (x_3) + \neg(\neg(x_4 \cdot (\neg x_1)))$
typically “·” is omitted and ‘(’, ‘)’ and ‘¬’ are simply reduced by priority,
e.g. $f = x_1\, x_2 + x_3 + x_4 \neg x_1$

Boolean Function Representation
Boolean Formula in SOP

- A cube is defined as a conjunction of literals, i.e. a product term.

Example

$C = x_1x_2'x_3$ represents the function with onset: $f^1 = \{(x_1=1,x_2=0,x_3=1)\}$ in the Boolean space spanned by x_1,x_2,x_3, or $f^1 = \{(x_1=1,x_2=0,x_3=1, x_4=0), (x_1=1,x_2=0,x_3=1,x_4=1)\}$ in the Boolean space spanned by x_1,x_2,x_3,x_4, or …



Boolean Function Representation
Boolean Formula in SOP

- If $C \subseteq f^1$, C the onset of a cube c, then c is an implicant of f
- If $C \subseteq \mathbf{B}^n$, and c has k literals, then $|C| = 2^{n-k}$, i.e., C has 2^{n-k} elements

Example

$c = xy'$ $(c:\mathbf{B}^3 \rightarrow \mathbf{B})$, $C = \{100, 101\} \subseteq \mathbf{B}^3$

$k = 2$, $n = 3$ $|C| = 2 = 2^{3-2}$

- An implicant with n literals is a minterm

Boolean Function Representation
Boolean Formula in SOP

- A function can be represented by a sum-of-cubes (products):

  $f = ab + ac + bc$

  Since each cube is a product of literals, this is a sum-of-products (SOP) representation or disjunctive normal form (DNF)
- An SOP can be thought of as a set of cubes F

  $F = \{ab, ac, bc\}$
- A set of cubes that represents f is called a cover of f.

  $F_1=\{ab, ac, bc\}$ and $F_2=\{abc, abc', ab'c, a'bc\}$

  are covers of

  $f = ab + ac + bc$.
- Mainly used in circuit synthesis; seldom used in Boolean reasoning

---

Boolean Function Representation
Boolean Formula in POS

- Product-of-sums (POS), or conjunctive normal form (CNF), representation of Boolean functions
  - Dual of the SOP representation

Example

$\varphi = (a+b'+c)\ (a'+b+c)\ (a+b'+c')\ (a+b+c)$

- A Boolean function in a POS representation can be derived from an SOP representation with De Morgan's law and the distributive law
- Mainly used in Boolean reasoning; rarely used in circuit synthesis (due to the asymmetric characteristics of NMOS and PMOS)

Boolean Function Representation
Boolean Network

- Used for two main purposes
  - as target structure for logic implementation which gets restructured in a series of logic synthesis steps until result is acceptable
  - as representation for Boolean reasoning engine
- Efficient representation for most Boolean problems
  - memory complexity is similar to the size of circuits that we are actually building
- Close to the input and output representations of logic synthesis

---

Boolean Function Representation
Boolean Network

A Boolean network is a directed graph C(G,N) where G are the gates and N $\subseteq$ (G$\times$G) are the directed edges (nets) connecting the gates.

Some of the vertices are designated:
Inputs: I $\subseteq$ G
Outputs: O $\subseteq$ G
I $\cap$ O = $\emptyset$

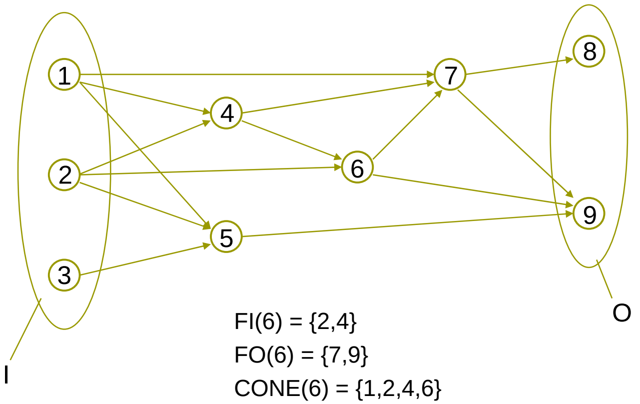
Each gate g is assigned a Boolean function f_g which computes the output of the gate in terms of its inputs.

Boolean Function Representation
Boolean Network

- The fanin FI(g) of a gate g are the predecessor gates of g:
  FI(g) = {g' | (g',g) $\in$ N} (N: the set of nets)
- The fanout FO(g) of a gate g are the successor gates of g:
  FO(g) = {g' | (g,g') $\in$ N}
- The cone CONE(g) of a gate g is the transitive fanin (TFI) of g and g itself
- The support SUPPORT(g) of a gate g are all inputs in its cone:
  SUPPORT(g) = CONE(g) $\cap$ I

Boolean Function Representation
Boolean Network

Example



FI(6) = {2,4}
FO(6) = {7,9}
CONE(6) = {1,2,4,6}
SUPPORT(6) = {1,2}

Boolean Function Representation
Boolean Network

- Circuit functions are defined recursively:

$$h_{g_i} = \begin{cases} x_i & \text{if } g_i \in I \\ f_{g_i}(h_{g_j},\dots,h_{g_k}),\ \ g_j,\dots,g_k \in FI(g_i) & \text{otherwise} \end{cases}$$

If G is implemented using physical gates with positive (bounded) delays for their evaluation, the computation of h_g depends in general on those delays.

Definition

A circuit C is called combinational if for each input assignment of C for t→∞ the evaluation of h_g for all outputs is independent of the internal state of C.

Proposition

A circuit C is combinational if it is acyclic. (converse not true!)

Boolean Function Representation
Boolean Network

General Boolean network:

- Vertex can have an arbitrary finite number of inputs and outputs

- Vertex can represent any Boolean function stored in different ways, such as:
  - SOPs (e.g. in SIS, a logic synthesis package)
  - BDDs (to be introduced)
  - AIGs (to be introduced)
  - truth tables
  - Boolean expressions read from a library description
  - other sub-circuits (hierarchical representation)

- The data structure allows general manipulations for insertion and deletion of vertices, pins (connection ports of vertices), and nets
  - general but far too slow for Boolean reasoning

Boolean Function Representation
Boolean Network

Specialized Boolean network:

- Non-canonical representation in general
  - computational effort of Boolean reasoning is due to this non-canonicity (c.f. BDDs)

- Vertices have fixed number of inputs (e.g. two)

- Vertex function is stored as label (e.g. OR, AND, XOR)

- Allow on-the-fly compaction of circuit structure
  - Support incremental, subsequent reasoning on multiple problems

---

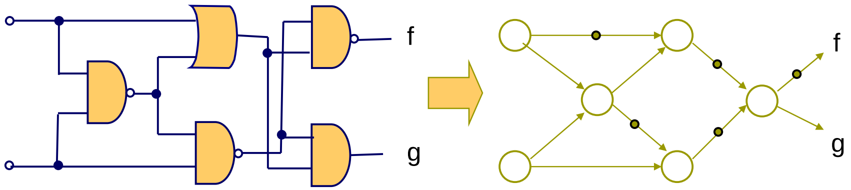
Boolean Function Representation
And-Inverter Graph

