Validity vs. Satisfiability

- **Validity:**
  - A sentence is valid if it is true in every interpretation (every interpretation is a model).
  - A sentence \( s \) is a valid consequence of a set \( S \) of sentences if \( (S \implies s) \) is valid.
  - Proof methods: Truth-Tables and Inference Rules

- **Satisfiability:**
  - A set of sentences is satisfiable if there exists an interpretation in which every sentence is true (it has at least one model).
  - Proof Methods: Truth-Tables and The Davis-Putnam-Logeman-Loveland procedure (DPLL).
**SAT: Propositional Satisfiability**

- An instance of SAT is defined as (X, S)
  - X: A set of 0-1 (propositional) variables
  - S: A set of sentences (formulas) on X
- Goal: Find an assignment f: X -> {0, 1} so that every sentence becomes true.
- SAT is the first NP-complete problem.
  - Good News: Thousands of problems can be transformed into SAT
  - Bad News: There are no efficient algorithms for SAT

---

**Truth Table for Satisfiability**

- A propositional formula \( \varphi \) is satisfiable iff one of the values of \( \varphi \) is True.
- Example: \( \varphi = (a \lor c) \land (b \lor c) \land (\neg a \lor \neg b \lor \neg c) \)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a \lor c</th>
<th>b \lor c</th>
<th>\neg a \lor \neg b \lor \neg c</th>
<th>\varphi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

\( \{ a = 0, b = 0, c = 1 \} \) is a model of \( \varphi \)
Simplification of Truth Table

- As long as \( \varphi \) has a value True, we may stop working.
- Several rows may be merged into one with don’t-care values (x)
- Example: \( \varphi = (a \lor c) \land (b \lor c) \land (\neg a \lor \neg b \lor \neg c) \)

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>a \lor c</th>
<th>b \lor c</th>
<th>\neg a \lor \neg b \lor \neg c</th>
<th>\varphi</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>False</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Truth Table as a Binary Tree

- Each internal node has a variable
- Two children represent True and False.
- The leaf nodes have the value of \( \varphi \)
- Each row of Truth Table is a path from the root to a leaf.
- Example: \( \varphi = (a \lor c) \land (b \lor c) \land (\neg a \lor \neg b \lor \neg c) \)
Simplification of Binary Tree

- Don't-care variables can be removed from the tree

Example: $\varphi = (a \lor c) \land (b \lor c) \land \neg a \lor \neg b \lor \neg c$

- Order of variables may affect the size of the tree

Example: $\varphi = (a \lor c) \land (b \lor c) \land \neg a \lor \neg b \lor \neg c$

The Davis-Putnam-Logmann-Loveland method exploit these ideas.
Conjunctive Normal Form (CNF)

\[ \varphi = (a \lor c) \land (b \lor c) \land (\neg a \lor \neg b \lor \neg c) \]

Propositional Clauses

- Every propositional constraint can be converted into a set of equivalent clauses.
  - \( S = \{ C_1, C_2, \ldots, C_m \} = C_1 \land C_2 \land \ldots \land C_m \)
- A clause is a disjunction of literals.
  - \( C = (L_1 \lor L_2 \lor \ldots \lor L_k) \)
- A literal is either a variable or the negation of a variable.
  - \( L = x \) or \( L = \neg x \)
- A set of clauses is also said to be in Conjunctive Normal Form (CNF).
Gate CNF

$\phi_d = [d \equiv \neg(a \& b)]$
$= [d \rightarrow \neg(a \& b)] \& [\neg(a \& b) \rightarrow d]$
$= (\neg d \lor \neg a \lor \neg b)[\neg d \rightarrow (a \& b)]$
$= (a \lor d)(b \lor d)(\neg a \lor \neg b \lor \neg d)$

DIMACS Format

This is an example of an SAT instance in DIMACS format

```
p cnf 3 5
1 2 0
1 3 0
2 0
-1 -2 0
-1 -3 0
-2 -3 0
X1 V X2
X1 V X3
-X1 V -X2
-X1 V -X3
-X2 V -X3
```

