SAT-Solving: From Davis-Putnam to Zchaff and Beyond

Day 1: SAT Basics

Lintao Zhang

Microsoft Research
Automated Reasoning: Motivations

- As a curiosity of mathematicians and inventors
  - Demonstrator, Charles Stanhope, 1777
  - Logic Machine, William Stanley Jevons, 1869
- Artificial Intelligence and foundation of mathematics
  - Mechanical theorem proving
  - Reasoning on knowledge base
- Electronic Design Automation
  - ATPG
  - Logic synthesis
- Verification of digital systems
  - Equivalence checking
  - Model checking
  - Safety of programs, concurrent processes

Lintao Zhang
How to Perform Automatic Reasoning?

- **Modeling**: Abstract the problem into logic
  - Boolean propositional logic
  - Temporal logic
  - Set theory
  - First order logic

- **Proof**: Use automatic decision procedures to determine the correctness (*validity*) of the resulting logic
  - SAT Solvers and BDDs
  - Model Checker
  - Theorem Provers
Propositional Logic

- **Variable Domain:** True/False or 1/0
- **Logic operations:** and $\land \cdot$, or $\lor +$, not $\neg \,'$
  - It’s also easy to express **Imply** $\rightarrow$, **equivalence** $\leftrightarrow$
- If a and b are Boolean, then these are propositional formulas:
  - $a \cdot b + a' \cdot c$
  - $1\cdot a = 0$
  - $1+a = 1$
- These are not propositional logic:
  - $3 + x = x + 3$; -- Integer domain
  - $\forall a \exists b (a+b)(a'+b')$ -- Quantifiers
  - If $a = b$ then $f(a)=f(b)$ -- Uninterpreted function
- It is the basis of all other logics.
What is SAT?

- Boolean Satisfiability (SAT).
- Operates on Boolean Propositional Logic
- Check if a complex logical relationship can ever be true (or satisfiable)
  - x OR y is true when x is true or y is true (satisfiable)
  - x AND (NOT x) can never be true (unsatisfiable)
- Tautology Checking
- Looks easy, but gets hard very quickly as the size of the problem increases
  - Size measured in terms of:
    - Number of variables
    - Number of operations
Why is SAT Important?

- Theoretical importance
  - It’s the first NP-Complete problem discovered by Cook in 1971
- It’s everywhere
  - Automatic Test Pattern Generation
  - Combinational Equivalence Checking
  - Bounded Model Checking
  - AI Planning
  - Theorem Proving
  - Software modeling and verification
  - ... ...
- We have powerful SAT solvers that can solve practical problems
  - SAT solving has been well studied for at least 40 years.
  - Recent breakthroughs make SAT solver highly efficient
    - Can handle over a million variables and operations
    - Seen wide use in the industry
  - Can we do better?
Course Schedule

- 3-day mini-course
  - Today: Basics of SAT solving
  - Tomorrow: Efficient Implementation of SAT solvers
  - Wednesday: Recent Developments in SAT research
- Emphasis on Engineering, not math or just algorithms
- Lectures in the morning, projects and discussion in the afternoon
- Main course project: Implementing an SAT solver
  - Require some knowledge of C/C++ and STL
Boolean n-Space

\[ B = \{0, 1\} \]

\[ B^0 \quad B^1 \]

\[ B^2 = B \times B \]

\[ B^3 \]

\[ B^4 \]
Boolean Functions

\( f(x) : B^n \rightarrow B \quad B = \{0, 1\} \quad x = \{x_1, x_2, \ldots, x_n\} \)

- \( x_1, x_2, \ldots, x_n \) are variables
- Each vertex of \( B^n \) is mapped to either 0 or 1
- The on-set of \( f \) is \( \{x | f(x) = 1\} = f^1 = f^{-1}(1) \)
- The off-set of \( f \) is \( \{x | f(x) = 0\} = f^0 = f^{-1}(0) \)
- If \( f^1 = B^n \), \( f \) is a tautology
- If \( f^0 = B^n \), i.e. \( f = \phi \), \( f \) is not satisfiable
- If \( f(x) = g(x) \) for all \( x \in B^n \), then \( f \) and \( g \) are equivalent
- Also referred to as logic functions
- How many logic functions are there?
The truth table for a function \( f: \mathbb{B}^n \rightarrow \mathbb{B} \) is a tabular representation of its value at each of the \( 2^n \) vertices of \( \mathbb{B}^n \).