- AND-INVERTER graphs (AIGs)

  vertices: 2-input AND gates

  edges: interconnects with (optional) dots representing INVs

- Hash table to identify and reuse structurally isomorphic circuits



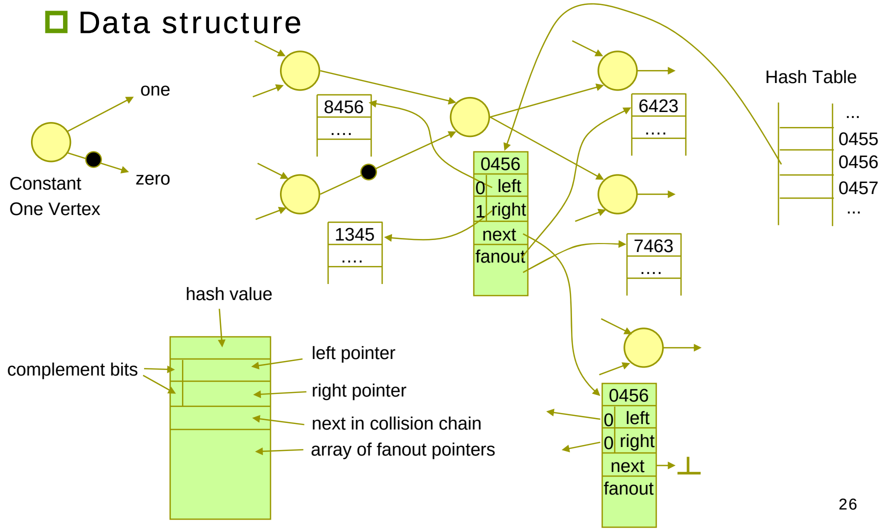
Boolean Function Representation
And-Inverter Graph

- Data structure for implementation
  - Vertex:
    - pointers (integer indices) to left- and right-child and fanout vertices
    - collision chain pointer
    - other data
  - Edge:
    - pointer or index into array
    - one bit to represent inversion
  - Global hash table holds each vertex to identify isomorphic structures
  - Garbage collection to regularly free un-referenced vertices

---

Boolean Function Representation
And-Inverter Graph

- Data structure



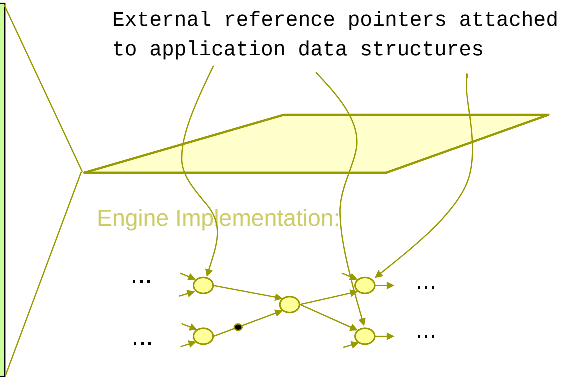
Boolean Function Representation
And-Inverter Graph

- AIG package for Boolean reasoning

Engine application:

- traverse problem data structure and build Boolean problem using the interface
- call SAT to make decision

```
Engine Interface:
void INIT()
void QUIT()
Edge VAR()
Edge AND(Edge p1,
         Edge p2)
Edge NOT(Edge p1)
Edge OR(Edge p1
        Edge p2)
...
int  SAT(Edge p1)
```


`External reference pointers attached to application data structures`

Boolean Function Representation
And-Inverter Graph

- Hash table look-up

```
Algorithm HASH_LOOKUP(Edge p1, Edge p2) {
  index = HASH_FUNCTION(p1,p2)
  p     = &hash_table[index]
  while(p != NULL) {
   if(p->left == p1 && p->right == p2) return p;
   p = p->next;
  }
  return NULL;
}
```

- Tricks:
  - keep collision chain sorted by the address (or index) of p
  - use memory locations (or array indices) in topological order for better cache performance

Boolean Function Representation
And-Inverter Graph

- AND operation

```
Algorithm AND(Edge p1,Edge p2){
  if(p1 == const1) return p2
  if(p2 == const1) return p1
  if(p1 == p2)     return p1
  if(p1 == ¬p2)    return const0
  if(p1 == const0 || p2 == const0) return const0

  if(RANK(p1) > RANK(p2)) SWAP(p1,p2)

  if((p = HASH_LOOKUP(p1,p2)) return p
  return CREATE_AND_VERTEX(p1,p2)
}
```

Boolean Function Representation
And-Inverter Graph

- NOT operation

```
Algorithm NOT(Edge p) {
  return TOOGLE_COMPLEMENT_BIT(p)
}
```

- OR operation

```
Algorithm OR(Edge p1,Edge p2){
  return (NOT(AND(NOT(p1),NOT(p2))))
}
```

Boolean Function Representation
And-Inverter Graph

- Cofactor operation

```
Algorithm POSITIVE_COFACTOR(Edge p,Edge v){
  if(IS_VAR(p)) {
    if(p == v) {
      if(IS_INVERTED(v) == IS_INVERTED(p)) return const1
      else                                 return const0
    }
    else                                   return p
  }
  if((c = GET_COFACTOR(p,v)) == NULL) {
    left  = POSITIVE_COFACTOR(p->left, v)
    right = POSITIVE_COFACTOR(p->right, v)
    c = AND(left,right)
    SET_COFACTOR(p,v,c)
  }
  if(IS_INVERTED(p)) return NOT(c)
  else               return c
}
```

Boolean Function Representation
And-Inverter Graph

- Similar algorithm for **NEGATIVE_COFACTOR**

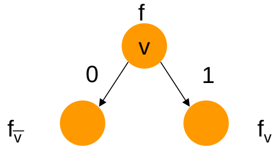
- Existential and universal quantifications can be built from AND, OR and COFACTORS

Exercise: Prove $(f \cdot g)_v = f_v \cdot g_v$ and $(\neg f)_v = \neg(f_v)$

Question: What is the worst-case complexity of performing quantifications over AIGs?

Boolean Function Representation
Binary Decision Diagram (BDD)

- A graphical representation of Boolean function
  - BDD is a Shannon cofactor tree:
    - $f = v\ f_v + v'\ f_{v'}$ (Shannon expansion)
    - vertices represent decision nodes (i.e. multiplexers) controlled by variables
    - leaves are constants “0” and “1”
    - two children of a vertex of f represent two subfunctions f_v and $f_{v'}$
  - Variable ordering restriction and reduction rules make (ROBDD) representation canonical



Boolean Function Representation
BDD – Canonicalization

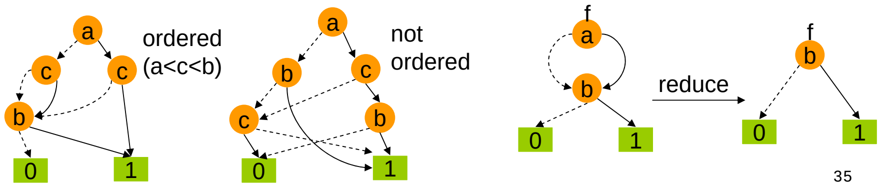
- General idea:
  - instead of exploring sub-cases by enumerating them in time, try to store sub-cases in memory
    - KEY: two hashing mechanisms:
      - unique table: find identical sub-cases and avoid replication
      - computed table: reduce redundant computation of sub-cases
- Represent logic functions as graphs (DAGs):
  - many logic functions can be represented compactly - usually better than SOPs
- Can be made canonical (ROBDD)
  - Shift the effort in a Boolean reasoning engine from SAT algorithm to data representation
- Many logic operations can be performed efficiently on BDD’s:
  - usually linear in size of input BDDs
  - tautology checking and complement operation are constant time
- BDD size critically depends on variable ordering

Boolean Function Representation
BDD – Canonicalization

- Directed acyclic graph (DAG)
  - one root node, two terminal-nodes, 0 and 1
  - each node has two children and is controlled by a variable
- Shannon cofactor tree, except reduced and ordered (ROBDD)
  - Ordered:
    - cofactor variables (splitting variables) in the same order along all paths

