**Boolean Function Representation**

**Binary Decision Diagram (BDD)**

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

**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
  - \( x_1 < x_2 < x_3 < \ldots < x_n \)
  - **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

\[ f = ab + a'c + bc'd \]

\[ \begin{align*}
  f &= ab + a'c + bc'd \\
  f &= ab + a'c + bc'd
\end{align*} \]
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
    - $(\text{node}(f_1) = \text{node}(g_1)) \land (\text{node}(f_2) = \text{node}(g_2)) \Rightarrow \text{node}(f) = \text{node}(g)$
    - and provides recursive argument that node(f) is unique when using the unique hash-table

Boolean Function Representation
BDD – Onset Counting

$F = b' + a'c' = ab' + a'c'b + a'c'$ (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 measured 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 $h = f_{\neg v}$, meaning “if $v$ then $g$ else $h$”

ITE Operator

- 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>Subset</th>
<th>Expression</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>AND(f, g)</td>
<td>f g</td>
</tr>
<tr>
<td>0010</td>
<td>f &gt; g</td>
<td>f' g</td>
</tr>
<tr>
<td>0011</td>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>0100</td>
<td>f &lt; g</td>
<td>f' g</td>
</tr>
<tr>
<td>0101</td>
<td>g</td>
<td>g</td>
</tr>
<tr>
<td>0110</td>
<td>XOR(f, g)</td>
<td>f \oplus g</td>
</tr>
<tr>
<td>0111</td>
<td>OR(f, g)</td>
<td>f + g</td>
</tr>
<tr>
<td>1000</td>
<td>NOR(f, g)</td>
<td>f \oplus g'</td>
</tr>
<tr>
<td>1001</td>
<td>XNOR(f, g)</td>
<td>f \oplus g'</td>
</tr>
<tr>
<td>1010</td>
<td>NOT(g)</td>
<td>f'</td>
</tr>
<tr>
<td>1011</td>
<td>f \leq g</td>
<td>f' + g</td>
</tr>
<tr>
<td>1100</td>
<td>NOT(f)</td>
<td>f</td>
</tr>
<tr>
<td>1101</td>
<td>f \geq g</td>
<td>f' + g</td>
</tr>
<tr>
<td>1110</td>
<td>NAND(f, g)</td>
<td>(f \oplus g)'</td>
</tr>
<tr>
<td>1111</td>
<td>f = g</td>
<td>f'</td>
</tr>
</tbody>
</table>

(v is top variable of f)
Recursive operation of ITE

\[ \text{Ite}(f, g, h) = f \cdot g + f' \cdot h \]
\[ = v \cdot (f \cdot g + f' \cdot h) + v' \cdot (f' \cdot g + f \cdot h) \]
\[ = \text{ite}(v, \text{ite}(f' \cdot g, f \cdot h), \text{ite}(f \cdot g, f' \cdot h)) \]

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

Example

\[ \text{Ite}(F, G, H) = \text{ite}(a, \text{ite}(\text{ite}(1, 0, 1), \text{ite}(0, 0, D)), \text{ite}(a, 0, D)) \]

Check:

\[ F = a + b \]
\[ G = ac \]
\[ H = b + d \]
\[ \text{Ite}(F, G, H) = (a + b)(ac) + a'b(b + d) = ac + a'b'd \]
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 \(\text{TOP VARIABLE}(F) > v\) return \(F\) // \(F\) does not depend on \(v\)
    if \(\text{TOP VARIABLE}(F) == v\) return ITE\((G, F_1, F_0)\)
    \(i = \text{COMPOSE}(F_1, v, G)\)
    \(e = \text{COMPOSE}(F_0, v, G)\)
    return ITE\((\text{TOP VARIABLE}(F), i, e)\)
}

Note:
1. \(F_1\) and \(F_0\) are the 1-child and 0-child of \(F\), respectively
2. \(G, i, e\) are not functions of \(v\)
3. If \(\text{TOP VARIABLE}\) of \(F\) is \(v\), then ITE\((G, F_1, F_0)\) 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 if \(\text{ite}(v, g, h)\) 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
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
  - To maintain strong canonical form, need to resolve 4 equivalences:
    - ITE(F, G, H) = ITE(F, G, H)
    - ITE(F, 0, G) = ITE(H, G, 0)
    - ITE(F, G, 0) = ITE(F, G, H)
    - ITE(F, G, 1) = ITE(H, G, H)

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

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

- Standard triples:
  - ITE(F, G, H) = ITE(F, 1, G)
  - ITE(F, G, F) = ITE(F, G, 0)
  - ITE(F, G, ¬F) = ITE(F, G, 1)
  - ITE(F, ¬F, G) = ITE(F, 0, G)

