Reduced Ordered Binary Decision Diagrams and And-Inverter Graphs

SUPRATIK CHAKRABORTY
Dept. of Computer Sc & Engg.
Indian Institute of Technology Bombay
Reduced Ordered Binary Decision Diagrams
Binary Decision Diagrams (BDDs)

Graphical representation [Lee, Akers, Bryant]
- Efficient representation & manipulation of Boolean functions in many practical cases
- Enables efficient verification/analysis of a large class of designs
- Worst-case behavior still exponential

Example: \( f = (x_1 \land x_2) \lor \neg x_3 \)
- Represent as binary tree
- Evaluating \( f \):
  - Start from root
  - For each vertex labeled \( x_i \)
    - take dotted branch if \( x_i = 0 \)
    - else take solid branch
Binary Decision Diagrams (BDDs)

» Underlying principle: Shannon decomposition
  + \( f(x_1, x_2, x_3) = x_1 \land f(1, x_2, x_3) \lor \neg x_1 \land f(0, x_2, x_3) \)
  = \( x_1 \land (x_2 \lor \neg x_3) \lor \neg x_1 \land (\neg x_3) \)
  + Can be applied recursively to \( f(1, x_2, x_3) \) and \( f(0, x_2, x_3) \)
    - Gives tree
  + Extend to \( n \) arguments

» Number of nodes can be exponential in number of variables

\[
f = (x_1 \land x_2) \lor \neg x_3
\]
Restrictions on BDDs

» Ordering of variables
  > In all paths from root to leaf, variable labels of nodes must appear in a specified order

» Reduced graphs
  > No two distinct vertices must represent the same function
  > Each non-leaf vertex must have distinct children

Not a ROBDD!

REDUCED ORDERED BDD (ROBDD): Directed Acyclic Graph
ROBDDs

- **Properties**
  - Unique (canonical) representation of $f$ for given ordering of variables
    - Checking $f_1 = f_2$ reduces to checking if ROBDDs are isomorphic
  - Shared subgraphs: size reduction
  - Every path doesn’t have all labels $x_1, x_2, x_3$
  - Every non-leaf vertex has a path to 0 and 1

So far good!
ROBDDs, if-then-else, Multiplexors

» The “ite” operator
  > İte (x, y, z) = (x ∧ y) ∨ (¬x ∧ z), informally “if (x) then (y) else (z)”
  > Can express any binary Boolean operation using “ite”
    + x ∧ y = İte (x, y, 0); ¬x = İte (x, 0, 1); x ∨ y = İte (x, 1, y);

» ROBDDs represent nested “ite” applications

f = İte(x₁, İte(x₂, 1, İte(x₃, 0, 1)), İte(x₃, 0, 1))

» ROBDDs represent multiplexor circuits
Operations on ROBDDs

» View ROBDDs as representing “ite” operators
  > top(f): topmost variable in ROBDD for f
  > \( f_v \): co-factor of f with respect to \( v \) (f with variable v set to 1)
  > \( f_{\neg v} \): co-factor of f with respect to \( \neg v \) (f with variable v set to 0)
  > \( f = \text{ite}(v, f_v, f_{\neg v}) \), where v = top(f)

» Negation
  > \( \text{NOT}(f) = \text{ite}(v, \text{NOT}(f_v), \text{NOT}(f_{\neg v})) \)
  > Recursive formulation; termination: \( \text{NOT}(0) = 1, \text{NOT}(1) = 0 \);
  > Simply swap 0-leaf and 1-leaf

» Binary Boolean operator \( \text{OP} (\wedge, \vee, \oplus, \rightarrow, \ldots) \)
  > Same ordering of variables in ROBDDs for f and g
  > \( v \): variable lowest in order among top(f) and top(g)
  > \( f \ \text{OP} \ g = \text{ite}(v, f_v \ \text{OP} \ g_v, f_{\neg v} \ \text{OP} \ g_{\neg v}) \)
  > Recursive formulation; termination: \( g \ \text{OP} \ h \), where \( g \in \{0, 1, h, \neg h\} \), \( h \in \{0, 1, g, \neg g\} \)
  > \( f_v = f_{\neg v} = f \) if f doesn’t depend on v
Operations on ROBDDs