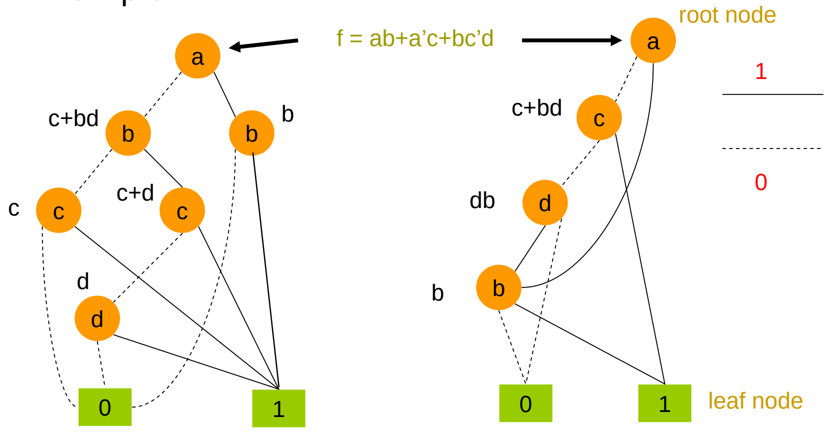
      $x_{i_1} < x_{i_2} < x_{i_3} < \dots < x_{i_n}$
  - Reduced:
    - any node with two identical children is removed
    - two nodes with isomorphic BDD’s are merged

      These two rules make any node in an ROBDD represent a distinct logic function



Boolean Function Representation
BDD

- Example



Same function with two different variable orders

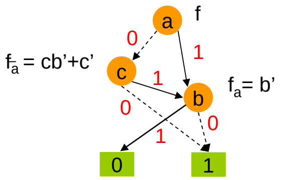
Boolean Function Representation
BDD – Canonicity of ROBDD

- Three components make ROBDD canonical (Bryant 1986):
  - unique nodes for constant “0” and “1”
  - identical order of case-splitting variables along each paths
  - a hash table that ensures
    - $(\text{node}(f_v) = \text{node}(g_v)) \wedge (\text{node}(f_{v'}) = \text{node}(g_{v'})) \Rightarrow \text{node}(f) = \text{node}(g)$

  and provides recursive argument that node(f) is unique when using the unique hash-table

Boolean Function Representation
BDD – Onset Counting

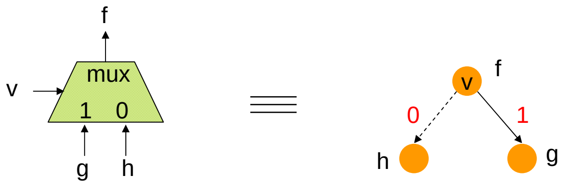
F = b’+a’c’ = ab’+a’cb’+a’c’ (all paths to the 1 node)



- By tracing all paths to the 1 node, we get a cover of pairwise disjoint cubes
- BDD does not explicitly enumerate all paths; rather it represents paths by a graph whose size is measures by its nodes
  - 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

Boolean Function Representation
BDD – ITE Operator

- Each BDD node can be written as a triplet: f = ite(*v*,g,h) = vg + v’h, where g = f_v and h = $f_{\bar{v}}$, meaning if *v* then g else h



(v is top variable of f)

Boolean Function Representation
BDD – ITE Operator

- ite(*f*,g,h) = fg + f’h
  - ITE operator can implement any two variable logic function. There are 16 such functions corresponding to all subsets of vertices of $\mathbf{B}^2$:

Table	Subset	Expression	Equivalent Form
0000	0	0	0
0001	AND(f, g)	f g	ite(f, g, 0)
0010	f > g	f g′	ite(f, g′, 0)
0011	f	f	f
0100	f < g	f′g	ite(f, 0, g)
0101	g	g	g
0110	XOR(f, g)	f ⊕ g	ite(f, g′, g)
0111	OR(f, g)	f + g	ite(f, 1, g)
1000	NOR(f, g)	(f + g)′	ite(f, 0, g′)
1001	XNOR(f, g)	f ⊕ g′	ite(f, g, g′)
1010	NOT(g)	g′	ite(g, 0, 1)
1011	f ≥ g	f + g′	ite(f, 1, g′)
1100	NOT(f)	f′	ite(f, 0, 1)
1101	f ≤ g	f′ + g	ite(f, g, 1)
1110	NAND(f, g)	(f g)′	ite(f, g′, 1)
1111	1	1	1

Boolean Function Representation
BDD – ITE Operator

- Recursive operation of ITE

$$
\begin{aligned}
&\mathrm{Ite}(f,g,h)\\
&= f\,g + f'\,h\\
&= v\,(f\,g + f'\,h)_v + v'\,(f\,g + f'\,h)_{v'}\\
&= v\,(f_v\,g_v + f'_v\,h_v) + v'\,(f_{v'}\,g_{v'} + f'_{v'}\,h_{v'})\\
&= \mathrm{ite}(v,\ \mathrm{ite}(f_v,g_v,h_v),\ \mathrm{ite}(f_{v'},g_{v'},h_{v'}))
\end{aligned}
$$

  - Let v be the top-most variable of BDDs f, g, h

Boolean Function Representation
BDD – ITE Operator

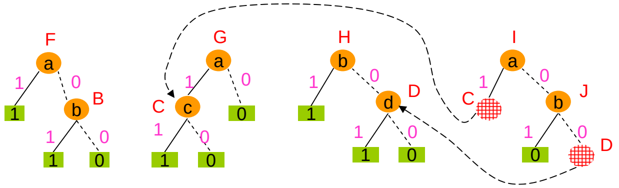
- Recursive computation of ITE

```
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
}
```

Boolean Function Representation
BDD – ITE Operator

- Example



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

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

Check: $F = a + b$
$G = ac$
$H = b + d$
$ite(F, G, H) = (a + b)(ac) + a'b'(b + d) = ac + a'b'd$

Boolean Function Representation
BDD – ITE Operator

- Tautology checking using ITE

```
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(f_v,g_v,h_v)
  if(i == NC) {
    INSERT_COMPUTED_TABLE(NC, HASH_KEY{f,g,h}) // special table!!
    return NC
  }
  e = ITE_CONSTANT(f_v',g_v',h_v')
  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;
}
```

Boolean Function Representation
BDD – ITE Operator

- Composition using ITE
  - Compose is an important operation, e.g. for building the BDD of a circuit backwards, Compose(*F*, *v*, *G*) : *F*(*v*, *x*) → *F*(*G*(*x*), *x*), means substitute *v* = *G*(*x*)

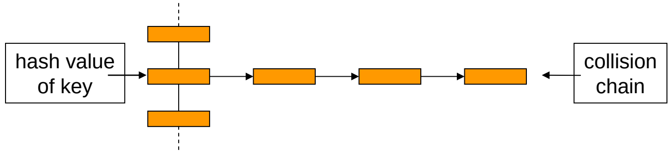
```
Algorithm COMPOSE(F,v,G) {
  if(TOP_VARIABLE(F) > v)  return F  // F does not depend on v
  if(TOP_VARIABLE(F) == v) return ITE(G,F1,F0)
  i = COMPOSE(F1,v,G)
  e = COMPOSE(F0,v,G)
  return ITE(TOP_VARIABLE(F),i,e)
}
```