DIMACS: Discrete Mathematics and Computer Science
More on Assignments

- Assignments: \( \{a = 0, b = 1\} = \neg a \land b \)
  - Partial (some variables still unassigned)
  - Complete (all variables assigned)
  - Conflicting (imply \( \neg \varphi \))

\[ \varphi = (a \lor c) \land (b \lor c) \land (\neg a \lor \neg b \lor \neg c) \]
\[ \varphi \rightarrow (a \lor c) \]
\[ \neg(a \lor c) \rightarrow \neg \varphi \]
\[ \neg a \land \neg c \rightarrow \neg \varphi \]

Literal & Clause Classification

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

\[ a \land b \text{ assigned and } c \text{ and } d \text{ unassigned} \]
Unit Clause Rule - Implications

- An unresolved clause is unit if it has exactly one unassigned literal
  \[ \varphi = (a \lor c)(b \lor c)(\neg a \lor \neg b \lor \neg c) \]
- A unit clause has exactly one option for being satisfied
  \[ a \land b \Rightarrow \neg c \]
  i.e. c must be set to 0.

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
  \[ \varphi = (a \lor c)(b \lor c)(b \lor \neg d)(\neg a \lor \neg b \lor d) \]
- Set c to 1; if \( \varphi \) becomes unsatisfiable, then it is also unsatisfiable when c is set to 0.
Resolution/Consensus

- General technique for deriving new clauses
  Example: $\omega_1 = (\neg a \lor b \lor c)$, $\omega_2 = (a \lor b \lor d)$
  Resolution:
  $$\text{res}(\omega_1, \omega_2, a) = (b \lor c \lor d)$$

- Complete procedure for satisfiability [Davis, JACM'60]
- Impractical for real-world problem instances
- Application of restricted forms has been successful!
  - E.g., always apply restricted resolution
    - $\text{res}(\neg a \lor \alpha, (a \lor \alpha), a) = (\alpha)$
    - $\alpha$ is a disjunction of literals

A Taxonomy of SAT Algorithms

**Complete**
- Resolution (original DP)
- Recursive learning (RL)
- BDDs (Binary Decision Diagram)
- ... (more complete)

**Incomplete**
- Continuous formulations
- Genetic algorithms
- Simulated annealing
- Tabu search
- ... (more incomplete)

A Taxonomy of SAT Algorithms

Complete
- SAT Algorithms
- Can prove unsatisfiability

Incomplete
- SAT Algorithms
- Cannot prove unsatisfiability

Complete
- Backtrack search (DP)
- Resolution (original DP)
- Recursive learning (RL)
- BDDs (Binary Decision Diagram)
- ... (more complete)

Incomplete
- Local search (hill climbing)
- Continuous formulations
- Genetic algorithms
- Simulated annealing
- Tabu search
- ... (more incomplete)
SAT:
A search problem

Simplification Rules: \(1 \lor C = 1, \ 0 \lor C = C\)

Unit Propagation:
Make all unit clauses true;
No splitting on them.
The Davis-Putnam-Logemann-Loveland Algorithm (1960)

```plaintext
function Satisfiable ( clause set S ) return { 0, 1 }
repeat /* unit propagation */
    for each unit clause L in S do
        delete from S every clause containing L
        delete -L from each C in S in which -L occurs
        if S is empty then return 1
        else if a clause in S is empty then return 0
    until no more new unit clauses changes
    choose a literal L occurring in S /* splitting */
    if Satisfiable( S U { L } ) then return 1
    else if Satisfiable( S U { -L } ) then return 1
    else return 0
```

DPLL uses Depth-First-Search

- DFS with Backtrack: Instead of maintaining a path of nodes, only one node is maintained. The node is modified when going down and everything is undone when going up.
- The branching factor is dictated by the splitting rule.
DPLL uses Backtrack Search

- Implicit enumeration
- Iterated unit-clause rule
  - Boolean constraint propagation
- Pure-literal rule
- Chronological backtracking in presence of conflicts
- The worst-time complexity is exponential in terms of the number of variables.

Implementing The DPLL Algorithm

- A destructive data structure is needed for clauses: Instead of copying clauses, modify them and then undo modification when backtracking.
- Efficient algorithms for unit-propagation.
- There are many choices for selecting a literal to split (heuristics are needed).
The n-queen problem

- Place n queens on an n x n chessboard so that no two queens attack each other.
- Conditions:
  - Each row has a unique queen
  - No two queens on the same column
  - No two queens on the same diagonal
- Use $n^2$ boolean variables: $q_{ij}$ is true iff the queen on row $i$ is in column $j$.