» `ite(f, g, h)`
  > Same ordering of variables in ROBDDs for f, g, h
  > `v = topmost variable among top(f), top(g), top(h)`
  > `ite(f, g, h) = ite(v, ite(f_v, g_v, h_v), ite(f_{¬v}, g_{¬v}, h_{¬v}))`
  > Recursive formulation; termination: `ite(1, g, h) = g; ite(0, g, h) = h; ite(f, g, g) = g`

» `compose(f, v, h)`
  > Same ordering of variables in ROBDDs for f, h
  > Replace variable v in f by function h
  > Let `u = top(f)`
  > `compose(f, v, h) = ite(u, compose(f_u, v, h_u), compose(f_{¬u}, v, h_{¬u}))`
  > Recursive formulation; termination: if `v < u`, `compose(f, v, h) = f;`
  > `if (v == u), compose(f, v, h) = ite(h, f_u, f_{¬u})`

» Is f satisfiable? Is f a tautology? Is f a contradiction?
  > Does ROBDD for f have a 1-leaf? Is ROBDD for f a single 1-leaf? A single 0-leaf?
Counting Satisfying Assignments

ROBDDs can be used to count satisfying assignments of \( f \)

- \( \text{count}(0) = 0; \ \text{count}(1) = 1 \)
- For all nodes \( g \) in bottom-up order
  - Let \( v = \text{top}(g) \)
  - Let \( a = \# \text{ variables in variable order between } v \) and \( \text{top}(g_{\neg v}) \)
  - Let \( b = \# \text{ variables in variable order between } v \) and \( \text{top}(g_v) \)
  - \( \text{count}(g) = \text{count}(g_{\neg v}) \times 2^a + \text{count}(g_v) \times 2^b \)
- \( \text{count}(f) = \# \text{ satisfying assignments of } f \)

- Polynomial in ROBDD size

- Counting satisfying assignments of CNF/DNF formulae
  - \#P-complete
  - For every variable order, ROBDD must be large for some CNF/DNF formulas
Example of ROBDD Operation

\[ f \lor g \]

\[ \text{ite}(x_1, \ f_1 \lor g_1, \ f_0 \lor g_0) \]

\[ \text{ite}(x_2, f_1 \lor g_{11}, f_1 \lor 1) \]

\[ \begin{align*}
\text{ite}(x_3, 1 \lor g_{11}, f_{10} \lor g_{11}) &= \text{ite}(x_3, 1, f_{10}) \\
&= f_1
\end{align*} \]
### ROBDDs: Complexity of Operations

<table>
<thead>
<tr>
<th>Operation ((G_i: \text{ROBDD for } f_i))</th>
<th>Time &amp; Size of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>▶️ reduce ((G))</td>
<td>(O(</td>
</tr>
<tr>
<td>▶️ complement((f))</td>
<td>(O(1)) time, (O(</td>
</tr>
<tr>
<td>▶️ apply ((\text{op, } f_1, f_2))</td>
<td>(O(</td>
</tr>
<tr>
<td>▶️ ite((f_1, f_2, f_3))</td>
<td>(O(</td>
</tr>
<tr>
<td>▶️ compose ((f_1, v, f_2))</td>
<td>(O(</td>
</tr>
<tr>
<td>▶️ satisfy-one((f))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>▶️ model-count((f))</td>
<td>(O(</td>
</tr>
<tr>
<td>▶️ restrict ((f, x_i, 1)) or restrict ((f, x_i, 0))</td>
<td>(O(</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key Takeaways

» Complexity polynomial in sizes of argument ROBDDs
  > If sizes can be kept under control, we are in business!
  > ROBDD size limiting factor in most applications

» If arguments to an operation are ROBDDs, result is also an ROBDD.

