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. $\mathbb{B} = \{0, 1\}$)

Boolean Space

- $\mathbb{B} = \{0, 1\}$
- $\mathbb{B}^2 = \{0, 1\} \times \{0, 1\} = \{00, 01, 10, 11\}$

Karnaugh Maps:

- $\mathbb{B}^0$
- $\mathbb{B}^1$
- $\mathbb{B}^2$
- $\mathbb{B}^3$
- $\mathbb{B}^4$

Boolean Lattices:
Boolean Function

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

Example

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

- Onset of \( f \), denoted as \( f^1 \), is \( f^1 = \{ v \in B^n | f(v) = 1 \} \).
  - If \( f^1 = B^n \), \( f \) is a tautology.

- Offset of \( f \), denoted as \( f^0 \), is \( f^0 = \{ v \in B^n | f(v) = 0 \} \).
  - If \( f^0 = B^n \), \( f \) is unsatisfiable. Otherwise, \( f \) is satisfiable.

- \( f^1 \) and \( f^0 \) are sets, not functions!

- Boolean functions \( f \) and \( g \) are equivalent if \( \forall v \in B^n, f(v) = g(v) \) where \( v \) is a truth assignment or Boolean valuation.

- A literal is a Boolean variable \( x \) or its negation \( x' \) (or \( x, \neg x \)) 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 $B^n$.
- There are $2^{2^n}$ distinct Boolean functions.
  - Each subset $f^1 \subseteq B^n$ of vertices in $B^n$ forms a distinct Boolean function $f$ with onset $f^1$.

$$\begin{array}{c|ccc|}
  x_1 & x_2 & x_3 & f \\
  \hline
  0 & 0 & 0 & 1 \\
  0 & 0 & 1 & 0 \\
  0 & 1 & 0 & 1 \\
  0 & 1 & 1 & 0 \\
  1 & 0 & 0 & 1 \\
  1 & 0 & 1 & 0 \\
  1 & 1 & 0 & 1 \\
  1 & 1 & 1 & 0 \\
\end{array}$$

Boolean Operations

Given two Boolean functions:

- $f : B^n \rightarrow B$
- $g : B^n \rightarrow B$

- $h = f \wedge g$ from AND operation is defined as
  $h^1 = f^1 \cap g^1$; $h^0 = B^n \setminus h^1$

- $h = f \vee g$ from OR operation is defined as
  $h^1 = f^1 \cup g^1$; $h^0 = B^n \setminus h^1$

- $h = \neg f$ from COMPLEMENT operation is defined as
  $h^1 = f^0$; $h^0 = f^1$
Cofactor and Quantification

Given a Boolean function:
\[ f : B^n \rightarrow B \], 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 \( h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \lor f(x_1, x_2, \ldots, 1, \ldots, x_n) \)

- **Universal quantification** over variable \(x_i\)
  \[ h = \forall x_i. \ f \] is defined as \( h = f(x_1, x_2, \ldots, 0, \ldots, x_n) \land f(x_1, x_2, \ldots, 1, \ldots, x_n) \)

- **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 $f : \mathbb{B}^n \rightarrow \mathbb{B}$ is a tabulation of its value at each of the $2^n$ vertices of $\mathbb{B}^n$.

In other words the truth table lists all minterms.

Example: $f = a'b'c'd + a'b'cd + a'bc'd + ab'c'd + ab'cd + abc'd + abcd' + abcd$

<table>
<thead>
<tr>
<th>$abcd$</th>
<th>$f$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>1</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>1</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>

The truth table representation is:
- impractical for large $n$
- canonical

If two functions are the same, then their canonical representations are isomorphic.

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 "\cdot" formula (AND operator)
  | \neg formula (complement)

Example:

$f = (x_1 \cdot x_2) + (x_3) + \neg(\neg(x_4 \cdot \neg x_1))$

typically "\cdot" is omitted and "(', ')" and \neg are simply reduced by priority,

e.g. $f = x_1 \cdot x_2 + x_3 + x_4 \neg x_1$
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 ...

\[ f = x_1 \]
\[ f = \overline{x_1}x_2 \]
\[ f = \overline{x_1}x_2x_3 \]

If \( C \subseteq f^1 \), \( C \) the onset of a cube \( c \), then \( c \) is an implicant of \( f \)

If \( C \subseteq B^n \), and \( c \) has \( k \) literals, then \( |C| = 2^{n-k} \), i.e., \( C \) has \( 2^{n-k} \) elements

Example

\[ c = xy' \] \((c:B^3 \rightarrow B)\), \( C = \{100, 101\} \subseteq B^3 \)
\( k = 2 \), \( n = 3 \) \quad \( |C| = 2 = 2^{3-2} \)

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

A **Boolean network** is a directed graph C(G,N) where G are the gates and N ⊆ (G×G) are the directed edges (nets) connecting the gates.