Example: 4-queen problem

- 16 variables: $q_{11}, q_{12}, q_{13}, q_{14}, q_{21}, ..., q_{44}$
- Each row has a unique queen
  1. $q_{11} | q_{12} | q_{13} | q_{14}$
  2. $q_{21} | q_{22} | q_{23} | q_{24}$
  3. $q_{31} | q_{32} | q_{33} | q_{34}$
  4. $q_{41} | q_{42} | q_{43} | q_{44}$
  5. -$q_{11} | -$q_{12}$
    ...
    -$q_{43} | -$q_{44}$
Example: 4-queen problem

- 16 variables: q11, q12, q13, q14, q21, ..., q44

- No two queens on the same column
  1. -q11 | -q21
  2. -q11 | -q31
  3. -q11 | -q41
  ...
  -q34 | -q44

- No two queens on the same diagonal
  1. -q11 | -q22, -q11 | -q33, -q11 | -q44
  2. -q12 | -q21, -q12 | -q23, -q12 | -q34
  ...

Satbox’s Results on n-queen prob.

<table>
<thead>
<tr>
<th>n</th>
<th>solutions</th>
<th>1 soln.</th>
<th>all soln.</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>92</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>10</td>
<td>724</td>
<td>0.00</td>
<td>0.04</td>
</tr>
<tr>
<td>20</td>
<td>&gt;100,000</td>
<td>0.00</td>
<td>240</td>
</tr>
<tr>
<td>40</td>
<td>-</td>
<td>0.16</td>
<td>-</td>
</tr>
<tr>
<td>60</td>
<td>-</td>
<td>1.30</td>
<td>-</td>
</tr>
<tr>
<td>80</td>
<td>-</td>
<td>4.30</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>-</td>
<td>11.20</td>
<td>-</td>
</tr>
</tbody>
</table>

- Times are in seconds.
- There are 1646800 clauses for n=100
The Einstein Puzzle

Supposedly, Albert Einstein wrote this riddle, and said 98% of the world could not solve it.

- There are 5 houses in five different colors.
- In each house lives a person with a different nationality.
- These 5 owners drink a certain drink, smoke a certain brand of tobacco and keep a certain pet.
- No owners have the same pet, smoke the same tobacco, or drink the same drink.
- The question is: Who owns the fish?

Hints to Einstein Puzzle

- The Brit lives in the red house
- The Swede keeps dogs as pets
- The Dane drinks tea
- The green house is adjacent on the left of the white house
- The green house owner drinks coffee
- The person who smokes Pall Mall raises birds
- The owner of the yellow house smokes Dunhill
- The man living in the house right in the center drinks milk
Hints to Einstein Puzzle (cont)

- The Norwegian lives in the first house
- The man who smokes Blends lives next to the one who keeps cats
- The man who keeps horses lives next to the one who smokes Dunhill
- The owner who smokes Bluemaster drinks juice
- The German smokes Prince
- The Norwegian lives next to the blue house
- The man who smokes Blend has a neighbor who drinks water.

Specify Einstein Puzzle in SAT

- The houses are presented by 1, 2, 3, 4, 5.
- Definition of colors */
  - #define red 0
  - #define green 1
  - #define white 2
  - #define blue 3
  - #define yellow 4
- #define color(x,y) ((x)+5*(y))
- Answer: 3 9 15 17 21
Specify Einstein Puzzle in SAT

- The houses are presented by 1, 2, 3, 4, 5.
- Definition of nationality
  - #define brit 5
  - #define swede 6
  - #define dane 7
  - #define norwegian 8
  - #define german 9
- #define lives(x,y) ((x)+5*(y))
- Answer: 28 35 37 41 49

