DPLL algorithm for SAT

[Davis, Putnam, Logemann, Loveland 1960,62]

Given: CNF formula $f(v_1, v_2, \ldots, v_k)$, and an ordering function $\text{Next\_Variable}$

Example: 

$$(a + b)(a + c)(a + b)$$

$C_1$ $C_2$ $C_3$

CONFLICT!

SAT!
Basic Backtracking Search

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. $(\neg b + \neg c + d)$
Basic Search with Implications

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. \((\neg b + \neg c + d)\)
DPLL algorithm: Unit clause rule

**Rule:** Assign to **true** any single literal clauses.

\[(a + b + c) \quad \text{||} \quad 0 \quad \text{||} \quad 0\]

Apply Iteratively: **Boolean Constraint Propagation (BCP)**

\[a(\bar{a} + c)(\bar{b} + c)(a + b + \bar{c})(\bar{c} + e)(\bar{d} + e)(\bar{c} + d + \bar{e})\]

\[c(\bar{b} + c)(\bar{c} + e)(\bar{d} + e)(\bar{c} + d + \bar{e})\]

\[e(\bar{d} + e)\]
Pure Literal Rule

• A variable is *pure* if its literals are either all positive or all negative

• Satisfiability of a formula is unaffected by assigning pure variables the values that satisfy all the clauses containing them

\[
\varphi = (a + c)(b + c)(b + \neg d)(\neg a + \neg b + d)
\]

• Set \( c \) to 1; if \( \varphi \) becomes unsatisfiable, then it is also unsatisfiable when \( c \) is set to 0.
Resolution (original DP)

- Iteratively apply resolution (consensus) to eliminate one variable each time
  - i.e., consensus between all pairs of clauses containing \( x \) and \( \neg x \)
  - formula satisfiability is preserved
- Stop applying resolution when,
  - Either empty clause is derived \( \Rightarrow \) instance is unsatisfiable
  - Or only clauses satisfied or with pure literals are obtained \( \Rightarrow \) instance is satisfiable

\[
\phi = (a + c)(b + c)(d + c)(\neg a + \neg b + \neg c)
\]

Eliminate variable \( c \)

\[
\phi_1 = (a + \neg a + \neg b)(b + \neg a + \neg b)(d + \neg a + \neg b)
= (d + \neg a + \neg b)
\]

Instance is SAT!
Stallmarck’s Method (SM) in CNF

• Recursive application of the branch-merge rule to each variable with the goal of identifying common conclusions

\[ \varphi = (a + b)(\neg a + c)(\neg b + d)(\neg c + d) \]

Try \( a = 0 \):

\[ (a = 0) \Rightarrow (b = 1) \Rightarrow (d = 1) \]

\( C(a = 0) = \{ a = 0, b = 1, d = 1 \} \)

Try \( a = 1 \):

\[ (a = 1) \Rightarrow (c = 1) \Rightarrow (d = 1) \]

\( C(a = 1) = \{ a = 1, c = 1, d = 1 \} \)

\[ C(a = 0) \cap C(a = 1) = \{ d = 1 \} \]

Any assignment to variable \( a \) implies \( d = 1 \).

Hence, \( d = 1 \) is a necessary assignment!

Recursion can be of arbitrary depth

Jan 24, 2012

EE-709@IITB
Recursive Learning (RL) in CNF

- Recursive evaluation of clause satisfiability requirements for identifying common assignments

\[ \varphi = (a + b)(\neg a + d) (\neg b + d) \]

Try \( a = 1 \):

\[ (a = 1) \implies (d = 1) \]

\[ C(a = 1) = \{a = 1, d = 1\} \]

Try \( b = 1 \):

\[ (b = 1) \implies (d = 1) \]

\[ C(b = 1) = \{b = 1, d = 1\} \]

\[ C(a = 1) \cap C(b = 1) = \{d = 1\} \]

Every way of satisfying \((a + b)\) implies \(d = 1\).

Hence, \(d = 1\) is a necessary assignment.