» Quantification:
  > ∃ x1. f(x1, x2, x3) = f(0, x2, x3) ∨ f(1, x2, x3)
  > ∀ x1. f(x1, x2, x3) = f(0, x2, x3) ∧ f(1, x2, x3)

Useful in model checking if next-state functions and characteristic functions of sets of states can be represented succinctly.
Variable Ordering Problem

\[ f = (x_1 \land x_2) \lor (x_3 \land x_4) \lor (x_5 \land x_6) \]

Order: \( x_1 < x_3 < x_5 < x_2 < x_4 < x_6 \)

Order: \( x_1 < x_2 < x_3 < x_4 < x_5 < x_6 \)
Variable Ordering Problem

ROBDD size

- Extremely sensitive to variable ordering
  
  + $f = (x_1 \land x_2) \lor (x_3 \land x_4) \lor \ldots \lor (x_{2n-1} \land x_{2n})$
    
    - $2n+2$ vertices for order $x_1 < x_2 < x_3 < x_4 < \ldots < x_{2n-1} < x_{2n}$
    
    - $2^{n+1}$ vertices for order $x_1 < x_{n+1} < x_2 < x_{n+2} < \ldots < x_n < x_{2n}$
  
  + $f = x_1 \land x_2 \land x_3 \land \ldots \land x_n$
    
    - $n+2$ vertices for all orderings

- Exponential regardless of variable ordering
  
  - Consider bits $i$ and $2n-i$ of product of two $n$-bit integers
  
  - For every variable order, one of the above BDDs exponential

- Determining best variable order for arbitrary functions is computationally intractable

Heuristics: Static ordering, Dynamic ordering
Variable Ordering Solutions

Static ordering

- Common heuristics based on “structure” (graph representation) of function
- Other heuristics: simulated annealing, genetic algorithms, machine learning ...
- Rules of thumb
  + Control variables closer to root
  + “Related” variables close in order

\[ f = (x_1 \land x_2) \lor (x_3 \land x_4) \lor (x_5 \land x_6) \]
  + \{x_1, x_2\}, \{x_3, x_4\} and \{x_5, x_6\}: sets of “related” variables
  + \(x_1 < x_2 < x_3 < x_4 < x_5 < x_6\) or \(x_5 < x_6 < x_1 < x_2 < x_3 < x_4\) gives smaller ROBDD than \(x_1 < x_3 < x_5 < x_2 < x_4 < x_6\)

\[ g = (a > b), \text{ where } a \text{ and } b \text{ are 32-bit unsigned integers} \]
  + \(a_{31} < b_{31} < b_{30} < b_{30} < \ldots a_1 < b_1 < a_0 < b_0\) gives smaller ROBDD than
    \(a_{31} < a_{30} < \ldots a_1 < a_0 < b_{31} < b_{30} < \ldots b_1 < b_0\) or
    \(a_0 < b_0 < a_1 < b_1 < \ldots a_{30} < b_{30} < a_{31} < b_{31}\)
Variable Ordering Solutions

**Dynamic ordering**

- Starts with user-provided static order
- If dynamic re-ordering triggered on-the-fly, evaluate benefits of re-ordering small subset of variables
  - If beneficial, re-order and repeat until no benefit
- Expensive in general, sophisticated triggers essential
- Key observation [Friedman]: Given ROBDD with $x_1 < ... x_i < x_{i+1} < ... x_n$,
  - Permuting $x_1 ... x_i$ has no effect on ROBDD nodes labeled by $x_{i+1} ... x_n$
  - Permuting $x_{i+1} ... x_n$ has no effect on ROBDD nodes labeled by $x_1 ... x_i$
  - Variables in adjacent levels easily swappable

---

![Diagram showing variable ordering and re-ordering](image)

Re-order $a$ & $b$
Variable Ordering Solutions

» Dynamic ordering

  > Sifting individual variables
    + For each ROBDD variable,
      - Swap with adjacent layer until it reaches root
      - Swap with adjacent layer until it reaches bottom
      - Reorder variable to position with minimum ROBDD size
      - Heuristic stopping conditions
    + Greedy approach may miss good orderings

  > Sifting groups of “related” variables
    + Keep “related” variables close in order when sifting with respect to others
    + Sift “related” variables within group
    + Can obtain good orders missed by sifting individual variables

  > All sifting heuristics are expensive; triggers must be carefully designed
Implementing ROBDD Packages