Example:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Intractable for large \( n \) (but canonical).

Canonical means that if two functions are equivalent, then their canonical representations are isomorphic.
Boolean Satisfiability

- Is there any satisfying assignment for the function, i.e., is there at least one point in the ON-set of the function?
- How hard is this?
  - Depends on how the function is represented.
    - Boolean n-cube, truth table
      - Easy once we have the representation
      - But representation size is exponential in $n$
    - How about other representation?
      - Boolean Formula
      - BDD
      - Circuit
**Literals**

- A literal is a variable or its negation.
  - $x_1$, $x_1'$ (also represented as $\neg x_1$)
- Literal $x_1$ represents a logic function $f$ where $f^1 = \{x|x_1=1\}$
- Literal $x_1'$ represents a logic function $g$ where $g^1 = \{x|x_1=0\}$

\[ f = x_1 \]
\[ g = x_1' \]
Boolean Formulas

- Boolean functions can be represented as formulas defined as catenations of:
  - Parenthesis $(,)$
  - Literals $x_1, x_1'$
  - Boolean operators $+$ (OR), $x$ or . (AND), NOT
  - NOT (Negation): $f' = h$ such that $h^1 = f^0$
  - AND (Conjunction): $(f \text{ AND } g) = h$ such that $h^1 = \{x | f(x) = 1 \text{ and } g(x) = 1\}$
  - OR (Disjunction): $(f \text{ OR } g) = h$ such that $h^1 = \{x | f(x) = 1 \text{ or } g(x) = 1\}$

- Usually replace $x$ with catenation
  - e.g. $x_1 \cdot x_2$ with $x_1 x_2$

- How many formulas can we have with $n$ variables?

- Examples:
  - $f = x_1 x_2' + x_1' x_2$
    $$= (x_1 + x_2) (x_1' + x_2')$$
  - $h = x_1 + x_2 x_3$
    $$= (x_1' (x_2' + x_3'))'$$
**Boolean Satisfiability (SAT)**

- Given a Boolean propositional formula, determine whether there exists a variable assignment that makes the formula evaluate to **true**.

- Formulas are often expressed in **Conjunctive Normal Form (CNF)**

\[(a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')\]

```
Variables  Literals  Clauses
```

Lintao Zhang
Boolean Satisfiability (SAT)

- Given a Boolean propositional formula, determine whether there exists a variable assignment that makes the formula evaluate to true.

- Formulas are often expressed in Conjunctive Normal Form (CNF)

\[(a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')\]
Boolean Satisfiability (SAT)

- Given a Boolean propositional formula, determine whether there exists a variable assignment that makes the formula evaluate to true.

- Formulas are often expressed in *Conjunctive Normal Form (CNF)*

\[(a+b+c)(a'+b'+c)(a'+b+c')(a+b'+c')\]

\[(a+b)(a'+b)(a+b')(a'+b')\]
Convert a Boolean Circuit into CNF

- Example: Combinational Equivalence Checking
Combinational Equivalence Checking

- Miter Circuit

Lintao Zhang
Modeling of Combinational Gates

\[(a + c')(b + c')(a' + b' + c)\]

\[(a' + c)(b' + c)(a + b + c')\]

\[(a' + b' + c')(a + b + c')(a + b' + c)(a' + b + c)\]