Recursion can be of arbitrary depth
SM vs. RL

- Both complete procedures for SAT
- Stallmarck’s method:
  - hypothetic reasoning based on variables
- Recursive learning:
  - hypothetic reasoning based on clauses
- Both can be integrated into backtrack search algorithms
Local Search

• Repeat $M$ times:
  – Randomly pick complete assignment
  – Repeat $K$ times (and while exist unsatisfied clauses):
    • Flip variable that will satisfy largest number of unsat clauses

\[ \varphi = (a + b)(\neg a + c)(\neg b + d)(\neg c + d) \]

Pick random assignment

\[ \varphi = (a + b)(\neg a + c)(\neg b + d)(\neg c + d) \]

Flip assignment on \(d\)

\[ \varphi = (a + b)(\neg a + c)(\neg b + d)(\neg c + d) \]

Instance is satisfied!
Comparison

- Local search is **incomplete**
  - If instances are known to be SAT, local search can be competitive
- Resolution is in general **impractical**
- Stallmarck’s Method (SM) and Recursive Learning (RL) are in general **slow**, though **robust**
  - SM and RL can derive too much **unnecessary** information
- For most EDA applications, **backtrack search (DP)** is currently the most promising approach!
  - **Augmented with techniques for inferring new clauses/implicates** (i.e. learning)!
Techniques for Backtrack Search

- Conflict analysis
  - Clause/implicate recording
  - Non-chronological backtracking
- Incorporate and extend ideas from:
  - Resolution
  - Recursive learning
  - Stallmarck’s method
- Formula simplification & Clause inference [Li, AAAI00]
- Randomization & Restarts [Gomes & Selman, AAAI98]
Clause Recording

• During backtrack search, for each conflict create clause that explains and prevents recurrence of same conflict

\[ \varphi = (a + b)(\neg b + c + d)(\neg b + e)(\neg d + \neg e + f) \ldots \]

Assume (decisions) \( c = 0 \) and \( f = 0 \)

Assign \( a = 0 \) and imply assignments

A conflict is reached: \( (\neg d + \neg e + f) \) is unsat

\((a = 0) \land (c = 0) \land (f = 0) \Rightarrow (\varphi = 0)\)

\((\varphi = 1) \Rightarrow (a = 1) \lor (c = 1) \lor (f = 1)\)

\( \therefore \) create new clause: \( (a + c + f) \)
Clause Recording

- Clauses derived from conflicts can also be viewed as the result of applying selective consensus

\[ \varphi = (a + b)(\neg b + c + d)(\neg b + e)(\neg d + \neg e + f) \ldots \]
Non-Chronological Backtracking

- During backtrack search, in the presence of conflicts, backtrack to one of the causes of the conflict

\[ \varphi = (a + b)(\neg b + c + d)(\neg b + e)(\neg d + \neg e + f)(a + c + f)(\neg a + g)(\neg g + b)(\neg h + j)(\neg i + k) \ldots \]

Assume (decisions) \( c = 0, f = 0, h = 0 \) and \( i = 0 \)

Assignment \( a = 0 \) caused conflict \( \Rightarrow \) clause \((a + c + f)\) created

\((a + c + f)\) implies \( a = 1 \)

A conflict is again reached: \((\neg d + \neg e + f)\) is unsat

\((a = 1) \land (c = 0) \land (f = 0) \Rightarrow (\varphi = 0)\)

\((\varphi = 1) \Rightarrow (a = 0) \lor (c = 1) \lor (f = 1)\)

:: create new clause: \((\neg a + c + f)\)
Non-Chronological Backtracking

Created clauses: \((a + c + f)\) and \((\neg a + c + f)\)

Apply consensus:
- new *unsat* clause \((c + f)\)

:. backtrack to most recent decision: \(f = 0\)

:. created clauses/implicates:
- \((a + c + f)\),
- \((\neg a + c + f)\), and
- \((c + f)\)
**Ideas from other Approaches**