» Shared ROBDDs
  > Multiple functions represented simultaneously as a multi-rooted DAG.
  > Each root and descendants form an ROBDD
  > Different roots can share subgraphs
  > Variable ordering same for all functions represented

» Unique Table: a hash table
  > Every ROBDD node characterized by \((v, f_1, f_0)\)
    + \(v\) : variable; \(f_1, f_0\): ROBDD node pointers to 1-child and 0-child
  > Hash table: key = \((v, f_1, f_0)\); value = pointer to ROBDD node for \((v, f_1, f_0)\)
  > When creating ROBDD node for \((v, f_1, f_0)\), first check in unique table
    + Create new node only if not in unique table; create and insert in unique table
    + Otherwise, re-use ROBDD node from unique table
  > Structural hashing

» Computed Table: a software cache
  > Hash table without collision chains
  > Remember results of recent ROBDD operations for later re-use
  > Key: (BinaryOp, \(f_1, f_2\)) or (TernaryOp, \(f_1, f_2, f_3\)); Value: ROBDD for result
  > Insert every time a new result is computed
Implementing ROBDD Packages

» Complement edges
  > Represent by bubbles
  > If a node is reached by a complement edge, take complement of function represented by the node
  > Complementation: O(1) time and space

f = \neg ((x_1 \land \neg x_2 \land x_3) \lor (\neg x_1 \land \neg x_3))

» Garbage collection
  > Keep track of references (pointers) to each ROBDD node
  > If RefCount(node) becomes 0, move to “death row”
    + Do not remove from Unique Table or Computed Table
    + #nodes > threshold \Rightarrow
      free all nodes in “death row” and remove from Unique & Computed Tables
    + Reference to node on “death row” (before it is freed) removes it from “death row”
Some BDD Variants

- **Multi-Terminal BDDs (MTBDDs)**
  - Leaf nodes can be any integer in 0 to k
  - Used in representing & manipulating multi-valued logic systems

- **Edge-Valued BDDs (EVBDDs)**
  - Used for multi-valued logic systems, representing linear expressions etc.

- **Binary Moment Diagrams (BMDs)**
  - Proven useful for reducing size of representation of integer multipliers
  - -- multiplier verification

- **Zero-Suppressed BDDs (ZBDDs)**
  - No 1-edge ever points to the 0 leaf
  - Canonical for a given variable order [Minato]
  - Useful for compactly representing sparse sets of elements
Some BDD Variants

» Partitioned ROBDDs

> Partition input space into disjoint “windows” \{w_1, \ldots, w_n\}
> Variable order \pi_i associated with window \textit{w}_i
  + May differ across windows
> Given Boolean function \textit{f},
  + For each window \textit{w}_i, let \textit{f}_i = \textit{w}_i \land \textit{f}
  + Represent \textit{f} as \{(\textit{w}_1, \textit{f}_1), \ldots, (\textit{w}_n, \textit{f}_n)\}
  + Each (\textit{w}_i, \textit{f}_i) represented as pair of ROBDDs using \pi_i
    - Using different \pi_i for different \textit{w}_i, significant reduction in representation size possible
  + \textit{f} = \textit{f}_1 \land \ldots \land \textit{f}_n
> Canonical for a given choice of \{w_1, \ldots, w_n\} and \{\pi_1, \ldots, \pi_n\} [Narayan]
BDD Packages Out There

» CUDD package (Colorado University)
» CMU BDD package
» TiGeR (commercial package)
» CAL (University of California, Berkeley)
» EHV
» ...
And-Inverter Graphs
And-Inverter Graphs (AIGs)

- AND and NOT functionally complete for Boolean algebra
- Associativity of AND \( \Rightarrow \) 2-input AND gate suffices
- Represent a Boolean function as a directed acyclic graph
  - Nodes are 2-input AND gates: indegree = 2 (always!), outdegree arbitrary
  - Bubbled edges represent logical negation

\[
(a \land b) \lor (d \land ((a \land \neg c) \lor (b \land c)))
\]
Properties of AIGs