Specify Einstein Puzzle in SAT

- The houses are presented by 1, 2, 3, 4, 5.
- Definition of drinks
  - #define tea 10
  - #define coffee 11
  - #define water 12
  - #define juice 13
  - #define milk 14
- #define drinks(x,y) ((x)+5*(y))
- Answer: 52 59 61 70 73
Clauses in DIMACS Format

printf("p cnf 125 1000\n"); // actual clauses: 885

for (k = 0; k < 5; k++) {
    // every house has a color
    for (a = 1; a <= 5; a++) printf("%d ", color(a, k));
    printf("0\n");
    for (a = 1; a <= 5; a++) {
        for (b = 1; b < a; b++)     // a color can be used once
            printf("-%d -%d 0\n", color(a, k), color(b, k));
        for (b = 0; b < 5 ; b++) if (b != k)
            // a house can have only one color.
            printf("-%d -%d 0\n", color(a, k), color(a, b));
    }
}

Clauses in DIMACS Format

// The Brit lives in the red house
for (a = 1; a <= 5; a++) {
    printf("-%d %d 0\n", lives(a, brit), color(a, red));
    printf("%d -%d 0\n", lives(a, brit), color(a, red));
}

// The Swede keeps dogs as pets
for (a = 1; a <= 5; a++) {
    printf("-%d %d 0\n", lives(a, swede), pets(a, dog));
    printf("%d -%d 0\n", lives(a, swede), pets(a, dog));
}
Clauses in DIMACS Format

// The man living in the house right in the center drinks milk
printf("%d 0\n", drinks(3, milk));

// The Norwegian lives in the first house
printf("%d 0\n", lives(1, norwegian));

Clauses in DIMACS Format

// The man who smokes Blends lives next to the one
// who keeps cats
printf("-%d %d 0\n", smokes(1, Blends), pets(2, cat));
printf("-%d %d 0\n", smokes(5, Blends), pets(4, cat));
for (a = 2; a <= 4; a++) {
    printf("-%d %d %d 0\n",
            smokes(a, Blends),
            pets(a-1, cat), pets(a+1, cat));
}
Sato’s Result

- Input file "einstein.cnf" is open.
- Reading clauses in DIMACS's format.
- There are 885 input clauses (3 unit, 251 subsumed, 637 retained).

Model #1: (indices of true atoms)
- 3 9 15 17 21 28 35 37 41 49 52 59 61 70 73 77 81 90 93 99
- 103 106 112 120 124

- The number of found models is 1.
- There are 13 branches (1 succeeded, 9 failed, 0 jumped).

---------------- Stats ----------------
- Run time (seconds) 0.00
- Build time 0.00
- Search time 0.00
- Mallocated (K bytes) 96.49

------------------- Sudoku Puzzle ---------------
- Fill numbers between 1 and 9 on a 9x9 square such that each row, column and each small 3x3 square is a permutation of 1 to 9.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>4</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>9</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>
SUDOKU Puzzle

- Fill numbers between 1 and 9 on a 9x9 square such that each row, each column and each small 3x3 square is a permutation of 1 to 9.

5 6 3 2 8 1 4 7 9
4 9 2 7 6 3 8 1 5
1 8 7 5 4 9 6 2 3

9 2 4 8 3 5 7 6 1
7 3 8 4 1 6 9 5 2
6 5 1 9 2 7 3 8 4

2 1 6 3 7 4 5 9 8
8 4 5 6 9 2 1 3 7
3 7 9 1 5 8 2 4 6

How to Code the Puzzle

- Let \( p(x,y,z) \) be the boolean variable such that the number at \((x, y)\) is \( z \), where \( x, y, z \) are in \{ 1..9 \}
- In DIMACS Fomat, \( p(x,y,z) = 81(x-1)+9(y-1)+z \)
- The total number of boolean variables is \( 9^3 = 729 \).
How to Code the Puzzle

• At cell \((x, y)\), the number is between 1 and 9:
  \[ \text{p}(x, y, 1) \lor \text{p}(x, y, 2) \lor \text{p}(x, y, 3) \lor \ldots \lor \text{p}(x, y, 9) \]
  There are 81 such clauses.

• Row \(x\) is a permutation of 1..9: For any \(z\),
  \[ -\text{p}(x, 1, z) \lor -\text{p}(x, 2, z), \text{ } -\text{p}(x, 1, z) \lor -\text{p}(x, 3, z), \ldots \]
  There are 81*36 such clauses.