Lintao Zhang
From Combinational Equivalence Checking to SAT

(a’ + b’ + c’)(a + b + c’)(a + b’ + c)(a’ + b + c)
(a + d)(b’ + d)(a’ + b + d’)
(a’ + e)(b + e)(a + b’ + e’)
(d + f’)(e + f’)(d’ + e’ + f)
(c’ + f + g’)(c + f’ + g’)(c + f + g)(c’ + f’ + g)
g
From Combinational Equivalence Checking to SAT

\[(a' + b' + c')(a + b + c')(a + b' + c)(a' + b + c)(a' + d)(b' + d)(a + b + d')(a' + e)(b + e)(a + b' + e')(d + f')(e + f')(d' + e' + f)(c' + f + g')(c + f' + g')(c + f + g)(c' + f' + g)(g)\]
Convert an Arbitrary Boolean Formula into CNF

- It is possible to convert an arbitrary function into CNF
  - Without introducing new variables, the size of the resulting formula will grow exponentially
    - Not practical
  - By introducing intermediate variables, the size of the resulting formula can grow linearly
    - How?
      - Number of intermediate variable equal to the number of Boolean operations
      - The resulting formula will have the same satisfiability as the original one
- It’s sufficient for a SAT solver to solve problems in CNF
  - Almost all modern SAT solver operates on CNF
Complexity of SAT

- A CNF formula is said to belong to $k$-SAT if each clause of the formula contains no more than $k$ literals.

- Classic Result:
  - Cook 1971: 3-SAT problem is NP-Complete.
  - NP complete: Class of problems for which no known solutions exists that takes less than $O(2^n)$ steps. However, it has not been proved that the problem needs at least an exponential number of steps. The common conjecture is that it does.
  - $k$-SAT is NP-complete for $k \geq 3$.

- The obvious lower bound for a SAT problem with $n$ variables is $2^n$.

- Currently, the best lower bound for a SAT problem with $n$ variables is due to Paturi etc., E.g. for satisfiable 3-SAT, the complexity for finding a solution is $O(2^{0.448n})$. 

Lintao Zhang
SAT Problems with Polynomial Complexity

- Some special SAT classes can be solved in polynomial time.
  - If a problem is solvable in polynomial time, we can use special algorithms to solve them efficiently.
  - Part of the original problem may belong to a polynomial solvable class, it is possible to exploit this property during the solving process. (e.g. Larrabee).
  - During the solution process, a problem state may evolve to one that has a polynomial solution. We can exploit heuristics that are likely to reduce a problem to one that is solvable in polynomial time quickly (e.g. SATO).
- 2-SAT problems can be solved in linear time wrt the size of the problem (Aspvall, Plass and Tarjan, 1979).
- A Horn formula can be solved in linear time wrt the size of the formula.
Horn Formulas

- Horn sentences are often generated from knowledge base reasoning:
  - rules: if x, y, z are true, then r is true
  - xyz → r
  - a → b
    - If a is true, then b must be true to make the formula true
    - if a is false, then the formula is true
    - (a’ + b)
  - xyz → r : (x’ + y’ + z’ + r)

- A CNF formula is Horn if every clause has at most one positive literal
  - What does it mean if a clause contains no positive literal?
  - What does it mean if a clause contains only one positive literal and no negative literal?

- A Horn formula can be solved in linear time wrt the size of the formula.
  - Do unit implication until no unit clause exists
  - If conflict, the formula is unsatisfiable
  - Else the formula can be satisfied by assigning all the unassigned variables with value 0
Problem Hardness and Phase Transition

- Not all SAT problems are hard
  - Many practical SAT instances can be solved very efficiently
  - The theory of NP-completeness is based on worst-case complexity.
  - To explain the behavior of algorithms in practice, the theory of average-case complexity is more appropriate.

- Use random generated SAT instances to explore the hardness distribution
  - Very different characteristics from the instances generated from real world applications
  - But are of great theoretical interests
Fixed-clause length model

- Generated by selecting clauses uniformly at random from the set of all possible (non-trivial) clauses of a given length, random k-SAT.