- **Non-canonical**
  - Structurally different AIGs for same function

- **Size:** # AND gates

- **Depth:** longest path from root to leaf

- **Efficient construction from circuits**
  - Linear in size of circuits built of AND, NOT, OR, NAND, NOR, XOR, XNOR, ... gates
  - Replace each gate by AIG equivalent; remove paired bubbles on edges
    - AIG equivalent of common Boolean gates: size & depth linear in # inputs

- Can be exponentially more succinct than CNF/DNF
  - n-input XOR gate
Properties of AIGs

» Can be exponentially more succinct than BDDs
  > n-bit integer multiplier
  > Consider bits \( i \) and \( 2n-i \) of product
    + For every BDD variable ordering, one of the above BDDs exponential in \( n \)
    + There exists a circuit, and hence AIG, for both bits that is linear in \( n \)

» Tautology checking requires checking satisfiability (SAT solving) of output

» Linear time conversion to CNF for SAT solving
  > Tseitin encoding: coming soon!

» Uniform structure of AIGs (2-input AND and NOT) useful in circuit-based SAT solving
Operations on AIGs

**Negation:**
- Add a bubbled edge
- Similar to complemented edges in ROBDD packages

**Binary/ternary Boolean operations**
- NOT, AND, OR, XOR, NAND, NOR, XNOR, ITE, ...

```plaintext
compose(f, v, h) = compose(\neg g, v, h) = \neg compose(g, v, h)
compose(g_1 \land g_2, v, h) = compose(g_1, v, h) \land compose(g_2, v, h)
Recursive formulation; termination: compose(v, v, h) = h; compose(u, v, h) = u
```
Operations on AIGs

» Is f satisfiable? Is f a tautology? Is f a contradiction?
  > Requires SAT solving: Is SAT(f) ? Is SAT(¬f)?

» Complexity of operations (preserve AIGs of arguments)

<table>
<thead>
<tr>
<th>Operation (G_i: AIG for f_i)</th>
<th>Time</th>
<th>Size of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>complement(f)</td>
<td>O(1)</td>
<td>O(</td>
</tr>
<tr>
<td>f_1 OP f_2</td>
<td>O(1)</td>
<td>O(</td>
</tr>
<tr>
<td>+ Binary Boolean operations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ite(f_1, f_2, f_3)</td>
<td>O(1)</td>
<td>O(</td>
</tr>
<tr>
<td>compose(f_1, v, f_2)</td>
<td>O(</td>
<td>G_1</td>
</tr>
<tr>
<td>satisfy-one(f)</td>
<td>NP-complete</td>
<td>O(n)</td>
</tr>
<tr>
<td>model-count(f)</td>
<td>#P-complete</td>
<td>O(n)</td>
</tr>
<tr>
<td>restrict(f, v, 0) or restrict(f, v, 1)</td>
<td>O(</td>
<td>G</td>
</tr>
</tbody>
</table>

AIGs + Powerful Modern SAT solvers:
Scalable approach to large industrial-scale problems
Example of AIG Operation

» $\text{compose}(f, v, h)$

- $\neg \text{compose}(\neg g_1, v, h)$
- $\neg \text{compose}(g_1, v, h)$
- $\neg (\text{compose}(u, v, h) \land \text{compose}(\neg v, v, h))$
- $\neg (u \land \neg \text{compose}(v, v, h))$
- $\neg (u \land \neg h)$
- $\neg (u \land \neg h)$
- $\neg (u \land \neg h)$
- $\neg (u \land \neg h)$
- $\neg (u \land \neg h) \land \neg (h \land \neg w)$
Simplifying AIGs: Rewrite Rules

Sampling of simple rewrite rules:

AIG based tools have tens of such rewrite rules

Peephole optimization restricted to AIGs of $\approx 4$ inputs

Empirically, very effective in reducing AIG sizes
Some locally size-reducing rewrite rules can be globally size-increasing in the presence of shared AIG nodes.