- Resolution, Stallmarck’s method and recursive learning can be incorporated into **backtrack search (DP)**
  - create additional clauses/implicates
    - anticipate and prevent conflicting conditions
    - identify necessary assignments
    - allow for non-chronological backtracking

**Resolution within DP:**

\[(a + b + c) (\neg a + b + d)\]

consensus \[(b + c + d)\]

**Unit clause**!(b + c + d) **Unit clause**!

Clause provides **explanation** for necessary assignment \(b = 1\)
Stallmarck’s Method within DP

\[ \varphi = (a + b + e)(\neg a + c + f)(\neg b + d)(\neg c + d + g) \]

Implications:

\((a = 0) \land (e = 0) \Rightarrow (b = 1) \Rightarrow (d = 1)\)

\((a = 1) \land (f = 0) \Rightarrow (c = 1) \Rightarrow (d = 1)\)

\((e = 0) \land (f = 0) \land (g = 0) \Rightarrow (d = 1)\)

Clausal form:

\((e + f + g + d) \quad \text{Unit clause!}\)

Clause provides explanation for necessary assignment \(d = 1\)
Recursive Learning within DP

\[ \varphi = (a + b + c)(\neg a + d + e)(\neg b + d + c) \]

Implications:

\( (a = 1) \land (e = 0) \Rightarrow (d = 1) \)
\( (b = 1) \land (c = 0) \Rightarrow (d = 1) \)
\( (c = 0) \land ((e = 0) \land (c = 0)) \Rightarrow (d = 1) \)

Clausal form:

\[ (c + e + d) \]

Clause provides explanation for necessary assignment \( d = 1 \)
The Power of Consensus

- Most search pruning techniques can be explained as particular ways of applying selective consensus
  - Conflict-based clause recording
  - Non-chronological backtracking
  - Extending Stallmarck’s method to backtrack search
  - Extending recursive learning to backtrack search
  - Clause inference conditions

- General consensus is computationally too expensive!
- Most techniques indirectly identify which consensus operations to apply!
  - To create new clauses/implicates
    - To identify necessary assignments
SAT Solvers Today

- Capacity:
  - Formulas upto a *million variables* and *3-4 million clauses* can be solved in *few hours*
  - Only for *structured instances* e.g. derived from real-world circuits & systems

- Tool offerings:
  - Public domain
    - GRASP: Univ. of Michigan
    - SATO: Univ. of Iowa
    - zChaff: Princeton University
    - BerkMin: Cadence Berkeley Labs.
  - Commercial
    - PROVER: Prover Technologies
Solving circuit problems as SAT

Input Vector Assignment? → Primary Output ‘i’ to 1?
SAT formulas for simple gates

\[(\overline{c} + a)(\overline{c} + b)(c + \overline{a} + \overline{b})\]

\[(c + \overline{a})(c + \overline{b})(\overline{c} + a + b)\]

\[(a + b)(\overline{a} + \overline{b})\]

\[(c + a)(c + b)(\overline{c} + \overline{a} + \overline{b})\]
Solving circuit problems as SAT

- Set of clauses representing function of each gate
- **Unit literal clause asserting output to '1'**

\[(\overline{b} + f)(\overline{c} + f)(b + c + \overline{f})\]
\[(d + g)(e + g)(\overline{d} + \overline{e} + \overline{g})\]
\[(a + \overline{h})(f + \overline{h})(\overline{a} + \overline{f} + \overline{h})\]
\[(h + \overline{i})(g + \overline{i})(\overline{h} + \overline{g} + i)\]
\[(i)\]
Combinational Equivalence Checking (CEC)

- Currently most practical and pervasive equivalence checking technology
- Nearly full automation possible
- Designs of up to several million gates verified in a few hours or minutes
- Hierarchical verification deployed
- Full chip verification possible
- **Key methodology**: Convert sequential equivalence checking to a CEC problem!
  - Match Latches & extract comb. portions for EC
CEC in Today’s ASIC Design Flow