- Three parameters: the number of variables N, the number of literals per clause K, and the number of clauses L.
  - Formulas with few clauses: under-constrained (usually satisfiable),
  - Formulas with many clauses: over-constrained (usually unsatisfiable)
  - Both under-constrained and over-constrained problems are much easier than problems of medium length
Phase transition behavior

- Problems which are very over-constrained are unsatisfiable and it is usually easy to determine this. Problems which are very under-constrained are satisfiable and it is usually easy to guess one of the many solutions.

- A phase transition tends to occur in between when problems are critically constrained, and it is difficult to determine if they are satisfiable or not.

- For random 2-SAT, the phase transition has been proven to occur at $L/N=1$.

- For random 3-SAT, the phase transition has been experimentally shown to occur around $L/N = 4.3$
Lintao Zhang

Hardness of 3SAT

Ratio of Clauses-to-Variables

DP Calls

- 50 var
- 40 var
- 20 var
The 4.3 Point

DP Calls

Ratio of Clauses-to-Variables

50 var
40 var
20 var

Probability

Mitchell, Selman, and Levesque 1991
Phase transition 2-, 3-, 4-, 5-, and 6-SAT
Threshold phenomena

- Threshold conjecture: for each $k$, there is some $c^*$ such that for each fixed value of $c < c^*$, random $k$-SAT with $n$ variables and $cn$ clauses is satisfiable with probability tending to 1 as $n \to \infty$, and when $c > c^*$, unsatisfiable with probability tending to 1.
- For the case of random 2-SAT, the conjecture has been shown true, and $c^* = 1$.
- Current status:
  - 3SAT threshold lies between $3.42 \sim 4.51$
The 2+p-SAT model

- Mixtures of problem classes, e.g., 2-SAT and 3-SAT ("moving between P and NP")
- Mixture of binary and ternary clauses
  
  \[ p = \text{fraction ternary} \]

  \[ p = 0.0 \quad \text{--- 2-SAT} \quad / \quad p = 1.0 \quad \text{--- 3-SAT} \]
Phase Transition for 2+p-SAT
**Computational Cost**

$p = 0.0, 0.2, 0.4, 0.6$, using tableau

Lintao Zhang
2+P Model

- $p < \sim 0.41$ --- model essentially behaves as 2-SAT
  - search proc. “sees” only binary constraints
  - smooth, continuous phase transition
- $p > \sim 0.41$ --- behaves as 3-SAT (exponential scaling)
  - abrupt, discontinuous scaling
SAT Algorithm: An Overview

- Davis, Putnam, 1960
  - Explicit resolution based
  - May explode in memory
- Davis, Logemann, Loveland, 1962
  - Search based.
  - Most successful, basis for almost all modern SAT solvers
  - Learning and non-chronological backtracking, 1996
- Stålmarcks algorithm, 1980s
  - Proprietary algorithm. Patented.
  - Commercial versions available
- Stochastic Methods, 1992
  - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
  - Local search and hill climbing
SAT Algorithm: An Overview

- Davis, Putnam, 1960
  - Explicit resolution based
  - May explode in memory

- Davis, Logemann, Loveland, 1962
  - Search based.
  - Most successful, basis for almost all modern SAT solvers
  - Learning and non-chronological backtracking, 1996

- Stålmarcks algorithm, 1980s
  - Proprietary algorithm. Patented.
  - Commercial versions available

- Stochastic Methods, 1992
  - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
  - Local search and hill climbing
Resolution

- Resolution of a pair of clauses with exactly **ONE** incompatible variable
  - Two clauses are said to have distance 1
  - \((a+b)(a'+c) = (a+b)(a'+c)(b+c)\)
Davis Putnam Algorithm


- Iteratively select a variable for resolution till no more variables are left.
- Can discard all original clauses after each iteration.

\[(a + b + c) (b + c' + f') (b' + e)\]

\[(a + c + e) (c' + e + f')\]

\[(a + e + f')\]

SAT

Potential memory explosion problem!

\[(a + b) (a + b') (a' + c) (a' + c')\]

\[(a) (a' + c) (a' + c')\]

\[(c) (c')\]

