SAT Solving
Part 1: Motivation, normal form translations and easy problem classes

Uwe Egly
Vienna University of Technology
Institute of Information Systems
Knowledge-Based Systems Group

May 2, 2010
Outline

Introduction and motivation

Translations to clausal normal form

Easy classes for the satisfiability problem

Concluding remarks for the first part
What is SAT and QSAT?

- Boolean satisfiability (SAT) and quantified B. SAT (QSAT) (we will consider formulas with Boolean quantifiers later)
- Operates on Boolean formulas (BFs) and quantified BFs (often restricted to normal forms; see later)
- Boolean case: Is a given BF satisfiable?
- Tautology, entailment, etc. checking reduced to SAT
- Looks easy, but gets hard very quickly as the size of the problem/formula increases
Why is SAT Important?

- Theoretical importance:
  - First NP-Complete problem discovered by S. A. Cook
  - It is everywhere
    - Automatic Test Pattern Generation
    - Combinational Equivalence Checking
    - Various AI problems like, e.g., planning
    - Theorem Proving
    - Software/hardware modeling and verification, . . .

- SAT solvers available that can solve practical problems
  - SAT solving has been well studied for at least 40 years
  - Recent breakthroughs: Can handle over a million variables
  - Seen wide use in the industry, but still a lot of problems
  - Can we do better?
The Early Machines and SAT Solvers

- 1869: William Stanley Jevons: *Logical Piano*
  process logical problems faster than humans
- 1885: Allan Marquand proposed electrical version of Piano
- 1936: Benjamin Burack built the 1. electronic logic machine
- 1947: Theodore A. Kalin and William Burckhart
  built a machine to SAT check prop. formulas with $\leq 12$ vars
- Computers arrived at the horizon and with them new opportunities
50 Years of SAT Solving Algorithms

- 1952: Quine ≈ 10 vars
- 1960: DP ≈ 10 vars
- 1962: DLL ≈ 10 vars
- 1986: BDDs ≈ 100 vars
- 1988: SOKRATES ≈ 3k vars
- 1992: GSAT ≈ 300 vars
- 1996: Stalmarck ≈ 1k vars
- 1996: GRASP ≈ 1k vars
- 1996: HANNIBAL ≈ 3k vars
- 1996: SATO ≈ 1k vars
- 2001: Chaff ≈ 10k vars
- 2002: Berkmin ≈ 10k vars
Modern SAT solvers often restrict input to normal forms (expected: easier language allows more efficient data structures)

We have already seen normal forms: **NNF** and **CNF**

- NNF characterized by two conditions:
  1. Negation signs occur only in front of atoms
  2. The only connectives are $\land$ and $\lor$

- NNF of $\phi$ (denoted by $\text{nnf}(\phi)$) and $\phi$ are equivalent!

Often: **Conjunctive Normal Form** (CNF)

- Conjunction (or set) of clauses (=disjunctions of literals)
- Often: clauses as sets (causes problems sometimes)

Different methods for translating BFs into CNFs yield different behaviour wrt proof search and proof complexity
The Traditional Approach for a CNF Translation

- Based on the application of distributivity laws
- Start with the formula $\phi$ and translate it to NNF
- Take $\text{nnf}(\phi)$ and replace the left side of the following equivalences by the right side (order does not matter!)
  
  1. $\phi \lor (\psi \land \chi) \equiv (\phi \lor \psi) \land (\phi \lor \chi)$
  2. $(\psi \land \chi) \lor \phi \equiv (\psi \lor \phi) \land (\chi \lor \phi)$
- Observe that $\text{nnf}(\phi) \equiv \text{cnf}(\phi) \equiv \phi$ holds
Exa: Transform $\phi$: $(p \land q \rightarrow r) \rightarrow (q \rightarrow r)$ to CNF

$$\text{nnf}(\phi): (p \land q \land \neg r) \lor (\neg q \lor r)$$

Formula

$$(p \lor \neg q \lor r) \land (q \lor \neg r \lor (\neg q \lor r))$$

$$(p \lor \neg q \lor r) \land (q \lor \neg q \lor r) \land (\neg r \lor \neg q \lor r)$$

Two disadvantages:

1. Disruption of the formula’s structure
2. $cnf(\phi)$ can be exponentially longer than $\phi$

Important concept for later: polarities of subformulas
Polarity Labels ($+, -, \pm$) of Subformulas

\[ (-\phi_1)^+ \rightsquigglyrightarrow (-\phi_1^-)^+ \]
\[ (-\phi_1)^- \rightsquigglyrightarrow (-\phi_1^+)^- \]
\[ (-\phi_1)^\pm \rightsquigglyrightarrow (-\phi_1^\pm)^\pm \]
\[ (\phi_1 \circ \phi_2)^q \rightsquigglyrightarrow (\phi_1^q \circ \phi_2^q)^q \]
\[ (\phi_1 \rightarrow \phi_2)^+ \rightsquigglyrightarrow (\phi_1^- \rightarrow \phi_2^+)^+ \]
\[ (\phi_1 \rightarrow \phi_2)^- \rightsquigglyrightarrow (\phi_1^+ \rightarrow \phi_2^-)^- \]
\[ (\phi_1 \rightarrow \phi_2)^\pm \rightsquigglyrightarrow (\phi_1^\pm \rightarrow \phi_2^\pm)^\pm \]
\[ (\phi_1 \leftrightarrow \phi_2)^q \rightsquigglyrightarrow (\phi_1^\pm \leftrightarrow \phi_2^\pm)^q \]

for $q \in \{+, -, \pm\}$ and $\circ \in \{\lor, \land\}$

$\Sigma^q(\phi)$: all subformula occurrences of $\phi$ occurring in polarity $q$

For simplicity, we restrict ourselves to input formulas without $\leftrightarrow$
Structure-preserving (or Definitional) NFTs

The Basic Idea

- Well known in logic (occurred relatively late in ATP and theory (Tseitin 1968))
- Consider the input formula $\phi$ as a tree
- Label each subformula occurrence (SFO) with a **new atom** (atom neither occurs in $\phi$ nor is it introduced before)
- Construct equivalences of the form
  
  $$l_\phi \leftrightarrow (l_{\phi_1} \circ l_{\phi_2})$$

  for SFOs $\phi_1 \circ \phi_2$

  where $l_\psi$ is the label for SF(O) $\psi$.
- Translate each $l_\phi \leftrightarrow (l_{\phi_1} \circ l_{\phi_2})$ to CNF using the NFT above
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

$$\begin{array}{c}
\land^+ \\
l_1 \\
p^+ \\
\rightarrow^+ \\
q^+ \\
r^- \\
\rightarrow^- \\
q^- \\
r^+ \\
\rightarrow^+ \\
l_1 \\
\leftrightarrow p
\end{array}$$
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

\[ \begin{array}{c}
\rightarrow^+ \\
\rightarrow^- \\
\land^+ \\
\end{array} \]

\[ \begin{array}{c}
l_1 \leftrightarrow p \\
l_2 \leftrightarrow q \\
\end{array} \]
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

- $\land^+$
- $\rightarrow^-$
- $\rightarrow^+$

$p^+ \quad q^+ \quad l_3^-$

Step 1: Label the Formula Tree

$l_1 \leftrightarrow p$
$l_2 \leftrightarrow q$
$l_3 \leftrightarrow r$
Example for a Translation: $\phi: (p \land q \to r) \to (q \to r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

$\land^+$

$\rightarrow^-$

$p^+$

$q^+$

$r^-$

$l_3$

$l_4$

$\rightarrow^+$

$q^-$

$r^+$

$l_1$

$l_2$

$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

$l_4 \leftrightarrow q$
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

```
          →+
         / | \
     →−  ∧+   →+
    / \\   / \ \\
  p+   q+ l3  r−  l4  r+  l5
  / \   /   \   /   \   /   \   /   \   /   \   /   \   /   \   /   \   /   \  
 l1   l2   l3  l4  l5
```

$p \leftrightarrow l_1$
$q \leftrightarrow l_2$
$r \leftrightarrow l_3$
$q \leftrightarrow l_4$
$r \leftrightarrow l_5$
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