Note:
1. `F1` and `F0` are the 1-child and 0-child of `F`, respectively
2. `G`, `i`, `e` are not functions of `v`
3. If TOP_VARIABLE of `F` is `v`, then ITE(`G`, `F1`, `F0`) does the replacement of `v` by `G`

Boolean Function Representation
BDD – Implementation Issues

Unique table:

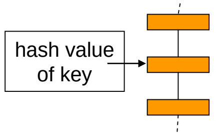
- avoids duplication of existing nodes
  - Hash-Table: hash-function(key) = value
  - identical to the use of a hash-table in AND/INVERTER circuits


hash value of key

collision chain

Computed table:

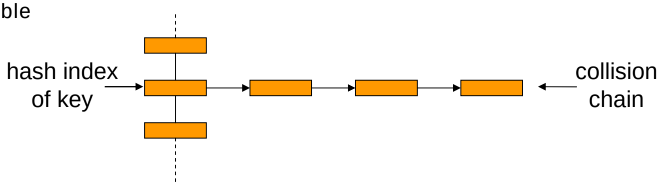
- avoids re-computation of existing results


hash value of key

No collision chain

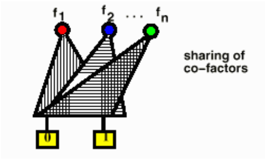
Boolean Function Representation
BDD – Implementation Issues

- Unique table


hash index of key

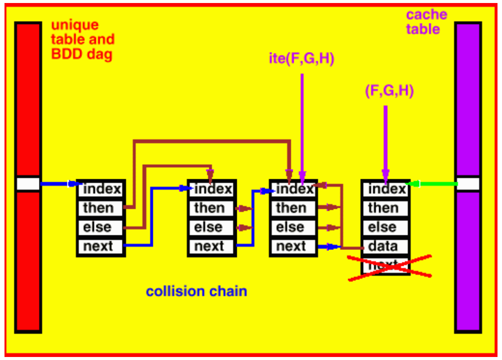
collision chain

- Before a node ite(*v*, *g*, *h*) is added to BDD database, 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* = ite(*v*, *g*, *h*) exists iff ite(*v*, *g*, *h*) is in the unique-table. There is only one pointer for ite(*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



Boolean Function Representation
BDD – Implementation Issues

- 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)

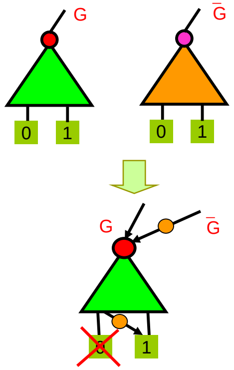


Boolean Function Representation
BDD – Implementation Issues

- Use of computed table
  - BDD packages often 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`)
    - 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

Boolean Function Representation
BDD – Implementation Issues

- Complemented edges
  - Combine inverted functions by using complemented edge
    - similar to AIG
    - reduces memory requirements
    - more importantly, makes operations NOT, ITE more efficient

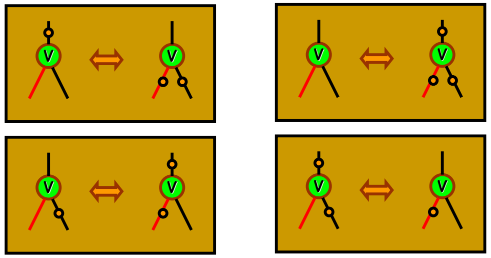


two different DAGs

only one DAG using complement pointer

Boolean Function Representation
BDD – Implementation Issues

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



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

Boolean Function Representation
BDD – Implementation Issues

- Complemented edges

Standard triples:
$\mathit{ite}(F, F, G) \Rightarrow \mathit{ite}(F, 1, G)$
$\mathit{ite}(F, G, F) \Rightarrow \mathit{ite}(F, G, 0)$
$\mathit{ite}(F, G, \neg F) \Rightarrow \mathit{ite}(F, G, 1)$
$\mathit{ite}(F, \neg F, G) \Rightarrow \mathit{ite}(F, 0, G)$

To resolve equivalences:
$\mathit{ite}(F, 1, G) \equiv \mathit{ite}(G, 1, F)$
$\mathit{ite}(F, 0, G) \equiv \mathit{ite}(\neg G, 1, \neg F)$
$\mathit{ite}(F, G, 0) \equiv \mathit{ite}(G, F, 0)$
$\mathit{ite}(F, G, 1) \equiv \mathit{ite}(\neg G, \neg F, 1)$
$\mathit{ite}(F, G, \neg G) \equiv \mathit{ite}(G, F, \neg F)$

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:

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

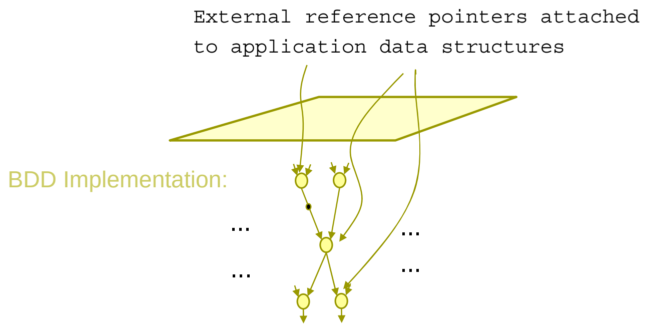
Choose the one such that the first and second argument of *ite* should not be complement edges (i.e. the first one above)

Boolean Function Representation
BDD – Implementation Issues

- Variable ordering – static
  - variable ordering is computed up-front based on the problem structure
  - works well for many practical combinational functions
    - general scheme: control variables first
    - DFS order is good for most cases
  - works bad for unstructured problems
    - e.g. using BDDs to represent arbitrary sets
  - lots of ordering algorithms
    - simulated annealing, genetic algorithms
    - give better results but extremely costly

Boolean Function Representation
BDD – Implementation Issues

- Variable ordering – dynamic
  - 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

---

Boolean Function Representation
BDD – Implementation Issues

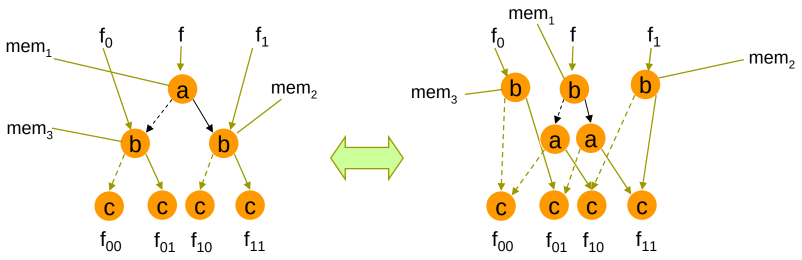
- Variable ordering – dynamic

  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



Boolean Function Representation
BDD – Implementation Issues

- Variable ordering – dynamic
  - 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
  - grouping of BDD variables:
    - for many applications, grouping variables gives better ordering
      - e.g. current state and next state variables in state traversal
    - grouping variables for sifting

Boolean Function Representation
BDD – Implementation Issues

- Garbage collection
  - Important to free and reuse memory of unused BDD nodes including
    - those explicitly freed by an external `bdd_free` operation
    - those temporary 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 referenced
      - decrement when node gets de-referenced
      - take care of counter-overflow
    - 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

Boolean Function Representation
BDD – Implementation Issues

- Garbage collection
  - Timing is crucial because garbage collection is expensive
    - immediately when node gets freed
      - bad because dead nodes get often reincarnated in subsequent operations
    - regular garbage collections based on statistics obtained during BDD operations
  - Computed-table must be cleared since not used in reference mechanism
  - Improving memory locality and therefore cache behavior