Some of the vertices are designated:

- **Inputs:** I ⊆ G
- **Outputs:** O ⊆ G
- I ∩ O = ∅

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: $SUPPORT(g) = CONE(g) \cap I$

Example

![Diagram of Boolean Network]

- $FI(6) = \{2,4\}$
- $FO(6) = \{7,9\}$
- $CONE(6) = \{1,2,4,6\}$
- $SUPPORT(6) = \{1,2\}$
Boolean Function Representation

Boolean Network

- Circuit functions are defined recursively:
  \[ h_{g_i} = \begin{cases} x_i & \text{if } g_i \in I \\ f_{g_i} (h_{g_j}, \ldots, h_{g_k}), & g_j, \ldots, g_k \in FI(g_i) \text{ otherwise} \end{cases} \]

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 \( t \to \infty \) 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!)

---

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

**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

---

**Diagram:**

- **Constant One Vertex**
  - zero
  - one

- **Hash Table**
  - ... 0455 0456 0457 ...

- **Complement Bits**
  - left pointer
  - right pointer
  - next in collision chain
  - array of fanout pointers

- **Hash Value**
  - 8456
  - 6423
  - 0456
  - 1345
  - 7463
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)

---

**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)
  }

- **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))))
  }
Cofactor operation

Algorithm \texttt{POSITIVE\_COFACTOR}(Edge p, Edge v) {
  if (IS\_VAR(p)) {
    if (p == v) {
      if (IS\_INVERTED(v) == IS\_INVERTED(p)) return \texttt{const1}
      else return \texttt{const0}
    } else return p
  }
  if ((c = GET\_COFACTOR(p,v)) == \texttt{NULL}) {
    left = \texttt{POSITIVE\_COFACTOR}(p->left, v)
    right = \texttt{POSITIVE\_COFACTOR}(p->right, v)
    c = \texttt{AND}(left, right)
    \texttt{SET\_COFACTOR}(p,v,c)
  }
  if (IS\_INVERTED(p)) return NOT(c)
  else return c
}
**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

![Diagram of BDD](image)

**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
      \[ x_1 < x_2 < x_3 < ... < x_n \]
  - 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

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
    - \( (\text{node}(f_v) = \text{node}(g_v)) \land (\text{node}(f_v) = \text{node}(g_v)) \Rightarrow \text{node}(f) = \text{node}(g) \)
    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: \( f = \text{ite}(v, g, h) = vg + v'h \), where \( g = f_v \) and \( h = f'_{\overline{v}} \), meaning \( \text{if } v \text{ then } g \text{ else } h \).

\[
\begin{array}{c}
f \\
\text{mux} \\
1 \rightarrow 0 \rightarrow \text{g}
\end{array}
\begin{array}{c}
v \leftarrow 0 \rightarrow 1 \rightarrow \text{h}
\end{array}
\]

\( (v \text{ is top variable of } f) \)

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

<table>
<thead>
<tr>
<th>Table</th>
<th>Subset</th>
<th>Expression</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>( \text{AND}(f, g) )</td>
<td>( fg )</td>
<td>( \text{ite}(f, g, 0) )</td>
</tr>
<tr>
<td>0010</td>
<td>( f &gt; g )</td>
<td>( fg' )</td>
<td>( \text{ite}(f, g', 0) )</td>
</tr>
<tr>
<td>0011</td>
<td>( f )</td>
<td>( f )</td>
<td>( f )</td>
</tr>
<tr>
<td>0100</td>
<td>( f &lt; g )</td>
<td>( f'g )</td>
<td>( \text{ite}(f, 0, g) )</td>
</tr>
<tr>
<td>0101</td>
<td>( g )</td>
<td>( g )</td>
<td>( g )</td>
</tr>
<tr>
<td>0110</td>
<td>( \text{XOR}(f, g) )</td>
<td>( f \oplus g )</td>
<td>( \text{ite}(f, g', g) )</td>
</tr>
<tr>
<td>0111</td>
<td>( \text{OR}(f, g) )</td>
<td>( f + g )</td>
<td>( \text{ite}(f, 1, g) )</td>
</tr>
<tr>
<td>1000</td>
<td>( \text{NOR}(f, g) )</td>
<td>( (f + g)' )</td>
<td>( \text{ite}(f, 0, g') )</td>
</tr>
<tr>
<td>1001</td>
<td>( \text{XNOR}(f, g) )</td>
<td>( f \oplus g' )</td>
<td>( \text{ite}(f, g, g') )</td>
</tr>
<tr>
<td>1010</td>
<td>( \text{NOT}(g) )</td>
<td>( g' )</td>
<td>( \text{ite}(g, 0, 1) )</td>
</tr>
<tr>
<td>1011</td>
<td>( f \geq g )</td>
<td>( f + g' )</td>
<td>( \text{ite}(f, 1, g') )</td>
</tr>
<tr>
<td>1100</td>
<td>( \text{NOR}(f) )</td>
<td>( f' )</td>
<td>( \text{ite}(f, 0, 1) )</td>
</tr>
<tr>
<td>1101</td>
<td>( f \leq g )</td>
<td>( f' + g )</td>
<td>( \text{ite}(f, g, 1) )</td>
</tr>
<tr>
<td>1110</td>
<td>( \text{NAND}(f, g) )</td>
<td>( (f g)' )</td>
<td>( \text{ite}(f, g', 1) )</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean Function Representation
BDD – ITE Operator

