DLL Algorithm Pseudo-code

```
DLL_iterative()
{
    Preprocess
    status = preprocess();
    if (status != UNKNOWN)
        return status;
    while (1)
    {
        Branch
        decide_next_branch();
        while (true)
        {
            Propagate implications of that branch and deal with conflicts
            status = deduce();
            if (status == CONFLICT)
            {
                blevel = analyze_conflict();
                if (blevel < 0)
                    return UNSATIFIABLE;
                else
                    backtrack(blevel);
            }
            else if (status == SATISFIABLE)
                return SATISFIABLE;
            else break;
        }
    }
}
```
DLL Algorithm Pseudo-code

```python
def DLL_iterative()
    status = preprocess();
    if (status != UNKNOWN)
        return status;
    while (1):
        decide_next_branch();
        while (true)
            status = deduce();
            if (status == CONFLICT)
                blevel = analyze_conflict();
                if (blevel < 0)
                    return UNSATISFIABLE;
                else
                    backtrack(blevel);
            else if (status == SATISFIABLE)
                return SATISFIABLE;
            else break;
```

Main Steps:

- **Pre-processing**
- **Branching**
- **Unit propagation** (apply unit rule)
- **Conflict Analysis** & **Backtracking**

Comparison:

Naïve 2-counters/clause vs 2-literal watching

- When a literal is set to 1, update counters for all clauses it appears in
- Same when literal is set to 0
- If a literal is set, need to update each clause the variable appears in
- During backtrack, must update counters

- No need for update
- Update watched literal
- If a literal is set to 0, need to update only each clause it is watched in
- No updates needed during backtrack! (why?)

Overall effect: Fewer clauses accesses in 2-lit
zChaff Relative Cache Performance

<table>
<thead>
<tr>
<th></th>
<th>1dlx_c_mc_ex_bp_f</th>
<th>Hanoe4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Num Access</td>
<td>Miss Rate</td>
</tr>
<tr>
<td>Z-Chaff</td>
<td>L1 24,029,356</td>
<td>4.75%</td>
</tr>
<tr>
<td></td>
<td>L2 1,659,877</td>
<td>4.63%</td>
</tr>
<tr>
<td>SATO</td>
<td>L1 188,352,975</td>
<td>36.76%</td>
</tr>
<tr>
<td>(-g100)</td>
<td>L2 79,422,894</td>
<td>9.74%</td>
</tr>
<tr>
<td>Grasp</td>
<td>L1 415,572,501</td>
<td>32.89%</td>
</tr>
<tr>
<td></td>
<td>L2 153,490,555</td>
<td>50.25%</td>
</tr>
</tbody>
</table>

The programs are compiled with -O3 using gcc 2.8.1 (for GRASP and Chaff) or gcc 2.8.1 (for Sato3.2.1) on Sun OS 4.1.2 Trace was generated with QPT quick tracing and profiling tool. Trace was processed with dineroIV, the memory configuration is similar to a Pentium III processor: L1: 16K Data, 16K Instruction, L2: 256k Unified. Both have 32 byte cache line, 4 way set associativity.

Key Ideas in Modern DLL SAT Solving

- **Data structures**: Implication graph
- **Conflict Analysis**: Learn (using cuts in implication graph) and use non-chronological backtracking
- **Decision heuristic**: must be dynamic, low overhead, quick to conflict/solution
- **Unit propagation (BCP)**: 2-literal watching helps keep memory accesses down

- **Principle**: Keep #(memory accesses)/step low
  - A step $\rightarrow$ a primitive operation for SAT solving, such as a branch
Other Techniques

• Random Restarts
  – Periodically throw away current decision stack and start from the beginning
    • Why will this change the search on restart?
  – Used in most modern SAT solvers

• Clause deletion
  – Conflict clauses take up memory
    • What’s the worst-case blow-up?
  – Delete periodically based on some heuristic ("age", length, etc.)

Proof Generation

• If the SAT solver returns "satisfiable", we can check that solution by evaluating the circuit
• If it returns "unsatisfiable", what then?
Proof

• Starting from facts (clauses), the SAT solver has presumably derived “unsatisfiable” (the empty clause)
• So there must be a way of going step-by-step from input clauses to the empty clause using rules
  – In fact, there’s only one rule: resolution

Resolution as a Cut in Implication Graph

\[ (V_2 + V_3' + V_5' + V_6') \]

\[ (V_3' + V_6 + V_4) \]
\[ (V_6 + V_5' + V_1') \]
\[ (V_2 + V_4' + V_6 + V_1) \]
\[ (V_2 + V_4' + V_6 + V_5') \]
\[ (V_2 + V_3' + V_5' + V_6) \]
Resolution Graph

- Nodes are clauses
- Edges are applications of resolution

Proof Checker

- Given resolution graph, how to check it?
- Traverse it, checking that each node is correctly obtained from its predecessor nodes using resolution
  - This generates proof
Incremental SAT Solving

- Suppose you have not just one SAT problem to solver, but many “slightly differing” problems over the same variables
- Can we re-use the search over many problems?
  - i.e. perform only “incremental” work
Operations Needed

1. Adding clauses
2. Deleting clauses

• Which is easy and which is hard?
  – If previous problem is unsat, how does an operation change it?
  – If previous is sat?
Deleting Clauses

Engineering Issues

- Too expensive to traverse graph
- Instead, group original clauses into groups
- Each derived clause belongs to all groups that it is resolved from
  - Implement with bit-vector
Binary Decision Diagrams

Boolean Function Representations

- Syntactic: e.g.: CNF, DNF, Circuit
- Semantic: e.g.: Truth table, Binary Decision Tree, BDD
Reduced Ordered BDDs

- Invented by Randal E. Bryant in mid-80s
  - IEEE Transactions on Computers 1986 paper is one of the most highly cited papers in EECS
- Useful data structure to represent Boolean functions
  - Applications in synthesis, verification, program analysis, …
- Commonly known simply as BDDs
- Many variants of BDDs have proved useful in other tasks
- Links to coding theory (trellises), etc.