$\rightarrow^+$

$\rightarrow^-$

$\land^+$

$p^+$

$q^+$

$l_1$ $l_2$ $l_3$ $l_4$ $l_5$

$l_6$ $r^-$

$l_1$ $l_2$ $l_3$ $l_4$ $l_5$

$l_6$ $r^+$

$l_1$ $l_2$ $l_3$ $l_4$ $l_5$

$l_6$ $p \land q$

$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

$l_4 \leftrightarrow q$

$l_5 \leftrightarrow r$

$l_6 \leftrightarrow p \land q$
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

```
         →+
        /   \
       →-   →+
      /     /   /
 l_6  ∧+  r-  q-  r+
     /    / \
 l_1  q+  l_3  l_4  l_5
     /     /   \
 l_2  r   l_4  l_5
```

- $l_1 \leftrightarrow p$
- $l_2 \leftrightarrow q$
- $l_3 \leftrightarrow r$
- $l_4 \leftrightarrow q$
- $l_5 \leftrightarrow r$
- $l_6 \leftrightarrow l_1 \land l_2$
Example for a Translation: \( \phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r) \)

Step 1: Label the Formula Tree

Tree of \( \phi^+ \) with polarities

\[
\begin{align*}
l_7 & \quad \rightarrow^- \\
l_6 & \quad \land^+ \\
l_3 & \quad r^- \\
l_4 & \quad q^- \\
l_5 & \quad r^+ \\
l_1 & \quad p^+ \\
l_2 & \quad q^+ \\
l_6 & \quad l_1 \land l_2 \\
l_7 & \quad p \land q \rightarrow r
\end{align*}
\]

\[
\begin{align*}
l_1 & \leftrightarrow p \\
l_2 & \leftrightarrow q \\
l_3 & \leftrightarrow r \\
l_4 & \leftrightarrow q \\
l_5 & \leftrightarrow r \\
l_6 & \leftrightarrow l_1 \land l_2 \\
l_7 & \leftrightarrow p \land q \rightarrow r
\end{align*}
\]
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

```
    l_7 ->+
     |  /
   r- | q- | r+
  /   /   /
l_6 l_3 l_4 l_5
```

Correspondences:

- $l_1 \leftrightarrow p$
- $l_2 \leftrightarrow q$
- $l_3 \leftrightarrow r$
- $l_4 \leftrightarrow q$
- $l_5 \leftrightarrow r$
- $l_6 \leftrightarrow l_1 \land l_2$
- $l_7 \leftrightarrow l_6 \rightarrow l_3$
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

- $l_1 \leftrightarrow p$
- $l_2 \leftrightarrow q$
- $l_3 \leftrightarrow r$
- $l_4 \leftrightarrow q$
- $l_5 \leftrightarrow r$
- $l_6 \leftrightarrow l_1 \land l_2$
- $l_7 \leftrightarrow l_6 \rightarrow l_3$
- $l_8 \leftrightarrow l_4 \rightarrow l_5$
Example for a Translation: $\phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of $\phi^+$ with polarities

- $l_1 \leftrightarrow p$
- $l_2 \leftrightarrow q$
- $l_3 \leftrightarrow r$
- $l_4 \leftrightarrow q$
- $l_5 \leftrightarrow r$
- $l_6 \leftrightarrow l_1 \land l_2$
- $l_7 \leftrightarrow l_6 \rightarrow l_3$
- $l_8 \leftrightarrow l_4 \rightarrow l_5$
- $l_9 \leftrightarrow l_7 \rightarrow l_8$
Example for a Translation: \( \phi: (p \land q \rightarrow r) \rightarrow (q \rightarrow r) \)