- Recursive operation of ITE

\[ \text{Ite}(f,g,h) = f \ g + f' \ h = v \ (f \ g + f' \ h)_v + v' \ (f \ g + f' \ h)_v' \]
\[ = v \ (f_v \ g_v + f'_v \ h_v) + v' \ (f'_v \ g'_v + f''_v \ h'_v) \]
\[ = \text{ite}(v, \text{ite}(f_v,g_v,h_v), \text{ite}(f'_v,g'_v,h'_v)) \]

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

---

Boolean Function Representation
BDD – ITE Operator

- Recursive computation of ITE

Algorithm \( \text{ITE}(f, g, h) \)

\[
\begin{align*}
&\text{if } (f == 1) \ \text{return } g \\
&\text{if } (f == 0) \ \text{return } h \\
&\text{if } (g == h) \ \text{return } g \\
&\text{v} = \text{TOP_VARIABLE}(f, g, h) \quad // \text{top variable from } f, g, h \\
&\text{fn} = \text{ITE}(f_v, g_v, h_v) \quad // \text{recursive calls} \\
&\text{gn} = \text{ITE}(f'_v, g'_v, h'_v) \\
&\text{if } (\text{fn} == \text{gn}) \ \text{return } \text{gn} \quad // \text{reduction} \\
&\text{if } (! (p = \text{HASH_LOOKUP_UNIQUE_TABLE}(v, fn, gn)) \{ \\
&\quad p = \text{CREATE_NODE}(v, fn, gn) \quad // \text{and insert into UNIQUE_TABLE} \\
&\} \\
&\text{INSERT_COMPUTED_TABLE}(p, \text{HASH_KEY}(f, g, h)) \\
&\text{return } p
\end{align*}
\]
Boolean Function Representation
BDD – ITE Operator

Example

I = ite(F, G, H)
   = ite(a, ite(F_a, G_a, H_a), ite(F_a, G_a, H_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_a, 0_a, H_a)))
   = ite(a, C, ite(b, ite(1, 0, 1), ite(0, 0, D)))
   = ite(a, C, ite(b, 0, D))
   = ite(a, C, J)

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

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(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; }
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(\(F, v, G\)) : \(F(v, x) \rightarrow F(G(x), x)\), 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)
}

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

Computed table:
- avoids re-computation of existing results
Before a node \( \text{ite}(v, g, h) \) 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 \( f = \text{ite}(v, g, h) \) exists iff \( \text{ite}(v, g, h) \) is in the unique-table. There is only one pointer for \( \text{ite}(v, g, h) \) and that is the address to the unique-table entry.

Unique-table allows single multi-rooted DAG to represent all users' functions.