• Column \(y\) is a permutation of 1..9: For any \(z\),
  \[ -\text{p}(1, y, z) \lor -\text{p}(2, y, z), \text{ } -\text{p}(1, y, z) \lor -\text{p}(3, y, z), \ldots \]

• Each small square is a permutation of 1..9: For any \(z\),
  \[ -\text{p}(1, 1, z) \lor -\text{p}(2, 2, z), \text{ } -\text{p}(1, 1, z) \lor -\text{p}(2, 3, z), \ldots \]

Resolution (original DP)

• Iteratively apply resolution (consensus) to eliminate one variable each time
  – i.e., resolution between all pairs of clauses containing \(x\) and \(\neg x\)
  – formula satisfiability is preserved

• Stop applying resolution when,
  – Either empty clause is derived ⇒ instance is unsatisfiable
  – Or only clauses satisfied or with pure literals are obtained ⇒ instance is satisfiable

\[ \varphi = (a \lor c)(b \lor c)(d \lor c)(\neg a \lor \neg b \lor \neg c) \]
Eliminate variable \(c\)

\[ \varphi_1 = (a \lor \neg a \lor b)(b \lor \neg a \lor \neg b)(d \lor \neg a \lor \neg b) \]
\[ = (d \lor \neg a \lor \neg b) \]
Instance is SAT!
Elimination of variables

- Let $S_n$ be the set of input clauses.
- Let variables in $S$ be ordered as $x_1, x_2, \ldots, x_n$ and we eliminate them from $x_n$ to $x_1$ by the following algorithm:
  
  For $k := n$ to 1 step -1 do {
    Let $S_k$ be divided into $S_k = P_k \cup N_k \cup S'$, where
    - $P_k$ contains all clauses having positive $x_k$
    - $N_k$ contains all clauses having negative $-x_k$
    Let $S(k-1) := S' \cup \text{Resolution}(P_k, N_k, x_k)$
    If the empty clause is in $S(k-1)$, return UNSAT
  }
  return SAT.
- Assume $\text{Resolution}(P_k, N_k, x_k)$ apply the resolution to any pair of clauses from $P_k$ and $N_k$ on $x_k$. Duplicate literals in a clause are removed. Tautology clauses are removed from $S_k$.

Completeness of Resolution

- If the algorithm returns UNSAT, then $S$ is unsatisfiable.
  - Because resolution is sound, if the empty clause is generated, then $S$ must be unsatisfiable.
- If the algorithm returns SAT, then $S$ is satisfiable, i.e., $S$ has a model.
  - We construct an assignment by assigning truth values from $x_1$ to $x_n$ such that when $x_k$ is assigned, then all clauses in $S_k$ are true under the current assignment.
  - This assignment is a model when $x_n$ is assigned a truth value.
Completeness of Resolution

- The procedure for assigning truth values from x1 to xn:
  - At first, either P1 or N1 is empty, otherwise, the empty clause will be generated from Resolution(P1, N1, x1). We assign P1 to true if N1 is empty; otherwise P1 to false.
  - Suppose we have assigned a truth value to x1, x2, ..., x(k-1). If a clause in Nk after removing –xk becomes false, we assign false to xk; otherwise assign true to xk.

- Claim: All clauses in Sk are true in the current assignment after xk is assigned a truth value.

- Proof:
  - Basic case: k = 1.
  - Inductive hypothesis: After x(k-1) is assigned, clauses in S(k-1) are true.
  - Inductive case: The assignment of xk will make every clause in Sk true.

Gate CNF

\[ a \quad b \quad \rightarrow d \]

\[ \varphi_d = [d \equiv \neg(a \& b)] \]

\[ = [d \rightarrow \neg(a \& b)] \& [\neg(a \& b) \rightarrow d] \]

\[ = (\neg d \lor \neg a \lor \neg b)[\neg d \rightarrow (a \& b)] \]

\[ = (a \lor d)(b \lor d)(\neg a \lor \neg b \lor \neg d) \]
Converting Formula into CNF

- Replace *equivalence, xor, etc.* by *and* ($\&$), *or* ($|$), *negation* ($\neg$)
  - $x = y$ by $(\neg x \mid y) \& (x \mid \neg y)$
  - $x \oplus y$ by $(\neg x \mid \neg y) \& (x \mid y)$
  - $x \rightarrow y$ by $(\neg x \mid y)$