Step 2: Translate the “Labeling Formulas” to Clauses

<table>
<thead>
<tr>
<th>Equivalences for SFOs in ( \phi )</th>
<th>Associated Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>( l_1 \leftrightarrow p )</td>
<td>( \neg l_1 \lor p )</td>
</tr>
<tr>
<td>( l_2 \leftrightarrow q )</td>
<td>( \neg l_2 \lor q )</td>
</tr>
<tr>
<td>( l_3 \leftrightarrow r )</td>
<td>( \neg l_3 \lor r )</td>
</tr>
<tr>
<td>( l_4 \leftrightarrow q )</td>
<td>( \neg l_4 \lor q )</td>
</tr>
<tr>
<td>( l_5 \leftrightarrow r )</td>
<td>( \neg l_5 \lor r )</td>
</tr>
<tr>
<td>( l_6 \leftrightarrow l_1 \land l_2 )</td>
<td>( \neg l_6 \lor l_1 \lor l_2 )</td>
</tr>
<tr>
<td>( l_7 \leftrightarrow l_6 \rightarrow l_3 )</td>
<td>( \neg l_7 \lor \neg l_6 \lor l_3 )</td>
</tr>
<tr>
<td>( l_8 \leftrightarrow l_4 \rightarrow l_5 )</td>
<td>( \neg l_8 \lor \neg l_4 \lor l_5 )</td>
</tr>
<tr>
<td>( l_9 \leftrightarrow l_7 \rightarrow l_8 )</td>
<td>( \neg l_9 \lor \neg l_7 \lor l_8 )</td>
</tr>
</tbody>
</table>
## Defining the Translations

<table>
<thead>
<tr>
<th>Formula $\phi^q$</th>
<th>$C_1(\phi)^q$</th>
<th>$C_2(\phi)^q$</th>
<th>$C_3(\phi)^q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p^+$</td>
<td>$\neg l_p \lor p$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$p^-$</td>
<td>$\neg l_\phi \lor \neg l_{\phi_1}$</td>
<td>$l_\phi \lor l_{\phi_1}$</td>
<td>$l_\phi \lor \neg l_{\phi_1} \lor \neg l_{\phi_2}$</td>
</tr>
<tr>
<td>$(\neg \phi_1)^+$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(\neg \phi_1)^-$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(\phi_1 \land \phi_2)^+$</td>
<td>$\neg l_\phi \lor l_{\phi_1} \lor l_{\phi_2}$</td>
<td>$l_\phi \lor \neg l_{\phi_1}$</td>
<td>$l_\phi \lor \neg l_{\phi_2}$</td>
</tr>
<tr>
<td>$(\phi_1 \lor \phi_2)^-$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(\phi_1 \rightarrow \phi_2)^+$</td>
<td>$\neg l_\phi \lor \neg l_{\phi_1} \lor l_{\phi_2}$</td>
<td>$l_\phi \lor \neg l_{\phi_1}$</td>
<td>$l_\phi \lor \neg l_{\phi_2}$</td>
</tr>
<tr>
<td>$(\phi_1 \rightarrow \phi_2)^-$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For any $\phi$, $C^q_\phi$ denotes the clauses for $\phi^q$ ($q \in \{+,-\}$)
The Structure-preserving Normal Forms

The definitional form, $\delta(\phi)$, of $\phi$ is $\hat{\delta}(\phi) \cup \{l_\phi\}$ with

$$\hat{\delta}(\phi) : \{C^+_{\psi}, C^-_{\psi} \mid \psi \in \Sigma^+(\phi)\} \cup \{C^+_{\psi}, C^-_{\psi} \mid \psi \in \Sigma^-(\phi)\}$$

The p-definitional form, $\delta^+_p(\phi)$, of $\phi$ is $\hat{\delta}^+_p(\phi) \cup \{l_\phi\}$ with

$$\hat{\delta}^+_p(\phi) : \{C^q_{\psi} \mid \psi \in \Sigma^+(\phi)\} \cup \{C^r_{\psi} \mid \psi \in \Sigma^-(\phi)\}$$

where $\{r\} = \{+, -\} \setminus \{q\}$

- Sat-equivalence: $\phi$ has a model iff $\delta(\phi)$ ($\delta^+_p(\phi)$) has one
- Several variants and optimizations available, e.g., no label for atoms or negations, don’t translate clauses, etc.
Properties of the Translation