---

**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

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

![Diagram of two different DAGs and one DAG using complement pointer](image_url)
Boolean Function Representation
BDD – Implementation Issues

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

  ```
  \begin{align*}
  \text{ite}(F, F, G) & \Rightarrow \text{ite}(F, 1, G) \\
  \text{ite}(F, G, F) & \Rightarrow \text{ite}(F, G, 0) \\
  \text{ite}(F, G, \neg F) & \Rightarrow \text{ite}(F, G, 1) \\
  \text{ite}(F, \neg F, G) & \Rightarrow \text{ite}(F, 0, G)
  \end{align*}
  ```

  - 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:
  
  ```
  \begin{align*}
  \text{ite}(F, F, G) & \Rightarrow \text{ite}(F, 1, G) \\
  \text{ite}(F, G, F) & \Rightarrow \text{ite}(F, G, 0) \\
  \text{ite}(F, G, \neg F) & \Rightarrow \text{ite}(F, G, 1) \\
  \text{ite}(F, \neg F, G) & \Rightarrow \text{ite}(F, 0, G)
  \end{align*}
  ```

  To resolve equivalences:
  
  ```
  \begin{align*}
  \text{ite}(F, 1, G) & \Rightarrow \text{ite}(G, 1, F) \\
  \text{ite}(F, 0, G) & \Rightarrow \text{ite}(\neg G, 1, \neg F) \\
  \text{ite}(F, G, 0) & \Rightarrow \text{ite}(G, F, 0) \\
  \text{ite}(F, G, 1) & \Rightarrow \text{ite}(\neg G, \neg F, 1) \\
  \text{ite}(F, G, \neg G) & \Rightarrow \text{ite}(G, F, \neg F)
  \end{align*}
  ```

  To maximize matches on computed table:
  
  1. First argument is chosen with smallest top variable.
  2. Break ties with smallest address pointer.  \textit{(breaks PORTABILITY!)}

  Triples:
  
  ```
  \begin{align*}
  \text{ite}(F, G, H) & \Rightarrow \text{ite}(\neg F, H, G) = \neg \text{ite}(F, \neg G, \neg H) = \neg \text{ite}(\neg F, \neg H, \neg G)
  \end{align*}
  ```

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

Variable ordering – dynamic
- Changes the order in the middle of BDD applications
  - Must keep same global order
- Problem: External pointers reference internal nodes!
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.

- 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

- 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 than 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

- **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.

![Diagram of BDD and ZBDD examples](image-url)
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 $x_1, x_2, x_3$

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)
    - decision 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)

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 $\varphi = $ 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

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
}
Example

SAT(NOT(9))??

1st case for 9:

conflict!
- undo all assignments
- backtrack

2nd case for 9:

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

1st case for 5:

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

```markdown

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Implication (cont’d)

- Table lookup implementation (27 cases):
  - Implication cases:

```markdown

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

- Conflict cases:

```markdown

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

- Split case:

```markdown

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
</tr>
</tbody>
</table>
```
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

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
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)) \Rightarrow (p1=0))
    }
    if((p' = HASH_LOOKUP(NOT(p1),p2)) {
        LEARN(((p=0) & (p'=0)) \Rightarrow (p2=0))
    }
}

Example (cont’d)

2nd case for 9 (original):

Queue  Assignments

2nd case for 9 (with static learning):

Solution cube: 1 = x, 2 = x, 3 = 0
Boolean Reasoning
SAT – AIG-based Decision Procedure

- **Learning – static**
  - Other learning based on contra-positive:
    - if \( P \Rightarrow Q \), then \( \neg Q \Rightarrow \neg P \)
  - \texttt{foreach} vertex \( v \) {
    \texttt{mark} = \texttt{ASSIGNMENT_MARK}()
    \texttt{IMPLY}(v)
    \texttt{LEARN_IMPLICATIONS}(v)
    \texttt{UNDO_ASSIGNMENTS}(mark)
    \texttt{IMPLY}(\neg(v))
    \texttt{LEARN_IMPLICATIONS}(\neg(v))
    \texttt{UNDO_ASSIGNMENTS}(mark)
  }
  - Problem: learned implications are far too many
    - solution: restrict learning to non-trivial implications and filter redundant implications

- **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 \( i-1 \)
  - \( ((x = 1) \Rightarrow (y = 1)) \land ((x = 0) \Rightarrow (y = 1)) \Rightarrow (y = 1) \)
  - 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
Algorithm \textsc{RECURSIVE\_LEARN}(\text{int level}) \{ 
\textbf{if}(v = \textsc{PICK\_SPLITTING\_VERTEX}()) \{ 
\quad \text{mark} = \textsc{ASSIGNMENT\_MARK}()
\quad \text{IMPLY}(v)
\quad \text{IMPL}1 = \textsc{RECURSIVE\_LEARN}(\text{level+1})
\quad \text{UNDO\_ASSIGNMENTS}(\text{mark})
\quad \text{IMPLY}(\text{NOT}(v))
\quad \text{IMPL}0 = \textsc{RECURSIVE\_LEARN}(\text{level+1})
\quad \text{UNDO\_ASSIGNMENTS}(\text{mark})
\quad \text{return} \text{IMPL}1 \cap \text{IMPL}0
\}
\textbf{else} \{ \quad // \text{completely justified}
\quad \text{return} \textsc{IMPLICATIONS}
\}
\}

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

![Diagram](image1.jpg)

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 \(I(G',E)\), \(G' \subseteq G\) are the gates of \(C\) with assigned values \(v_g\) ≠ unknown, \(E \subseteq G' \times G'\) 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 \(g_j\).

![Diagram](image2.jpg)
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.

- 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.

\[ C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_{n-1} \Rightarrow C_n \quad (C_1: \text{decision vertices}) \]

- 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!

\[ (C_i \Rightarrow \text{Conflict}) \Rightarrow (\neg \text{Conflict} \Rightarrow \neg C_i) \]

- 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
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)

Decision Tree:

- Decision levels: 5
- The decision tree branches at each level, with decision vertices indicated by circles and decision levels by numbers.

- Decision levels that cause a conflict:
  - {2,4}
  - {2,0}
  - {2,3}
  - {4,3}
  - {4,0}
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) \(\Leftrightarrow\) 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: \( y = x_1 \oplus x_2 \oplus x_3 \oplus \ldots \oplus x_n \) (parity function)
      - Circuit size is linear in the number of variables
  - 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

- Generated chess-board Karnaugh map
- CNF (or DNF) formula has \(2^{n-1}\) terms (exponential in the # vars)
Boolean Reasoning
SAT – CNF-based Decision Procedure

- Circuit to CNF conversion
  - Example
    - Single gate
      \[ \overline{a} \lor \overline{b} \lor c \land a \land \overline{c} \land b \lor \overline{c} \]
    - Connected gates
      \[ \overline{1} \lor 2 \lor 4 \land 1 \lor \overline{4} \land \overline{2} \lor \overline{4} \]
      \[ \overline{2} \lor 3 \lor 5 \land 2 \lor \overline{5} \land \overline{3} \lor \overline{5} \]
      \[ 2 \lor \overline{3} \lor 6 \land \overline{4} \lor \overline{5} \lor 7 \land 4 \lor \overline{7} \land \overline{5} \lor \overline{7} \]
      \[ 4 \lor 6 \lor 8 \land \overline{5} \lor \overline{8} \land \overline{6} \lor \overline{8} \]
      \[ 7 \lor 8 \lor 9 \land \overline{7} \lor \overline{9} \land \overline{8} \lor \overline{9} \]
  - Justify to “0”

- 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. \((a + b + \neg c)\)
3. \((\neg a + b + \neg c)\)
4. \((a + c + d)\)
5. \((\neg a + c + d)\)
6. \((\neg a + c + \neg d)\)
7. \((\neg b + \neg c + \neg d)\)
8. \((\neg b + \neg c + d)\)

Source: Kareem 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+\neg b+c)
  - \((a=0) \land (b=1) \Rightarrow (c=1)\)
Boolean Reasoning
SAT – CNF-based Decision Procedure

- **Implication**
  - **Example**

```
\[ \neg a + \neg b + c \cdot (a + \neg c) \cdot (b + \neg c) \]
```

Non-implication cases:

- All clauses satisfied

```
1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0
```

- Not all clauses satisfied (avoid exploring this part)

```
1 \quad x \quad x \quad x \quad x \quad x \quad 0
```

Implication cases:

```
\text{(Implied by the conjunction of clauses)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{(Implied by the conjunction of clauses)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```

```
\text{Implication cases: (cont'd)}
```
Boolean Reasoning
SAT — CNF-based Decision Procedure

□ DPLL (w/ implication)

1. \((a + b + c)\)
2. \((a + b + \neg c)\)
3. \((\neg a + b + \neg c)\)
4. \((a + c + d)\)
5. \((\neg a + c + d)\)
6. \((\neg a + c + \neg d)\)
7. \((\neg b + \neg c + \neg d)\)
8. \((\neg b + \neg c + d)\)

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 1 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

![Decision level](image)

- DPLL (conflict-based learning)

```plaintext
1  | (a + b + c) | 9  | (¬b + ¬c)
2  | (a + b + ¬c) | 10 | (¬a + ¬b)
3  | (¬a + b + ¬c) | 11 |
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
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

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 \((a \Rightarrow b) \land (b \Rightarrow a)\), then \((a = b)\)
  - very powerful for formal equivalence checking

- Incremental SAT solving
  - solving similar CNF formulas in a row
  - share learned clauses