((a ∨ b) ∧ (b ∨ c)) ↔ ((a ∧ c) ∨ b)

Locally size-reducing

Globally size-increasing

Rewrite rules must be carefully designed & applied
Simplifying AIGs: Structural Hashing

- Ideally, AIG should not have two nodes representing the same function

\[ r = p \land q; \quad s = p \land q \]  

- **Structural hashing (strash) when constructing AIG**
  - No two nodes must have both inputs same
    - Maintain hash table with inputs \((i_1, i_2)\) as key and pointer to node representing \((i_1 \land i_2)\) as value
  - No nodes with constant inputs or single input
    - Use rewrites
Simplifying AIGs: strash

Using strash

- Before creating a new node with inputs \((i_1, i_2)\), check if hash table has entry with key \((i_1, i_2)\) or \((i_2, i_1)\)
  
  + Alternatively, impose ordering on nodes, check for \((\min(i_1, i_2), \max(i_1, i_2))\)

- If yes, use value from hash table

- Else, create new node with inputs \((i_1, i_2)\) and make a new entry in hash table
Beyond strash & rewrite

Even after using strash & rewrite, AIG may contain nodes representing the same or complemented function.

\[ p = a \oplus b; \quad q = \neg (a \oplus b) \]

Desirable to merge \( p \) and \( q \), and use bubbles on outgoing edges to denote complementation.

strash does not help here!

Solution: Check if \((p \leftrightarrow q)\) or \((p \leftrightarrow \neg q)\) is a tautology (SAT solving!)

If yes, merge nodes & use bubbles appropriately on outgoing edges.
Simplifying AIGs: fraig

Functionally Reduced AIG (FRAIG)

- No two nodes represent the same or complemented function
  - Similar idea used for ROBDDs as well!
- For every pair \((u, v)\) of nodes (including constant node 1)
  - If \((u \leftrightarrow v)\) is a tautology, merge nodes \(u\) and \(v\) in AIG
  - If \((u \leftrightarrow \neg v)\) is a tautology, merge nodes \(u\) and \(v\) in AIG, and use bubbles appropriately on outgoing edges
  - Use simplification rules and structural hashing all throughout
Simplifying AIGs: fraig, strash, rewrite
Fraig-ing ≠ Canonicalization

- **Canonical** \( \Rightarrow \) only 1 representation for a function

- **Two FRAIGs** for \( a \land b \land c \land d \)
  - Individually, fraig-reduced & strash-reduced
  - Structurally very different

- **Semi-canonical**
  - Within the same AIG manager, only one representation for a function after fraig-ing

- **fraig-ing can be expensive**
  - AIG with \( n \) nodes \( \Rightarrow n(n-1)/2 \) SAT solver calls (naïve approach!)
  - Solution:
    - Restrict pairs of nodes for which SAT solver calls needed
    - fraig infrequently – design sophisticated trigger conditions
Practical fraig-ing

» Efficiently partition nodes into classes
  > Put all nodes (including inputs, internal nodes, constants) in one partition
  > Repeat sufficiently many times
    + Pick random 0/1 input vector and evaluate all nodes
    + Refine each partition: nodes evaluating to 1, nodes evaluating to 0
  > Use input vectors that distinguish all pairs of primary inputs
  > Nodes in same class potentially functionally equivalent
  > Nodes in different classes certainly not functionally equivalent

» Check functional equivalence of node pairs in same class
  > Node pair (a, b): Is \((a \land \neg b) \lor (b \land \neg a)\) satisfiable?  \[
  \text{SAT solver calls}
  \]
  > If UNSAT,  \(a \leftrightarrow b\): Merge \(a\) and \(b\)
  > If SAT, use satisfying assignment to refine partition containing \(a\) and \(b\)
  > Prune pairs by strash-ing
    + \((a \leftrightarrow b)\) and \((c \leftrightarrow d)\) \(\Rightarrow\) \((a \land c) \leftrightarrow (b \land d)\)
Other Operations on AIGs

» AIGs useful for synthesis, verification, technology mapping, ... of circuits

» Some other common operations (package-dependent)

  > balance
    + Minimize depth of AIG
    + Useful for optimizing delays

  > refactor
    + Introduce cut in AIG (several cost-functions can be used)
    + Express function of AIG node in terms of cut: cut-function
    + Factor cut-function, if possible, & accept change if #nodes in AIG reduces

  > collapse
    + Recursively compose functions of fanin nodes into fanout nodes, building global functions using BDD/SOP/POS
    + AIG to BDD/SOP/POS conversion
    + Doesn’t scale to large circuits
AIG to CNF