- Retains the structure of the formula (by labels for SFOs)
- For each SFOs, there are at most three clauses
- Each clause has at most three literals
- Normal form is linear-time computable
- $\phi$ and its definitional translation not logically equivalent (new labels change signature $\implies$ logical equivalence lost)
- $\phi$ has a model iff its definitional translation has one holds (this is the important prop.; cf 1st order ATP and Skolemization)
- Models of $\phi$ and its def. translation can be translated
- Generalizations used to extend calculi by extensions (resulting in stronger calculi which p-simulate, e.g., the cut rule)
An Application of the Translation: Circuits to CNF

\[ p \land q \geq 1 \quad O_1 \]
\[ r \geq 1 \quad N_1 \]
\[ O_2 \]

\[ s \land O_2 \geq 1 \quad A_2 \]
An Application of the Translation: Circuits to CNF

Use the definitional translation to get a sat-equivalent clause set
Historical Remarks on Structure-preserving NFTs

- As a starting point: Davis’ article in Handbook on AR (HAR, edited by J.A. Robinson and A. Voronkov, 2001)
- Tseitin 1968: definit. translation + the extension principle
- Cook and Reckhow 1979 (JSL): limited extension (compare calculi with (limited) extension with calculi wo)
- Eder 1984, 1992: definitional translation for first-order logic
- Plaisted and Greenbaum 1986: mainly p-definitional translation for first-order logic + some optimizations
- Boy de la Tour 1992: mix of structure-preserving and traditional translations; goal: get short normal form
- Baaz et al.: Normal Form Transformations in HAR (implications of different NFTs to proof complexity in FOL)
Algorithms for 2-SAT

- Special case of a CNF: each clause has at most 2 literals
- Often used as a simplification procedure in (Q)SAT solvers
- Unlike 3-SAT, 2-SAT is decidable in polynomial time (degree of the polynomial depends on the method used)
- E.g., for resolution, there is a quadratic time procedure
- We discuss linear procedure based on implication graphs
  - Aspvall, Plass, Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. IPL 8(3) 121-123, 1979 (Err. 14(4) 195, 1982)
### Implications Associated to a 2-CNF

<table>
<thead>
<tr>
<th>Clause $C$</th>
<th>Implication(s) $I(C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$</td>
<td>$\neg p \rightarrow p$</td>
</tr>
<tr>
<td>$\neg p$</td>
<td>$p \rightarrow \neg p$</td>
</tr>
<tr>
<td>$p \lor q$</td>
<td>$\neg p \rightarrow q$ $\neg q \rightarrow p$</td>
</tr>
<tr>
<td>$p \lor \neg q$</td>
<td>$p \rightarrow q$ $\neg q \rightarrow \neg p$</td>
</tr>
<tr>
<td>$\neg p \lor q$</td>
<td>$p \rightarrow \neg q$ $q \rightarrow \neg p$</td>
</tr>
<tr>
<td>$\neg p \lor \neg q$</td>
<td>$\neg p \lor \neg q$ $\neg p \lor \neg q$</td>
</tr>
</tbody>
</table>

$u \rightarrow v$ means: "If literal $u$ is true then literal $q$ is also true"
Construction of the Implication Graph (IG)

Given a 2-CNF $\phi$, construct the IG $G(\phi) = (V_\phi, E_\phi)$:

$V_\phi = \{ p, \neg p \mid \text{atom } p \text{ occurs in } \phi \}$

$E_\phi = \{ I(C) \mid C \in \phi \}$

Example: $C_1 : p \lor q \quad C_2 : \neg p \lor q$
Construction of the Implication Graph (IG)

Given a 2-CNF $\phi$, construct the IG $G(\phi) = (V_\phi, E_\phi)$:

$$V_\phi = \{ p, \neg p \mid \text{atom } p \text{ occurs in } \phi \}$$

$$E_\phi = \{ I(C) \mid C \in \phi \}$$

Example: $C_1 : p \lor q$  $C_2 : \neg p \lor q$

\[
\begin{array}{ccc}
p & q \\
\neg p & \neg q
\end{array}
\]
Construction of the Implication Graph (IG)

Given a 2-CNF $\phi$, construct the IG $G(\phi) = (V_\phi, E_\phi)$:

\[
V_\phi = \{ p, \neg p \mid \text{atom } p \text{ occurs in } \phi \}
\]
\[
E_\phi = \{ I(C) \mid C \in \phi \}
\]

Example: $C_1 : p \lor q \quad C_2 : \neg p \lor q$

![Diagram of the IG for $C_1$ and $C_2$]
Construction of the Implication Graph (IG)