Cofactors

- A Boolean function $F$ of $n$ variables $x_1, x_2, \ldots, x_n$
  - $F : \{0,1\}^n \rightarrow \{0,1\}$
- Suppose we define new Boolean functions of $n$-1 variables as follows:
  - $F_{x_1}(x_2, \ldots, x_n) = F(1, x_2, x_3, \ldots, x_n)$
  - $F_{x_1'}(x_2, \ldots, x_n) = F(0, x_2, x_3, \ldots, x_n)$
- $F_{x_1}$ and $F_{x_1'}$ are cofactors of $F$. 
Shannon Expansion

- $F(x_1, \ldots, x_n) = x_i \cdot F_{x_i} + x_i' \cdot F_{x_i'}$

- Proof?

Shannon expansion with many variables

- $F(x, y, z, w) = xy F_{xy} + x'y F_{xy} + xy' F_{xy'} + x'y' F_{xy'}$
Properties of Cofactors

- Suppose you construct a new function \( H \) from two existing functions \( F \) and \( G \): e.g.,
  - \( H = F' \)
  - \( H = F \cdot G \)
  - \( H = F + G \)
  - Etc.

- What is the relation between cofactors of \( H \) and those of \( F \) and \( G \)?

Very Useful Property

- Cofactor of NOT is NOT of cofactors
- Cofactor of AND is AND of cofactors
- …

- Works for any binary operator
BDDs from Truth Tables

Truth Table

Binary Decision Tree

Binary Decision Diagram (BDD)

Ordered Binary Decision Diagram (OBDD)

Reduced Ordered Binary Decision Diagram (ROBDD, simply called BDD)

Example: Odd Parity Function

Binary Decision Tree
Nodes & Edges

• How is $F$ related to $\alpha$, $F_1$, $F_2$?

Ordering

Impose arbitrary order: $a < b < c < d$

Why didn’t this part change?
Reduction

- Identify Redundancies

- 3 Rules:
  1. Merge equivalent leaves
  2. Merge isomorphic nodes
  3. Eliminate redundant tests

Merge Equivalent Leaves

```
  a  a  ...  a  \rightarrow  a
```

"a" is either 0 or 1
Merge Isomorphic Nodes

Eliminate Redundant Tests
Final ROBDD for Odd Parity Function

Example of Rule 3
What can BDDs be used for?

- Uniquely representing a Boolean function
  - And a Boolean function can represent sets

- Satisfiability solving!

(RO)BDDs are canonical

- Theorem (R. Bryant): If $G$, $G'$ are ROBDD’s of a Boolean function $f$ with $k$ inputs, using same variable ordering, then $G$ and $G'$ are identical.
Sensitivity to Ordering

- Given a function with n inputs, one input ordering may require exponential \# vertices in ROBDD, while other may be linear in size.

- Example: \( f = x_1 x_2 + x_3 x_4 + x_5 x_6 \)
  
\[ x_1 < x_2 < x_3 < x_4 < x_5 < x_6 \]

- \( x_1 < x_4 < x_5 < x_2 < x_3 < x_6 \)

Applying an Operator to BDDs

- Two options:
  1. Construct an operator for each logic operator: AND, OR, NOT, EXOR, ...
  2. Build a few core operators and define everything else in terms of those

Advantage of 2:
- Less programming work
- Easier to add new operators later by writing “wrappers”
Core Operators

• Just two of them!
1. Restrict(Function F, variable v, constant k)
   • Shannon cofactor of F w.r.t. v=k

2. ITE(Function I, Function T, Function E)
   • “if-then-else” operator

ITE

• Just like:
  – “if then else” in a programming language
  – A mux in hardware
• ITE(I(x), T(x), E(x))
  – If I(x) then T(x) else E(x)
The ITE Function

- $\text{ITE}( I(x), T(x), E(x) )$
- $= I(x) \cdot T(x) + I'(x) \cdot E(x)$

What good is the ITE?

- How do we express NOT?
- OR?
- AND?
How do we implement ITE?

- Divide and conquer!
- Use Shannon cofactoring...
- Recall: Operator of cofactors is Cofactor of operators...

ITE Algorithm

ITE (bdd I, bdd T, bdd E) {
    if (terminal case) { return computed result; }
    else { // general case
        Let x be the topmost variable of I, T, E;
        PosFactor = ITE(I_x, T_x, E_x);  
        NegFactor = ITE(I'_x, T'_x, E'_x);
        R = new node labeled by x;
        R.low = NegFactor;
        R.high = PosFactor;
        Reduce(R);
        return R;
    }
}
Terminal Cases

- \( \text{ITE}(1, \ T, \ E) = \)
- \( \text{ITE}(0, \ T, \ E) = \)
- \( \text{ITE}(I, \ T, \ T) = \)
- \( \text{ITE}(I, 1, 0) = \)
- \( \ldots \)

General Case

- Still need to do cofactor (Restrict)
- How hard is that?
  - Which variable are we cofactoring out? (2 cases)
Practical Issues

• Previous calls to ITE are cached
  – “memoization”

• Every BDD node created goes into a “unique table”
  – Before creating a new node R, look up this table
  – Avoids need for reduction

Sharing: Multi-Rooted DAG

• BDD for 4-bit adder
• Each output bit (of the sum & carry) is a distinct rooted BDD
• But they share sub-DAGs
Wrap-up

• What you know: SAT Solving, BDD Basics
• Finish BDDs, actually get to model checking!