» Most modern SAT solvers accept input formulae in conjunctive normal form (CNF)
  > Literals: variables & their complements, e.g. \( a, b, \neg c \)
  > Clauses: disjunction of literals, e.g. \( (a \lor b \lor \neg c) \)
  > Cubes: conjunction of literals, e.g. \( (a \land b \land \neg c) \)
  > CNF formula: conjunction of clauses, product-of-sums
    e.g. \( (a \lor b \lor \neg c) \land (\neg a \lor \neg b \lor d) \)
  > DNF formula: disjunction of cubes, sum-of-products
    e.g. \( (a \land c \land \neg d) \lor (\neg b \land d \land c) \)

» Given AIG for \( f \), generating \( f \) in CNF can blow up badly
  > Start from DNF formula \( f \) (size: \( |f| \))
  > Build AIG for \( f \) (size: \( O(|f|) \))
  > Convert AIG to CNF
    + Effectively convert DNF to CNF: known worst-case exponential blow-up
AIG to CNF

Solution: Given AIG for f, generate equisatisfiable g in CNF

Equisatisfiability

- f and g equisatisfiable iff both f and g are satisfiable or both are unsatisfiable
- (a ∨ b) equisatisfiable with (c ∧ d), not equisatisfiable with d ∧ (¬ d ∨ c) ∧ ¬ c
- Checking satisfiability of g tells us if f is satisfiable

Tseitin encoding

- Given a Boolean circuit (AIG is a special case) for f, generate equisatisfiable g
  + g is in CNF
  + Every satisfying assignment for f gives unique satisfying assignment for g and vice-versa
  + Size of g linear in size of circuit (AIG) for f
- Widely used in SAT solving context
**AIG to CNF: Tseitin Encoding**

- **Given Boolean circuit (or AIG) for f**
  - Associate new variable with output of each gate
  - For each Boolean gate, generate formula expressing output in terms of inputs
  - Convert each formula generated above to CNF
    - Boolean gate with fixed # inputs $\Rightarrow$ CNF formula for gate is of fixed size
  - Conjoin CNF formula for each gate and output variable of circuit

$$
\begin{align*}
t_1 & \leftrightarrow (\neg a \land \neg b) \\
t_2 & \leftrightarrow (a \land b) \\
t_3 & \leftrightarrow (\neg t_1 \land \neg t_2) \\
f & \leftrightarrow (t_3 \land c)
\end{align*}
$$

$$
\begin{align*}
(\neg t_1 \lor \neg a) & \land (\neg t_1 \lor \neg b) \land (a \lor b \lor t_1) \\
(\neg t_2 \lor a) & \land (\neg t_2 \lor b) \land (\neg a \lor \neg b \lor t_2) \\
(\neg t_3 \lor \neg t_1) & \land (\neg t_3 \lor \neg t_2) \land (t_1 \lor t_2 \lor t_3) \\
(\neg f \lor t_3) & \land (\neg f \lor c) \land (\neg t_3 \lor \neg c \lor f) \land f
\end{align*}
$$
AIG To CNF: Tseitin Encoding

**Bijection between SAT assignments of f and g**

For every SAT assignment of $f$

> Evaluate all gate outputs for given assignment of inputs

> $a = 1, b = 0, c = 1$ gives $a = 1, b = 0, c = 1, t_1 = 0, t_2 = 0, t_3 = 1, f = 1$

For every SAT assignment of $g$

> Project assignment on variables of $f$

> $a = 1, b = 0, c = 1, t_1 = 0, t_2 = 0, t_3 = 1, f = 1$ gives $a = 1, b = 0, c = 1$

$t_1 \leftrightarrow (\neg a \land \neg b) \\
(\neg t_1 \lor \neg a) \land (\neg t_1 \lor \neg b) \land \\
(a \lor b \lor t_1) \\
\land$

$t_2 \leftrightarrow (a \land b) \\
(\neg t_2 \lor a) \land (\neg t_2 \lor b) \land \\
(\neg a \lor \neg b \lor t_2) \\
\land$