Given a 2-CNF $\phi$, construct the IG $G(\phi) = (V_\phi, E_\phi)$:

$V_\phi = \{p, \neg p \mid \text{atom } p \text{ occurs in } \phi\}$

$E_\phi = \{I(C) \mid C \in \phi\}$

Example: $C_1 : p \lor q$ \hspace{1cm} $C_2 : \neg p \lor q$

Next: Construct all strongly connected components of $G(\phi)$
A linear 2-SAT Procedure

Reminder: **Strongly connected components**

- A directed graph is called **strongly connected** if each node is reachable from each other node via a path
- A **strongly connected component** (SCC) is a maximal strongly connected subgraph
- Tarjan (1972) provides a linear-time algorithm to compute all SCCs of a (directed) graph

**Important link and consequence:**

A 2-CNF $\phi$ is SAT iff no $p$ and $\neg p$ belong to the same SCC

2-SAT is solvable in linear time
SCC(G): Compute all SCCs of Directed Graph G

**Input:** Directed graph G  
**Output:** All SCCs of G

1. Call DFS(G) to compute finishing time $f(u)$ for each vertex $u$
2. Compute transpose $G^t$ of $G$ (reverse edge arrows)
3. Call DFS($G^t$), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
4. Output the vertices of each tree in DFS-forest obtained by the second DFS call as a separate SCC
SCC(G) on Example Graph G: Part 1
1. Call DFS($G$) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS($G$) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS(G) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS($G$) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS\((G)\) to compute finishing time \(f(u)\) for each vertex \(u\)
1. Call DFS(G) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS($G$) to compute finishing time $f(u)$ for each vertex $u$.
1. Call DFS(\(G\)) to compute finishing time \(f(u)\) for each vertex \(u\)
1. Call DFS($G$) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS(\(G\)) to compute finishing time \(f(u)\) for each vertex \(u\)
1. Call DFS(G) to compute finishing time $f(u)$ for each vertex $u$.
1. Call DFS($G$) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS(G) to compute finishing time $f(u)$ for each vertex $u$. 

To illustrate this, consider the example graph $G$ with vertices $a, b, c, d, e, f, g, h$ and the following edge labels:

- $a$ to $e$: 13
- $e$ to $a$: 12
- $a$ to $b$: 1
- $b$ to $a$: 11
- $b$ to $c$: 1/10
- $c$ to $b$: 2/7
- $c$ to $d$: 8/9
- $d$ to $c$: 5/6
- $d$ to $h$: 3/4
- $h$ to $f$: 13
- $f$ to $b$: 3/4
- $f$ to $g$: 8/9
- $g$ to $f$: 5/6

Using DFS, the finishing time for each vertex is determined as follows:

- Vertex $a$: finishing time 13
- Vertex $b$: finishing time 11
- Vertex $c$: finishing time 1/10
- Vertex $d$: finishing time 8/9
- Vertex $e$: finishing time 12
- Vertex $f$: finishing time 3/4
- Vertex $g$: finishing time 2/7
- Vertex $h$: finishing time 5/6
1. Call DFS(G) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS(G) to compute finishing time $f(u)$ for each vertex $u$
1. Call DFS(G) to compute finishing time $f(u)$ for each vertex $u$
2. Compute transpose $G^t$ of $G$ (reverse edge arrows)
3. Call $\text{DFS}(G^t)$, but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
3. Call DFS($G^t$), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
SCC(G) on Example Graph $G$: Part 2

3. Call DFS($G^f$), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
3. Call DFS($G^t$), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
3. Call DFS($G^t$), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
3. Call $\text{DFS}(G^f)$, but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
3. Call DFS($G^t$), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
3. Call DFS($G^i$), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
SCC(G) on Example Graph G: Part 2

4. Output the vertices of each tree in DFS-forest obtained by the second DFS call as a separate SCC: (a, b, e), (c, d), (f, g), (h)
BinSAT: Deciding 2-SAT Without SCCs

- The above algorithm uses SCCs to decide 2-SAT
- All elements in an SCC are equivalent
- Above algorithm requires two “runs”
- Can we avoid the SCC construction and get a faster procedure?
BinSAT: The Algorithm