( )

UNSAT

Lintao Zhang
SAT Algorithm: An Overview

- Davis, Putnam, 1960
  - Explicit resolution based
  - May explode in memory
- Davis, Logemann, Loveland, 1962
  - Search based.
  - Most successful, basis for almost all modern SAT solvers
  - Learning and non-chronological backtracking, 1996
- Stålmarcks algorithm, 1980s
  - Proprietary algorithm. Patented.
  - Commercial versions available
- Stochastic Methods, 1992
  - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
  - Local search and hill climbing
Search Tree of SAT Problem

\[(x_1' + x_2')\]
\[(x_1' + x_2 + x_3')\]
\[(x_1' + x_3 + x_4')\]
\[(x_1 + x_4)\]

Lintao Zhang
Deduction Rules for SAT

- **Unit Literal Rule:** If an unsatisfied clause has all but one of its literals evaluate to 0, then the free literal must be implied to be 1.

  \[(a + b + c)(d' + e)(a + b + c' + d)\]

- **Conflicting Rule:** If all literals in a clause evaluate to 0, then the formula is unsatisfiable in this branch.

  \[(a + b + c)(d' + e)(a + b + c' + d)\]
Search Tree of SAT Problem

\[(x_1' + x_2')\]
\[(x_1' + x_2 + x_3')\]
\[(x_1' + x_3 + x_4')\]
\[(x_1 + x_4)\]
Search Tree of SAT Problem

\[(x_1' + x_2')\]
\[(x_1' + x_2 + x_3')\]
\[(x_1' + x_3 + x_4')\]
\[(x_1 + x_4)\]
Search Tree of SAT Problem

\[
\begin{align*}
(x_1' + x_2') \\
(x_1' + x_2 + x_3') \\
(x_1' + x_3 + x_4') \\
(x_1 + x_4)
\end{align*}
\]
Lintao Zhang

Search Tree of SAT Problem

\[(x_1' + x_2')\]
\[(x_1' + x_2 + x_3')\]
\[(x_1' + x_3 + x_4')\]
\[(x_1 + x_4)\]
Search Tree of SAT Problem

\[(x'_1 + x'_2)\]
\[(x'_1 + x_2 + x'_3)\]
\[(x'_1 + x_3 + x'_4)\]
\[(x_1 + x_4)\]
DLL Algorithm


- Basic framework for many modern SAT solvers
- Also known as DPLL for historical reasons
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)
Basic DLL Procedure - DFS

\[(a' + b + c)\]
\[(a + c + d)\]
\[(a + c + d')\]
\[(a + c' + d)\]
\[(a + c' + d')\]
\[(b' + c' + d)\]
\[(a' + b + c')\]
\[(a' + b' + c)\]
Basic DLL Procedure - DFS

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

\(\iff\) Decision
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Decision
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Implication Graph

Lintao Zhang
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Implication Graph

Conflict!
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

⇐ Backtrack
Basic DLL Procedure - DFS

Lintao Zhang

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

\[d=1\]
\[c=1\]
\[a=0\]
\[d=1\]
\[c=1\]
\[d=0\]

Conflict!

Forced Decision
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

⇐ Backtrack
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Forced Decision
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

c=0
(a + c' + d')
d=0
(a + c' + d)
d=1

Conflict!

Decision
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)
Basic DLL Procedure - DFS

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

\(c=1\)
\(d=0\)
\(\text{Conflict!}\)

\(\text{Forced Decision}\)
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

⇐ Backtrack
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Forced Decision
Basic DLL Procedure - DFS

(a' + b + c)  
(a + c + d)  
(a + c + d')  
(a + c' + d)  
(a + c' + d')  
(b' + c' + d)  
(a' + b + c')  
(a' + b' + c)
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

Conflict!
Basic DLL Procedure - DFS

(a' + b + c)
(a + c + d)
(a + c + d')
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