Boolean Function Representation
BDD – Variants

- MDD: Multi-valued DD
  - have more then two branches
  - can be implemented using a regular BDD package with binary encoding
    - the binary variables for one MV variable do not have to stay together and thus potentially better ordering

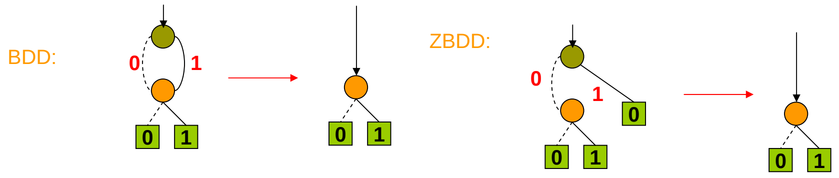
- ADD: (Algebraic BDDs) MTBDD
  - multi-terminal BDDs
  - decision tree is binary
  - multiple leaves, including real numbers, sets or arbitrary objects
  - efficient for matrix computations and other non-integer applications

- FDD: Free-order BDD
  - variable ordering differs
  - not canonical anymore

- …

Boolean Function Representation
BDD – Variants

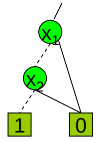
- Zero suppressed BDD (ZDD)
  - ZBDDs were invented by Minato to efficiently represent sparse sets. They have turned out to be useful in implicit methods for representing primes (which usually are a sparse subset of all cubes).
  - Different reduction rules:
    - BDD: eliminate all nodes where then 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.



Boolean Function Representation BDD – Variants

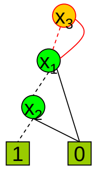
Theorem: ZBDDs are canonical given a variable ordering and the support set

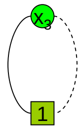
Example



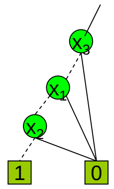
BDD

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

Boolean Function Representation Summary

- Sum of products
  - Good for circuit synthesis
- Product of sums
  - Good for Boolean reasoning
- Boolean network
  - Generic network
    - Good for multi-level circuit synthesis
  - And-inverter graph
    - Good for Boolean reasoning
- Binary decision diagram
  - Good for Boolean reasoning

Boolean Reasoning

Reading:
Logic Synthesis in a Nutshell
Section 2

most of the following slides are by
courtesy of Andreas Kuehlmann

Boolean Reasoning Satisfiability (SAT)

- Boolean reasoning engines need:
  - a data structure to represent problem instances
  - a decision procedure to decide about SAT or UNSAT

- Fundamental tradeoff
  - canonical data structure (e.g. truth table, ROBDD)
    - data structure uniquely represents function
    - decision procedure is trivial (e.g., just pointer comparison)
    - Problem: size of data structure is in general exponential

  - non-canonical data structure (e.g. AIG, CNF)
    - systematic search for satisfying assignment
    - size of data structure is linear
    - Problem: decision may take an exponential amount of time

Boolean Reasoning
SAT

- Basic SAT algorithms:
  - branch and bound algorithm
    - branching on the assignments of primary inputs only or those of all variables
      - E.g. PODEM vs. D-algorithms in ATPG
- Basic data structures:
  - circuits or CNF formulas
  - SAT on circuits is **identical** to the justification part in ATPG
    - 1st half of ATPG: justification
      - find an input assignment that forces an internal signal to a required value
    - 2nd half of ATPG: propagation
      - make a signal change at an internal signal observable at some outputs (can be easily formulated as SAT over CNF formulas)

Boolean Reasoning
SAT vs. Tautology

- SAT:
  - find a truth assignment to the inputs making a given Boolean formula true
  - NP-complete
- Tautology:
  - find a truth assignment to the inputs making a given Boolean formula false
  - coNP-complete
- SAT and Tautology are dual to each other
  - checking SAT on formula φ = checking Tautology on formula $\neg\varphi$, and vice versa

---

Boolean Reasoning
SAT – AIG-based Decision Procedure

- General Davis-Putnam procedure
  - search for consistent assignment to entire cone of requested vertex in AIG by systematically trying all combinations (may be partial)
  - keep a queue of vertices that remain to be justified
    - pick decision vertex from the queue and case split on possible assignments
    - for each case
      - propagate as many implications as possible
      - generate more vertices to be justified
      - if conflicting assignment encountered, undo all implications and backtrack
      - recur to next vertex from queue

Boolean Reasoning
SAT – AIG-based Decision Procedure

- General Davis-Putnam procedure

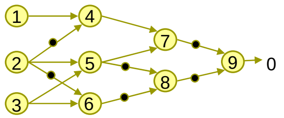
```
Algorithm SAT(Edge p) {
    queue = INIT_QUEUE(p)
    if(!IMPLY(p)) return FALSE
    return JUSTIFY(queue)
}

Algorithm JUSTIFY(queue) {
  if(QUEUE_EMPTY(queue)) return TRUE
  mark = ASSIGNMENT_MARK()
  v = QUEUE_NEXT(queue)  // decision vertex
  if(IMPLY(NOT(v)) {
    if(JUSTIFY(queue)) return TRUE
  }                                  // conflict
  UNDO_ASSIGNMENTS(mark)
  if(IMPLY(v)) {
    if(JUSTIFY(queue)) return TRUE
  }                                  // conflict
  UNDO_ASSIGNMENTS(mark)
  return FALSE
}
```

Boolean Reasoning
SAT – AIG-based Decision Procedure

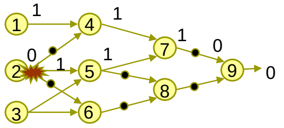
- Example

SAT(NOT(9))??



Queue	Assignments
9	9

1st case for 9:



Queue	Assignments
9	9 7 4 5 1 2

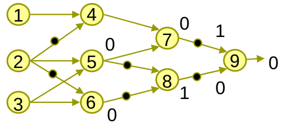
conflict !
- undo all assignments
- backtrack

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Example (cont'd)

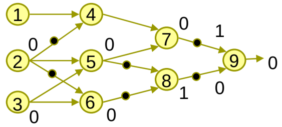
2nd case for 9:

Note:
vertex 7 is justified
by 8->5->7



Queue	Assignments
5 6	9 7 8 5 6

1st case for 5:

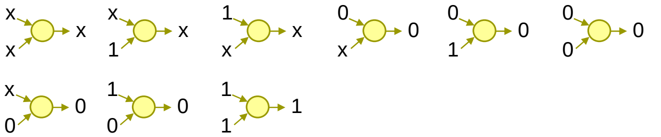


Queue	Assignments
	9 7 8 5 6 2 3

Solution cube: 1 = x, 2 = 0, 3 = 0

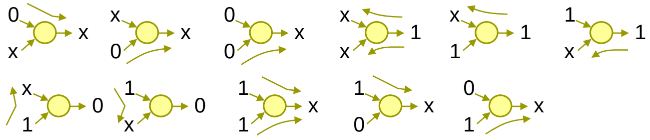
Boolean Reasoning
SAT – AIG-based Decision Procedure

- Implication
  - Fast implication procedure is key for efficient SAT solver!
    - don't move into circuit parts that are not sensitized to current SAT problem
    - detect conflicts as early as possible
  - Table lookup implementation (27 cases):
    - No-implication cases:

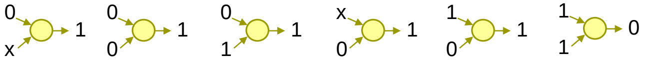


Boolean Reasoning
SAT – AIG-based Decision Procedure

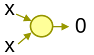
- Implication (cont'd)
  - Table lookup implementation (27 cases):
    - Implication cases:



- Conflict cases:

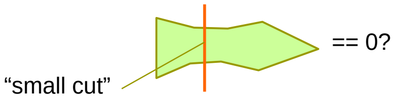


- Split case:



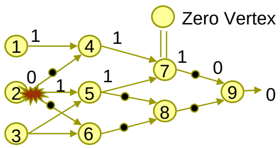
Boolean Reasoning
SAT – AIG-based Decision Procedure

- Case split
  - Different heuristics work well for particular problem classes
  - Often depth-first heuristic is good because it generates conflicts quickly
  - Mixture of depth-first and breadth-first schedule
  - Other heuristics:
    - pick the vertex with the largest fanout
    - count the polarities of the fanout separately and pick the vertex with the highest count in either polarity
    - run a full implication phase on all outstanding case splits and count the number of implications one would get
    - pick vertices that are involved in small cut of the circuit



Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning
  - Learning is the process of adding "shortcuts" to the circuit structure that avoids case splits
    - static learning:
      - global implications are learned
    - dynamic learning:
      - learned implications only hold in current part of the search tree
  - Learned implications are stores as additional network
  - Example (cont'd)
    - 1st case for vertex 9 lead to conflict
    - If we were to try the same assignment again (e.g. for the next SAT call), we would get the same conflict => merge vertex 7 with zero-vertex



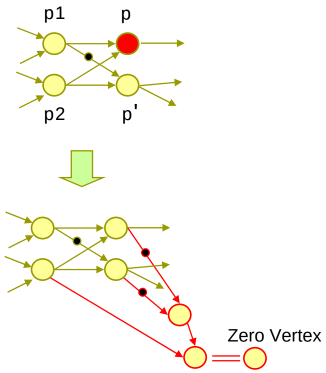
- if rehashing is invoked
vertex 9 is simplified and
and merged with vertex 8

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – static
  - Implications that can be learned structurally from the circuit
    - Add learned structure as circuit

Use hash table to find structure in circuit:

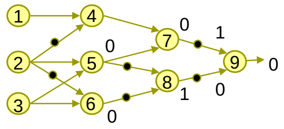
```
Algorithm CREATE_AND(p1,p2) {
  . . . // create new vertex p
  if((p'=HASH_LOOKUP(p1,NOT(p2))) {
    LEARN(((p=0)&(p'=0)) ⇒ (p1=0))
  }
  if((p'=HASH_LOOKUP(NOT(p1),p2)) {
    LEARN(((p=0)&(p'=0)) ⇒ (p2=0))
  }
}
```



Boolean Reasoning
SAT – AIG-based Decision Procedure

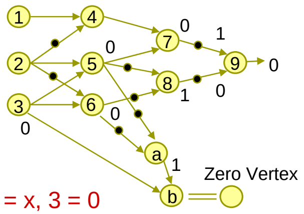
- Example (cont'd)

2nd case for 9 (original):



Queue	Assignments
5 6	9 7 8 5 6

2nd case for 9 (with static learning):



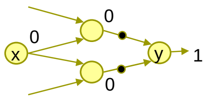
Queue	Assignments
	9 7 8 5 6 a 3

Solution cube: 1 = x, 2 = x, 3 = 0

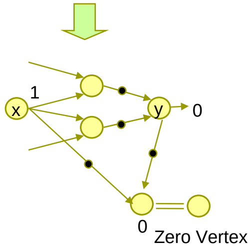
Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – static
  - Other learning based on contra-positive: if (P ⇒ Q), then (¬Q ⇒ ¬P)

```
foreach vertex v {
  mark = ASSIGNMENT_MARK()
  IMPLY(v)
  LEARN_IMPLICATIONS(v)
  UNDO_ASSIGNMENTS(mark)
  IMPLY(NOT(v))
  LEARN_IMPLICATIONS(NOT(v))
  UNDO_ASSIGNMENTS(mark)
}
```



$$((x=0)\Rightarrow(y=1))\Rightarrow((y=0)\Rightarrow(x=1))$$



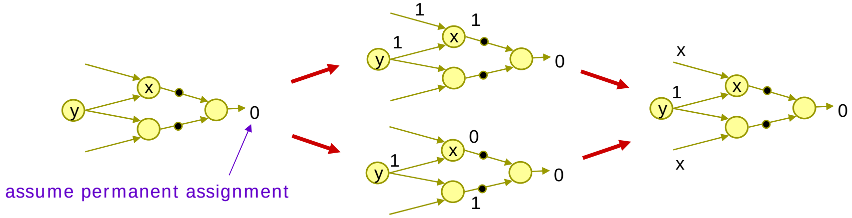
  - Problem: learned implications are far too many
    - solution: restrict learning to non-trivial implications and filter redundant implications

---

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – static and recursive
  - Compute the set of all implications for both case splits on level *i*
    - Static learning of constants, equivalences
  - Intersect both split cases to learn for level *i*–1

$$((x=1)\Rightarrow(y=1)\wedge(x=0)\Rightarrow(y=1)) \quad\Rightarrow\quad (y=1)$$



assume permanent assignment

  - Apply learning recursively until all case splits exhausted
    - recursive learning is complete but very expensive in practice for levels > 2, 3
    - restrict learning level to fixed number→ becomes incomplete

---

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – static and recursive

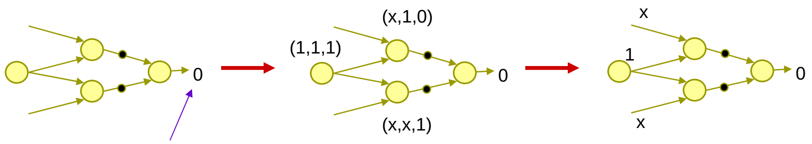
```
Algorithm RECURSIVE_LEARN(int level) {
  if(v = PICK_SPLITTING_VERTEX()) {
    mark = ASSIGNMENT_MARK()
    IMPLY(v)
    IMPL1 = RECURSIVE_LEARN(level+1)
    UNDO_ASSIGNMENTS(mark)
    IMPLY(NOT(v))
    IMPL0 = RECURSIVE_LEARN(level+1)
    UNDO_ASSIGNMENTS(mark)
    return IMPL1 ∩ IMPL0
  }
  else {    // completely justified
    return IMPLICATIONS
  }
}
```

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – dynamic
  - Learn implications in a sub-tree of searching
    - cannot simply add permanent structure because not globally valid
      - add and remove learned structure (expensive)
      - add branching condition to the learned implication
        - of no use unless we prune the condition (conflict learning)
      - use implication and assignment mechanism to assign and undo assigns
        - e.g., dynamic recursive learning with fixed recursion level
    - Dynamic learning of equivalence relations (Stalmarck procedure)
      - learn equivalence relations by dynamically rewriting the formula

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – dynamic
  - Efficient implementation of dynamic recursive learning with level 1:
    - consider both sub-cases in parallel
    - use 27-valued logic in the `IMPLY` routine
      `(level0-value, level1-choice1, level1-choice2)`
      `({0,1,x}, {0,1,x}, {0,1,x})`
    - automatically set learned values for level0 if both level1 choices agree, e.g.,

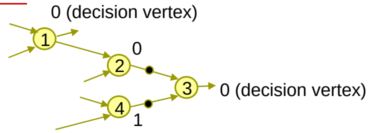


assume temporary assignment

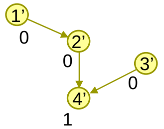
Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – conflict-based (c.f. structure-based)
  - Idea: Learn the situation under which a particular conflict occurred and assert it to 0
    - `IMPLY` will use this “shortcut” to detect similar conflict earlier
  - Definition: An implication graph is a directed Graph I(G’,E), G’ ⊆ G are the gates of C with assigned values $v_g \neq$ unknown, E ⊆ G’×G’ are the edges, where each edge $(g_i,g_j) \in E$ reflects an implication for which an assignment of gate g_i leads to the assignment of gate g_j.