- Push *negation* as low as possible:
  - $\neg (x \mid y)$ by $(\neg x \& \neg y)$
  - $\neg (x \& y)$ by $(\neg x \mid \neg y)$
  - $\neg \neg x$ by $x$

- Distribute *or* over *and*
  - $x \mid (y \& z)$ by $(x \mid y) \& (x \mid z)$
  - $(y \& z) \mid x$ by $(y \mid x) \& (z \mid x)$

Techniques for Backtrack Search

- Conflict analysis
  - Clause/implicate recording
  - Non-chronological backtracking

- Incorporate and extend ideas from:
  - Resolution

- Formula simplification & Clause inference

- Randomization & Restarts
Clause Recording

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

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

Assume (decisions) \( c = 0 \) and \( f = 0 \)
Assign \( a = 0 \) and imply assignments
A conflict is reached: \((\neg d \lor \neg e \lor f)\) is unsatisfiable

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

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

:. create new clause: \((a \lor c \lor f)\)

Clause Recording

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

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

resolution

\[
(a \lor c \lor d)
\]

\[
(a \lor e)
\]

\[
(a \lor c \lor \neg e \lor f)
\]

\[
(a \lor c \lor f)
\]

Unit clause: prevents conflict and implies assignment \( a = 1 \)
Clause \((a \lor c \lor f)\) would have prevented the conflict!
Non-Chronological Backtracking

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

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

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

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

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

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

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

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

\[ \therefore \text{create new clause: } (\neg a \lor c \lor f) \]

Non-Chronological Backtracking

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

Apply resolution:

- new unsat clause \( (c \lor f) \)

\[ \therefore \text{backtrack to most recent decision: } f = 0 \]

\[ \therefore \text{created clauses/implicates: } (a \lor c \lor f), \]

\( (\neg a \lor c \lor f), \) and \( (c \lor f) \)
Circuit Satisfiability

\[ \phi = \left[ d \equiv \neg (ab) \right] \left[ e \equiv \neg (b \lor c) \right] \left[ f \equiv \neg d \right] \left[ g \equiv d \land e \right] \left[ h \equiv f \land g \right] \]

\[ \phi = h \left[ d \equiv \neg (ab) \right] \left[ e \equiv \neg (b \lor c) \right] \left[ f \equiv \neg d \right] \left[ g \equiv d \land e \right] \left[ h \equiv f \land g \right] \]
Equivalence Checking

- Combinational Circuits:

\[ C_A \]
\[ C_B \]
\[ z = 1 \]?

If \( z = 1 \) is unsatisfiable, the two circuits are equivalent!

SAT Problem Hardness in EDA

- Bounded Model Checking (BMC)
- Superscalar processor verification
- FPGA routing
- Equivalence Checking (CEC)
- Circuit Delay Computation
- Test Pattern Generation (ATPG):
  - Stuck-at, Delay faults, etc.
  - Redundancy Removal
- Noise analysis
- ...

Hardest
Easiest
Unknown
Conclusions

- Many recent SAT algorithms and (EDA) applications
- Hard Applications
  - Bounded Model Checking
  - Combinational Equivalence Checking
  - Superscalar processor verification
  - FPGA routing
- “Easy” Applications
  - Test Pattern Generation: Stuck-at, Delay faults, etc.
  - Redundancy Removal
  - Circuit Delay Computation
- Other Applications
  - Noise analysis, etc.

Conclusions

- Complete vs. Incomplete algorithms
  - Backtrack search (DP)
  - Resolution (original DP)
  - Recursive learning
  - Local search
- Techniques for backtrack search (infer implicates)
  - conflict-induced clause recording
  - non-chronological backtracking
  - resolution, SM and RL within backtrack search
  - formula simplification & clause inference conditions
  - randomization & restarts
Research Directions

- **Algorithms:**
  - Explore relation between different techniques
    - backtrack search; conflict analysis; recursive learning; branch-merge rule; randomization & restarts; clause inference; local search (?) BDDs (?)
  - Address specific solvers (circuits, incremental, etc.)
  - Develop visualization aids for helping to better understand problem hardness

- **Applications:**
  - Industry has applied SAT solvers to different applications
  - SAT research requires challenging and representative publicly available benchmark instances!