  - To resolve equivalences:
    - ITE(F, 1, G) = ITE(G, 1, F)
    - ITE(F, 0, G) = ITE(¬G, 1, ¬F)
    - ITE(F, G, 0) = ITE(G, F, 0)
    - ITE(F, G, 1) = ITE(¬G, ¬F, 1)
    - ITE(F, ¬G, ¬F) = ITE(G, F, ¬F)

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

- Triples:
  - ITE(F, G, H) = ITE(¬F, H, G) = ¬ITE(F, G, H)
  - Choose the one such that the first and second argument of ITE should not be complemented 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

- Variable ordering – dynamic
  - Changes the order in the middle of BDD applications
    - must keep same global order
  - Problem: External pointers reference internal nodes!
  - 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.
**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 $x_1, x_2, x_3$

**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**

**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

**Reading:**

*Logic Synthesis in a Nutshell*

Section 2

most of the following slides are by courtesy of Andreas Kuehlmann
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 \( \phi \) = checking Tautology on formula \( \neg \phi \), 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

SAT – AIG-based Decision Procedure

- General Davis-Putnam procedure
  Algorithm\textsc{SAT}(Edge p) \{
  queue = \textsc{INIT\_QUEUE}(p)
  if(!\textsc{IMPLY}(p)) return FALSE
  return \textsc{JUSTIFY}(queue)
\}

Algorithm\textsc{JUSTIFY}(queue) \{
  if(\textsc{QUEUE\_EMPTY}(queue)) return TRUE
  mark = \textsc{ASSIGNMENT\_MARK}()
  v = \textsc{QUEUE\_NEXT}(queue) // decision vertex
  if(\textsc{IMPLY}(\neg(v))) \{
    if(\textsc{JUSTIFY}(queue)) return TRUE
    \} // conflict
  UNDO\_ASSIGNMENTS(mark)
  if(\textsc{IMPLY}(v)) \{
    if(\textsc{JUSTIFY}(queue)) return TRUE
    \} // conflict
  UNDO\_ASSIGNMENTS(mark)
  return FALSE
\}
Example

SAT(NOT(9))??

1st case for 9:

Queue Assignments

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:
  - Conflict cases:
  - Split case:
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 avois 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

```
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))
  }
}
```

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

Learning – static
- Other learning based on contra-positive:
  if \((P \implies Q)\), then \((\neg Q \implies \neg P)\)

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

- 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 \(i-1\)

\[((x=1) \implies (y=1)) \land (x=0) \implies (y=1)\] \implies (y=1)

- 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
    \[(\text{level0-value}, \text{level1-choice1}, \text{level1-choice2})\] \[\{(0,1,x), (0,1,x), (0,1,x)\}\]
  - automatically set learned values for level0 if both level1 choices agree, e.g.,

![Diagram](attachment:image.png)

**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
  \[I(G,E), G' \subseteq G\] the gates of C with assigned values \(v_g \neq \text{unknown}, E \subseteq G' \times G'\) 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](attachment:image.png)

- 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
      \[C_1 \Rightarrow C_2 \Rightarrow \ldots \Rightarrow C_{n-1} \Rightarrow C_n\] (\(C_i\): 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!

- 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)

**Decision Tree:**

- Decision levels: 5
- Decision Tree:

**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)

- 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
        - 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
DPLL procedure

Algorithm DPLL() {
    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**

- **Implication**
  - Example
    - $a \land c \iff (\neg a \land \neg b \land \neg c) \land (a \land \neg c) \land (b \land \neg c)$

  - Non-implication cases:
    - [Diagram showing all clauses satisfied]
    - [Diagram showing not all clauses satisfied (avoid exploring this part)]

- **DPLL (w/ implication)**
  - Steps:
    1. $a + b + c$
    2. $a + b + \neg c$
    3. $(\neg a + b + \neg c)$
    4. $(a + c + \neg c)$
    5. $(\neg a + c + \neg c)$
    6. $(\neg a + c + \neg c)$
    7. $(a + b + c + \neg c)$
    8. $(\neg a + c + \neg c)$

Source: Karen A. Sakallah, Univ. of Michigan

**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 I 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**

![UIP on level 5](image)

- **DPLL (conflict-based learning)**
  1. $(a + b + c)$
  2. $(a + b + ¬c)$
  3. $(¬a + b + ¬c)$
  4. $(a + c + d)$
  5. $(¬a + c + a)$
  6. $(¬a + c + ¬a)$
  7. $(¬b + c + ¬c)$
  8. $(¬b + ¬c + d)$

Source: Kenem 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

- **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