- RTL Design
- Synthesis & optimization
- DFT insertion
- IO Insertion
- Placement
- Clock tree synthesis
- Routing
- ECO
Major Industrial Offerings of CEC

- Formality *(Synopsys)*
- Conformal Suite *(Verplex, now Cadence)*
- FormalPro *(Mentor Graphics)*

**Typical capabilities of these tools:**
- Can handle circuits of up to several million gates flat in up to a few hours of runtime
- Comprehensive **debug tool** to pinpoint error-sources
- **Counter-example display & cross-link** of RTL and gate-level netlists for easier debugging
- Ability to **checkpoint** verification process and restart from same point later
- **What if** capability (unique to FormalPro)
Combinational Equivalence Checking

- Functional Approach
  - transform output functions of combinational circuits into a unique (canonical) representation
  - two circuits are equivalent if their representations are identical
  - efficient canonical representation: BDD

- Structural
  - identify structurally similar internal points
  - prove internal points (cut-points) equivalent
  - find implications
Functional Equivalence

• If BDD can be constructed for each circuit
  ➢ represent each circuit as shared (multi-output) BDD
    ❖ use the same variable ordering!
  ➢ BDDs of both circuits must be identical

• If BDDs are too large
  ➢ cannot construct BDD, memory problem
  ➢ use partitioned BDD method
    • decompose circuit into smaller pieces, each as BDD
    • check equivalence of internal points
Functional Decomposition

- Decompose each function into functional blocks
  - represent each block as a BDD (partitioned BDD method)
  - define cut-points \( (z) \)
  - verify equivalence of blocks at cut-points
  - starting at primary inputs
Cut-Points Resolution Problem

- If *all pairs* of cut-points \((z_1, z_2)\) are equivalent
  - so are the two functions, \(F, G\)
- If *intermediate* functions \((f_2, g_2)\) are not equivalent
  - the functions \((F, G)\) may still be equivalent
  - this is called **false negative**

- **Why do we have false negative?**
  - functions are represented in terms of *intermediate* variables
  - to prove/disprove equivalence must represent the functions in terms of *primary inputs* (BDD composition)
Cut-Point Resolution – Theory

• Let $f_1(x) = g_1(x) \ \forall x$
  - if $f_2(z,y) \equiv g_2(z,y), \ \forall z,y$ then $f_2(f_1(x),y) \equiv g_2(f_1(x),y) \Rightarrow F \equiv G$
  - if $f_2(z,y) \neq g_2(z,y), \ \forall z,y$ then $f_2(f_1(x),y) \neq g_2(f_1(x),y) \not\Rightarrow F \neq G$

We cannot say if $F \equiv G$ or not

• False negative
  - two functions are equivalent, but the verification algorithm declares them as different.
Cut-Point Resolution

- How to verify if negative is *false* or *true*?

- Procedure 1: create a miter (XOR) between two potentially equivalent nodes/functions
  - perform ATPG test for *stuck-at 0*
  - find test pattern to prove \( F \neq G \)
  - efficient for true negative
  - (gives *test vector*, a proof)
  - inefficient when there is no test

\[
\begin{align*}
0, & \ F \equiv G \text{ (false negative)} \\
1, & \ F \neq G \text{ (true negative)}
\end{align*}
\]
Cut-Point Resolution

• Procedure 2: create a BDD for \( F \oplus G \)

- perform satisfiability analysis (SAT) of the BDD
  - if BDD for \( F \oplus G = \emptyset \), problem is not satisfiable, *false* negative
  - BDD for \( F \oplus G \neq \emptyset \), problem is satisfiable, *true* negative

\[
F \oplus G = \begin{cases} 
\emptyset, & F \equiv G \text{ (false negative)} \\
\text{Non-empty,} & F \neq G
\end{cases}
\]

Note: must compose BDDs until they are equivalent, or expressed in terms of primary inputs

- the SAT solution, if exists, provides a *test vector* (proof of non-equivalence) – as in ATPG
- unlike the ATPG technique, it is effective for false negative (the BDD is empty!)
Thank you