Circuit:

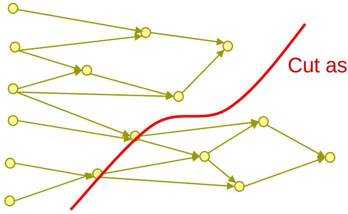


Implication graph:



Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – conflict-based
  - The roots (w/o fanin-edges) of the implication graph correspond to the decision vertices, the leaves correspond to the implication frontier



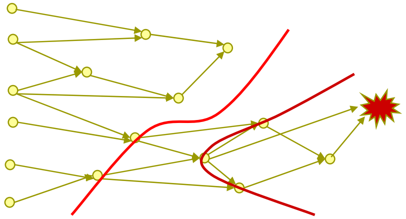
Cut assignment (C_i)

  - There is a strict implication order in the graph from the roots to the leaves
    - We can completely cut the graph at any point and identify value assignments to the cut vertices, we result in identical implications toward the leaves

$C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_{n-1} \Rightarrow C_n$ (C_1: decision vertices)

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Learning – conflict-based
  - If an implication leads to a conflict, any cut assignment in the implication graph between the decision vertices and the conflict will result in the same conflict!

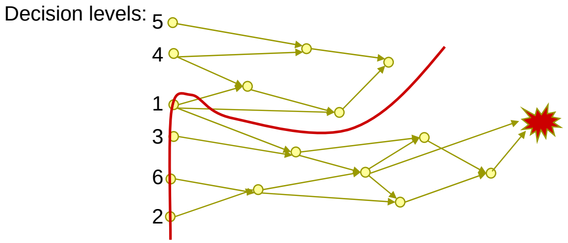


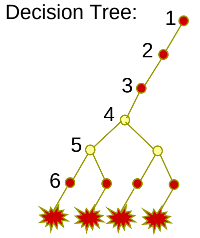
($C_i \Rightarrow$ Conflict) $\Rightarrow$ (NOT(Conflict) $\Rightarrow$ NOT(C_i))

  - We can learn the complement of the cut assignment as circuit
    - find minimal cut in the implication graph I (costs less to learn)
    - find dominator vertex if exists
    - restrict size of cuts to be learned, otherwise exponential blow-up

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Non-chronological backtracking
  - If we learned only cuts on decision vertices, only the decision vertices that are in the support of the conflict are needed



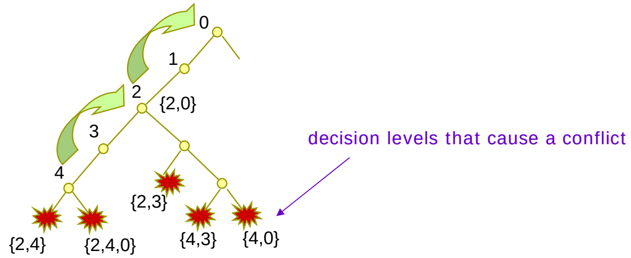


  - The conflict is **fully symmetric** with respect to the unrelated decision vertices!!
    - Learning the conflict would prevent checking the symmetric parts again

      BUT: It is too expensive to learn all conflicts (any cut)

Boolean Reasoning
SAT – AIG-based Decision Procedure

- Non-chronological backtracking
  - We can still avoid exploring symmetric parts of the decision tree by tracking the decision support vertices of a conflict
    - If no conflict of the first choice on a decision vertex depends on that vertex, the other choice will result in symmetric conflicts and their evaluation can be skipped!
  - By tracking the implications of the decision vertices we can skip decision levels during backtrack



Boolean Reasoning
SAT – CNF-based Decision Procedure

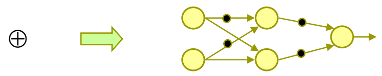
- CNF
  - Product-of-Sums (POS) representation of Boolean function
  - Describes solution using a set of constraints
    - very handy in many applications because new constraints can be simply added to the list of existing constraints
    - very common in AI community
  - Example

    $\varphi = (a+b'+c)(a'+b+c)(a+b'+c')(a+b+c)$

  - SAT on CNF (POS) ⇔ TAUTOLOGY on DNF (SOP)

Boolean Reasoning
SAT – CNF-based Decision Procedure

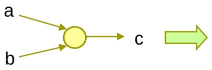
- Circuit to CNF conversion
  - Encountered often in practical applications
  - Naive conversion from circuit to CNF:
    - multiply out expressions of circuit until two level structure
    - Example: $y = x_1 \oplus x_2 \oplus x_2 \oplus \dots \oplus x_n$ (parity function)
      - circuit size is linear in the number of variables



      - generated chess-board Karnaugh map
      - CNF (or DNF) formula has 2^{n-1} terms (exponential in the # vars)
  - Better approach:
    - introduce one variable per circuit vertex
    - formulate the circuit as a conjunction of constraints imposed on the vertex values by the gates
    - uses more variables but size of formula is linear in the size of the circuit

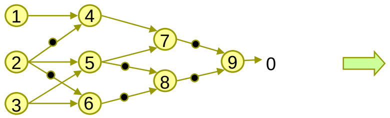
Boolean Reasoning
SAT – CNF-based Decision Procedure

- Circuit to CNF conversion
  - Example
    - Single gate



(¬a + ¬b + c)(a + ¬c)(b + ¬c)

    - Connected gates



(¬1 + 2 + 4)(1 + ¬4)(¬2 + ¬4)
(¬2 + ¬3 + 5)(2 + ¬5)(3 + ¬5)
(2 + ¬3 + 6)(¬2 + ¬6)(3 + ¬6)
(¬4 + ¬5 + 7)(4 + ¬7)(5 + ¬7)
(5 + 6 + 8)(¬5 + ¬8)(¬6 + ¬8)
(7 + 8 + 9)(¬7 + ¬9)(¬8 + ¬9)
(¬9)

Justify to “0”

Boolean Reasoning
SAT – CNF-based Decision Procedure

- DPLL procedure

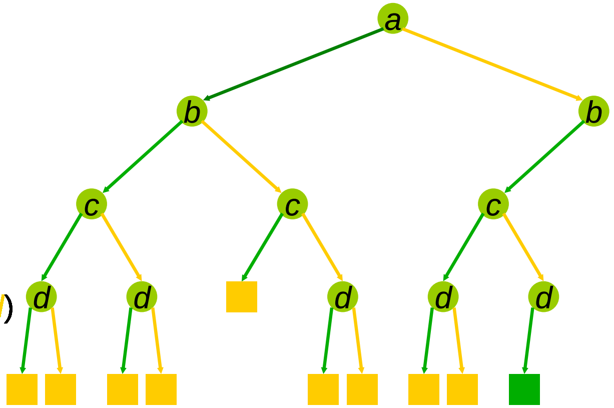
```
Algorithm DPLL() {
  while ChooseNextAssignment() {
     while Deduce() == CONFLICT  {
        blevel = AnalyzeConflict();
        if (blevel < 0) return UNSATISFIABLE;
        else Backtrack(blevel);
     }
   }
   return SATISFIABLE;
}
```

ChooseNextAssignment picks next decision variable and assignment
Deduce does Boolean Constraint Propagation (implications)
AnalyzeConflict backprocesses from conflict and produces learnt-clause
Backtrack undoes assignments