⇐ Backtrack
(a' + b + c)
(a + c + d)
(a + c' + d)
(a + c' + d')
(b' + c' + d)
(a' + b + c')
(a' + b' + c)

(a' + b + c)  
(a + c + d)  
(a + c' + d)  
(a + c' + d')  
(b' + c' + d)  
(a' + b + c')  
(a' + b' + c)

Forced Decision

Lintao Zhang
Basic DLL Procedure - DFS

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

\[a=1\]
\[b=1\]
\[c=1\]
\[d=1\]
Basic DLL Procedure - DFS

\[
\begin{align*}
(a' + b + c) \\
(a + c + d) \\
(a + c + d') \\
(a + c' + d) \\
(a + c' + d') \\
(b' + c' + d) \\
(a' + b + c') \\
(a' + b' + c)
\end{align*}
\]

\[
\begin{array}{l}
\text{a} & \quad 0 & \quad 1 \\
\text{b} & \quad 0 & \quad 1 \\
\text{c} & \quad 0 & \quad 1 \\
\text{d} & \quad 0 & \quad 1 \\
\text{c} & \quad 0 & \quad 1 \\
\text{a} & \quad 0 & \quad 1 \\
\text{b} & \quad 0 & \quad 1 \\
\text{c} & \quad 0 & \quad 1 \\
\text{d} & \quad 0 & \quad 1 \\
\text{c} & \quad 0 & \quad 1 \\
\text{a} & \quad 0 & \quad 1 \\
\text{b} & \quad 0 & \quad 1 \\
\text{c} & \quad 0 & \quad 1 \\
\text{d} & \quad 0 & \quad 1 \\
\text{c} & \quad 0 & \quad 1
\end{array}
\]

SAT
Implications and Boolean Constraint Propagation

- **Implication**
  - A variable is forced to be assigned to be True or False based on previous assignments.

- **Unit clause rule (rule for elimination of one literal clauses)**
  - An **unsatisfied** clause is a **unit** clause if it has exactly one unassigned literal.

\[(a + b' + c)(b + c')(a' + c')\]

\[a = T, \ b = T, \ c \text{ is unassigned}\]

- The unassigned literal is implied because of the unit clause.

- **Boolean Constraint Propagation (BCP)**
  - Iteratively apply the unit clause rule until there is no unit clause available.

- Workhorse of DLL based algorithms.
Features of DLL

- Eliminates the exponential memory requirements of DP
- Exponential time is still a problem
- Limited practical applicability – largest use seen in automatic theorem proving
- The original DLL algorithm has seen a lot of success for solving random generated instances.
Some Notes

- There are another rules proposed by the original DLL paper, which is seldom used in practice
  - **Pure literal rule**: if a variable only occur in one phase in the clause database, then the literal can be simply assigned with the value *true*

- The original DP paper also included the unit implication rule to simplify the clauses generated from resolution
  - Still may result in memory explosion

- DLL and DP algorithms are tightly related
  - Fundamentally, both are based on the resolution operation
SAT Algorithm: An Overview

- Davis, Putnam, 1960
  - Explicit resolution based
  - May explode in memory
- Davis, Logemann, Loveland, 1962
  - Search based.
  - Most successful, basis for almost all modern SAT solvers
  - Learning and non-chronological backtracking, 1996
- Stålmarcks algorithm, 1980s
  - Proprietary algorithm. Patented.
  - Commercial versions available
- Stochastic Methods, 1992
  - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
  - Local search and hill climbing
Stålmarck’s Algorithm

M. Sheeran and G. Stålmarck “A tutorial on Stålmarck’s proof procedure”, *Proc. FMCAD*, 1998

- **Algorithm:**
  - Using triplets to represent formula
    - Closer to a circuit representation
  - Branch on variable relationships besides on variables
    - Ability to add new variables on the fly
  - Breadth first search over all possible trees in increasing depth
Stålmarck’s algorithm (A Vastly Simplified Version)