**Algorithm:** TempPropUnit

**Input:** A literal $x$ to be tentatively assigned.

If $\text{tempval}(x) = false$ /* temporary conflict: $S \models \overline{x} \rightarrow x$/

1. Set $S := \text{PropUnit}(S \cup \{x\})$; **return**;
2. $\text{tempval}(x) := true$; $\text{tempval}(\overline{x}) := false$;

**foreach** $y \overline{x} \in S$ **do**

   **if** $\square \in S$ **or** $\text{permval}(x) \neq NIL$ **then** **return**;

   **if** $\text{permval}(y) = NIL$ **and** $\text{tempval}(y) \neq true$ **then** TempPropUnit(y);

**end**

---

**Algorithm:** BinSAT

**Input:** A set $S$ of binary clauses

**Result:** Unsatisfiable or a model of $S$

**foreach** literal $x$ in $S$ **do** $\text{tempval}(x) := \text{permval}(x) := Nil$;

$S := \text{PropUnit}(S)$;

**while** $\square \not\in S$ **and there exists a literal $x$ with tempval($x$) = permval($x$) = Nil** **do**

   TempPropUnit($x$);

**end**

**if** $\square \in S$ **then** **return** Unsatisfiable;

**else** **return** GetModel();
BinSAT: Some Comments

- **PropUnit**: any implementation of unit resolution which
  - reports forced assignments in permval and
  - generates $\square$ when a global contradiction occurs

- **GetModel**: for each variable $x$, return $permval(x)$ (if $\neq$ Nil) and $tempval(x)$ otherwise

- BinSAT handles **tentative** and **permanent** assignments

- Tentatively assign literal $x$ and propagate consequences by TempPropUnit (=depth-first search applying unit resolution)

- If contradiction occurs, assign $x$ **permanently** and compute consequences (=entailed literals) from this assignment
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \( a \) to true (by TempPropUnit(a)) results in a temp. conflict
- Propagate (by PropUnit(\( S \cup \{ \neg b \} \))
- The resulting permvals are: \( \neg b \) is true and \( \neg a \) is true
- Backtrack (bt) and continue the dfs . . .
- List exhausted \( \Rightarrow \) bt and continue from \( b \)
- Continuing this way yields the final result: permval(\( \neg a \)) = permval(\( \neg b \)) = true, tempval(c) = tempval(d) = tempval(e) = true
BinSAT: Example

\[ S = \{-a \lor b, -b \lor c, -c \lor -b, -c \lor d, -d \lor e, -e \lor c\} \]