$t_3 \leftrightarrow (\neg t_1 \land \neg t_2) \\
(\neg t_3 \lor \neg t_1) \land (\neg t_3 \lor \neg t_2) \land \\
(t_1 \lor t_2 \lor t_3) \\
\land$

$f \leftrightarrow (t_3 \land c) \\
(\neg f \lor t_3) \land (\neg f \lor c) \land \\
(\neg t_3 \lor \neg c \lor f) \land f \\
\land f$
AIG Packages & Tools Out There

» Several tools support AIG-based inputs, outputs, reasoning
  > Synthesis, Optimization, SAT solving, Model Checking, Verification

» AIGER format
  > Compact binary and ASCII formats for representing AIGs
  > Large suite of tools to convert to and from other circuit representations
    + mv_blif, smv, equations, cnf, …
  > Allows specification of latches/flip-flops as well

» abc (University of California, Berkeley)
  > Perhaps the most widely used AIG package
  > Synthesis, optimization, technology mapping, verification tool-suite integrated with AIG package
ROBDDs, AIGs & Variants: Applications

» Extensively used in CAD applications for digital hardware

» Some interesting applications
  > Combinational logic equivalence checking
    + Is a combinational circuit functionally equivalent to another?
  > Sequential machine equivalence checking
    + Using combinational equivalence of next-state logic
    + Representing transition relations and state spaces in symbolic methods
  > Symbolic and bounded model checking
    + Representing sets of states symbolically using characteristic functions
  > Test pattern generation
    + Automatic Test Pattern Generation (ATPG) essentially tries to come up with satisfying instances of a Boolean formula
ROBDDs, AIGs & Variants: Applications

- Timing verification
  - For representing false paths in a circuit succinctly
  - For representing discretized time encoded as binary values
- Representing sets using characteristic functions
- Symbolic simulation
  - Assign variables and/or constants to circuit inputs and determine output values in terms of variables
  - Representing sets of constant values
- Logic synthesis and optimization

>> Other domains: Combinatorics, manipulating classes of combined Boolean algebraic expressions ...
Example Application

» Combinational equivalence checking

Given two combinational designs

> Same number of inputs and outputs
> Determine if each output of Design 1 is functionally equivalent to corresponding output of Design 2
> Design 1 could be a set of logic equations
> Design 2 could be a gate level/transistor level circuit
Example Application

- ROBDD for every function is a canonical representation
- Construct ROBDDs and check if corresponding graphs are isomorphic
  - ROBDD isomorphism is a simple problem
- Construct AIGs, and check for functional equivalence of corresponding AIG nodes (fraig, strash, rewrite)
- Miter-based approach:

Design 1

Design 2

F

Designs functionally equivalent iff F is identical to 0 (unsatisfiable)
Example Application

» Equivalence checking reduces to checking for unsatisfiability of miter output
  > ROBDD-based solution:
    + Build ROBDD for miter output
    + If ROBDD has a non-leaf vertex or has a 1 leaf, F is satisfiable
  > AIG-based solution
    + Build AIG for miter output using strash, fraig, rewrite
    + Check if CNF formula equisatisfiable to output F is satisfiable

» Lots of smarter techniques, heuristics exist in literature
  > Build on top of basic idea explained here

» Worst case complexity necessarily bad
  + Unsatisfiability: co-NP complete
Example Application

- **Sequential equivalence checking**
  - Restricted case: Reduces to combinational equivalence

- **Given sequential machines M1 and M2 with correspondence between state variables**
  - Equivalence checking of M1 and M2 reduces to combinational equivalence checking of next-state and output logic

![Diagram](image_url)

Comb Logic1

FF

Comb Logic2

FF

Given Equivalence
Equivalence Checkers

» Commercial equivalence checkers in the market
  > Abstract, Avant!, Cadence, Synopsys, Verplex ...

» Several advanced public-domain tools
  > abc, NuSMV, ...

» For best results, knowledge about structure crucial
  > Divide and conquer
  > Learning techniques useful for determining implication
  > State of the art tools claim to infer information about circuit structure automatically
    + Potentially pattern match for known subcircuits -- Wallace Tree multipliers, Manchester Carry Adders