Boolean Reasoning
SAT – CNF-based Decision Procedure

- DPLL (basic case splitting)

1 (*a* + *b* + *c*)
2 (*a* + *b* + ¬*c*)
3 (¬*a* + *b* + ¬*c*)
4 (*a* + *c* + *d*)
5 (¬*a* + *c* + *d*)
6 (¬*a* + *c* + ¬*d*)
7 (¬*b* + ¬*c* + ¬*d*)
8 (¬*b* + ¬*c* + *d*)



Source: Karem A. Sakallah, Univ. of Michigan

Boolean Reasoning
SAT – CNF-based Decision Procedure

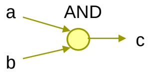
- Implication
  - Implications in a CNF formula are caused by unit clauses
    - A unit clause is a CNF term for which all variables except one are assigned
      - the value of that clause can be implied immediately
    - Example

      clause (a+¬b+c)

      (a=0)(b=1)⇒(c=1)

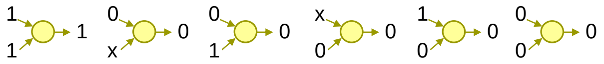
Boolean Reasoning
SAT – CNF-based Decision Procedure

- Implication
  - Example

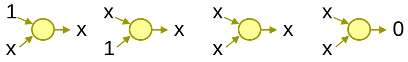


(¬a+¬b+c)(a+¬c)(b+¬c)

Non-implication cases:



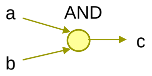
All clauses satisfied



Not all clauses satisfied (avoid exploring this part)

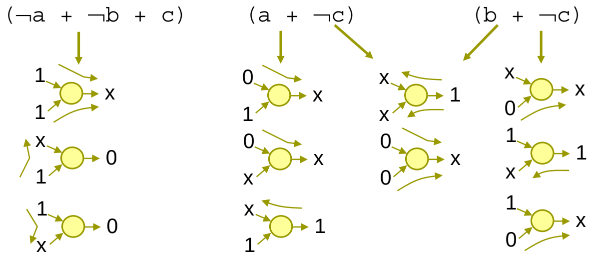
Boolean Reasoning
SAT – CNF-based Decision Procedure

- Implication
  - Example (cont'd)



(¬a+¬b+c)(a+¬c)(b+¬c)

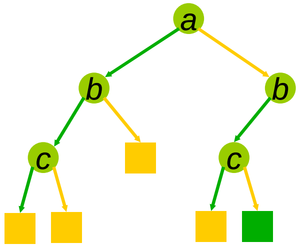
Implication cases:

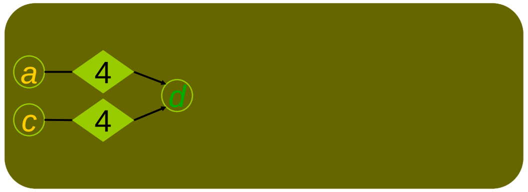


Boolean Reasoning
SAT – CNF-based Decision Procedure

- DPLL (w/ implication)

1 (a + b + c)
2 (a + b + ¬c)
3 (¬a + b + ¬c)
4 (a + c + d)
5 (¬a + c + d)
6 (¬a + c + ¬d)
7 (¬b + ¬c + ¬d)
8 (¬b + ¬c + d)





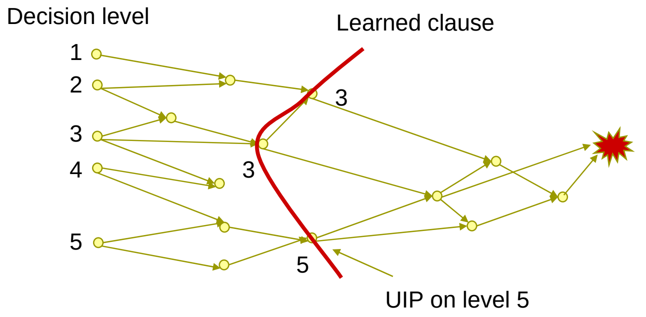
Source: Karem A. Sakallah, Univ. of Michigan

Boolean Reasoning
SAT – CNF-based Decision Procedure

- Conflict-based learning
  - Important detail for cut selection:
    - During implication processing, record decision level for each implication
    - At conflict, select earliest cut such that exactly one node of the implication graph lies on current decision level
      - Either decision variable itself
      - Or UIP ("unique implication point") that represents a dominator node for current decision level in conflict graph
  - By selecting such cut, implication processing will automatically flip decision variable (or UIP variable) to its complementary value

Boolean Reasoning
SAT – CNF-based Decision Procedure

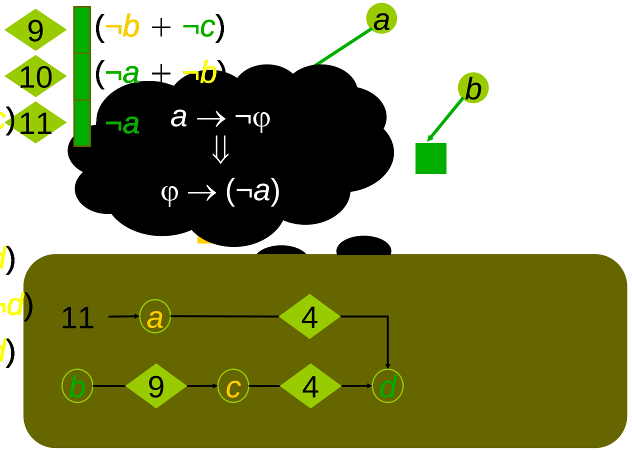
- Conflict-based learning
  - UIP detection
    - Store with each implication the decision level, and a time stamp (integer that is incremented after each decision)
      - UIP on decision level l has the property that all following implications towards the conflict have a larger time stamp
      - When back processing from conflict, put all implications that are to be processed on heap, keeping the one with smallest time stamp on top
      - If during processing there is only one variable on current decision level on heap then that variable must be a UIP



Boolean Reasoning
SAT – CNF-based Decision Procedure

- DPLL (conflict-based learning)

1 (a + b + c)
2 (a + b + ¬c)
3 (¬a + b + ¬c)
4 (a + c + d)
5 (¬a + c + d)
6 (¬a + c + ¬d)
7 (¬b + ¬c + ¬d)
8 (¬b + ¬c + d)
9 (¬b + ¬c)
10 (¬a + ¬b)
11 ¬a



Source: Karem A. Sakallah, Univ. of Michigan

Boolean Reasoning
SAT – CNF-based Decision Procedure

- Implementation issues
  - Clauses are stores in arrays
  - Track change-sensitive clauses (two-literal watching)
    - all literals but one assigned -> implication
    - all literals but two assigned -> clause is sensitive to a change of either literal
    - all other clauses are insensitive and do not need to be observed
  - Learning:
    - learned implications are added to the CNF formula as additional clauses
      - limit the size of the clause
      - limit the “lifetime” of a clause, will be removed after some time
  - Non-chronological back-tracking
    - similar to circuit case

Boolean Reasoning
SAT – CNF-based Decision Procedure

- Implementation issues (cont’d)
  - Random restarts:
    - stop after a given number of backtracks
      - start search again with modified ordering heuristic
      - keep learned structures !
    - very effective for satisfiable formulas, often also effective for unsat formulas
  - Learning of equivalence relations:
    - if (a ⇒ b) ∧ (b ⇒ a), then (a = b)
    - very powerful for formal equivalence checking
  - Incremental SAT solving
    - solving similar CNF formulas in a row
    - share learned clauses