- Try both sides of a branch to find forced decisions (relationships between variables)

\[(a + b) \ (a' + c) \ (a' + b) \ (a + d)\]
Stålmarck’s algorithm (A Vastly Simplified Version)

- Try both sides of a branch to find forced decisions

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

\[
\begin{align*}
\text{a=0} & \quad \text{b=1} \\
\text{d=1} & \quad \text{a=0} \Rightarrow \text{b=1, d=1}
\end{align*}
\]
Stålmárck’s algorithm (A Vastly Simplified Version)

- Try both side of a branch to find forced decisions

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

- \(a = 1 \Rightarrow b = 1, d = 1\)
- \(a = 0 \Rightarrow b = 1, d = 1\)
- \(a = 1 \Rightarrow b = 1, c = 1\)
Stålmarck’s algorithm (A Vastly Simplified Version)

- Try both sides of a branch to find forced decisions

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

- Repeat for all variables
- Repeat for all pairs, triples,… till either SAT or UNSAT is proved

\[
\begin{align*}
a=0 & \Rightarrow b=1, d=1 \\
a=1 & \Rightarrow b=1, c=1 \\
\Rightarrow & \ b=1
\end{align*}
\]
SAT Algorithm: An Overview

- Davis, Putnam, 1960
  - Explicit resolution based
  - May explode in memory
- Davis, Logemann, Loveland, 1962
  - Search based.
  - Most successful, basis for almost all modern SAT solvers
  - Learning and non-chronological backtracking, 1996
- Stålmarcks algorithm, 1980s
  - Proprietary algorithm. Patented.
  - Commercial versions available
- Stochastic Methods, 1992
  - Unable to prove unsatisfiability, but may find solutions for a satisfying problem quickly.
  - Local search and hill climbing
Local Search (GSAT, WSAT)


- View the solution space as a set of points connected to each other.
- There is cost function which needs to be minimized that can be computed for each point.
- Local search involves starting at some point in the solution space, and moving to adjacent points in an attempt to lower the cost function.
- The search is said to be greedy if it does not ever increase the cost function.
Local Search for Max-SAT

- MAX-SAT:
  - Find an assignment that satisfies the most number of clauses
  - Cost function for a given assignment: number of unsatisfied clauses
- Local search has been shown to work well for MAX-SAT
- Cost function for SAT?
  - Can continue to use number of unsatisfied clauses
  - However, only points with a cost function of 0 are of interest
Algorithm of GSAT

Procedure GSAT
for i := 1 to MAX-TRIES
    T := a randomly generated truth assignment
    for j := 1 to MAX-FLIPS
        if T satisfies α then return T
        flip the variable that results in the greatest decrease in the number of unsatisfied clauses (decrease ≥ 0)
    end for
end for
return “No satisfying assignment found”

- decrease = 0 is referred to as a “sideways” move
- sequence of sideways moves is a “plateau”
- success depends on ability to move between successively lower plateaus
Properties of GSAT

- Seems to work well on randomly generated 3-CNF problems
- Can get stuck in a local minima
- Not guaranteed to be complete

![Figure 1. GSAT’s search space on a randomly-generated 100 variable 3CNF formula with 430 clauses.](image)
Getting out of Local Minima

- **Random Walk Strategy**
  - with probability $p$, pick a variable occurring in some unsatisfied clause and flip its assignment;
  - with probability $(1-p)$, follow the standard GSAT scheme, i.e make the best possible local move

- **Random Noise Strategy**
  - similar to random walk, except that do not restrict the variable to be flipped to be in an unsatisfied clause

- **Simulated Annealing**
  - make random flips
  - probabilistically accept “bad moves”
Conclusions about Local Search

- Many local search algorithms exist
  - GSAT, WalkSAT, DLM etc.
  - Differs on how to get out of local minimum
- Incomplete, unable to prove unsatisfiability
  - How to make local search complete is still an open question
- Can be vastly superior than systematic search based algorithms on certain satisfiable formulas
- Has some application in AI planning, limited use in EDA or formal verification