- Tentative assignment of \(a\) to true (by TempPropUnit(a)) results in a temp. conflict
- Propagate (by PropUnit(\(S \cup \{¬b\}\))
- The resulting permvals are: \(¬b\) is true and \(¬a\) is true
- Backtrack (bt) and contionue the dfs . . .
- List exhausted \(⇒\) bt and contionue from \(b\)
- Continuing this way yields the final result: \(permval(¬a) = permval(¬b) = true\), \(tempval(c) = tempval(d) = tempval(e) = true\)
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \(a\) to true (by TempPropUnit(a)) results in a temp. conflict
- Propagate (by PropUnit(\(S \cup \{\neg b\}\))
- The resulting permvals are: \(\neg b\) is true and \(\neg a\) is true
- Backtrack (bt) and continue the dfs . . .
- List exhausted \(\Rightarrow\) bt and continue from \(b\)
- Continuing this way yields the final result: \(\text{permval}(\neg a) = \text{permval}(\neg b) = \text{true}, \text{tempval}(c) = \text{tempval}(d) = \text{tempval}(e) = \text{true}\)
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \( a \) to true (by TempPropUnit(a)) results in a temp. conflict
- Propagate (by PropUnit(\( S \cup \{ \neg b \} \))
- The resulting permvals are: \( \neg b \) is true and \( \neg a \) is true
- Backtrack (bt) and continue the dfs . . .
- List exhausted \( \Rightarrow \) bt and continue from \( b \)
- Continuing this way yields the final result: \( \text{permval}(\neg a) = \text{permval}(\neg b) = \text{true}, \text{tempval}(c) = \text{tempval}(d) = \text{tempval}(e) = \text{true} \)
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \( a \) to true (by TempPropUnit(a)) results in a temp. conflict
- Propagate (by PropUnit(\( S \cup \{\neg b\} \))
- The resulting permvals are: \( \neg b \) is true and \( \neg a \) is true
- Backtrack (bt) and continue the dfs . . .
- List exhausted \( \Rightarrow \) bt and continue from \( b \)
- Continuing this way yields the final result: \( \text{permval} (\neg a) = \text{permval} (\neg b) = \text{true} \), \( \text{tempval} (c) = \text{tempval} (d) = \text{tempval} (e) = \text{true} \)
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \( a \) to true (by TempPropUnit(\( a \))) results in a temp. conflict
- Propagate (by PropUnit(\( S \cup \{ \neg b \} \))
- The resulting permvals are: \( \neg b \) is true and \( \neg a \) is true
- Backtrack (bt) and continue the dfs . . .
- List exhausted \( \Rightarrow \) bt and continue from \( b \)
- Continuing this way yields the final result: permval(\( \neg a \)) = permval(\( \neg b \)) = true, tempval(\( c \)) = tempval(\( d \)) = tempval(\( e \)) = true
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \( a \) to true (by TempPropUnit(\( a \))) results in a temp. conflict
- Propagate (by PropUnit(\( S \cup \{ \neg b \} \))
- The resulting permvals are: \( \neg b \) is true and \( \neg a \) is true
- Backtrack (bt) and continue the dfs . . .
- List exhausted \( \Rightarrow \) bt and continue from \( b \)
- Continuing this way yields the final result:
  \( \text{permval}(\neg a) = \text{permval}(\neg b) = \text{true} \), \( \text{tempval}(c) = \text{tempval}(d) = \text{tempval}(e) = \text{true} \)
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \( a \) to true (by TempPropUnit(a)) results in a temp. conflict
- Propagate (by PropUnit(\( S \cup \{ \neg b \} \))
- The resulting permvals are: \( \neg b \) is true and \( \neg a \) is true
- Backtrack (bt) and continue the dfs . . .
- List exhausted \( \Rightarrow \) bt and continue from \( b \)
- Continuing this way yields the final result: permval(\( \neg a \)) = permval(\( \neg b \)) = true, tempval(\( c \)) = tempval(\( d \)) = tempval(\( e \)) = true
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \( a \) to true (by TempPropUnit(a)) results in a temp. conflict
- Propagate (by PropUnit(\( S \cup \{ \neg b \} \))
- The resulting permvals are: \( \neg b \) is true and \( \neg a \) is true
- Backtrack (bt) and continue the dfs . . .
- List exhausted \( \Rightarrow \) bt and continue from \( b \)
- Continuing this way yields the final result: \( \text{permval}(\neg a) = \text{permval}(\neg b) = \text{true} \), \( \text{tempval}(c) = \text{tempval}(d) = \text{tempval}(e) = \text{true} \)
BinSAT: Example

\[ S = \{ \neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c \} \]

- Tentative assignment of \( a \) to true (by TempPropUnit(a)) results in a temp. conflict
- Propagate (by PropUnit(\( S \cup \{ \neg b \} \))
- The resulting permvals are: \( \neg b \) is true and \( \neg a \) is true
- Backtrack (bt) and continue the dfs ...
- List exhausted \( \Rightarrow \) bt and continue from \( b \)
- Continuing this way yields the final result: \( \text{permval}(\neg a) = \text{permval}(\neg b) = \text{true} \), \( \text{tempval}(c) = \text{tempval}(d) = \text{tempval}(e) = \text{true} \)
Horn Satisfiability: The Graph Structure

- **Horn clause**: clause with at most one positive literal
- $p \leftarrow q_1, \ldots, q_n$ stands for $p \lor \neg q_1 \lor \cdots \lor \neg q_n$
- Restriction allows for efficient Horn-SAT algorithms (e.g., based on graphs)

\[
\begin{align*}
p_1 & \leftarrow \\
p_3 & \leftarrow \\
p_6 & \leftarrow \\
p_0 & \leftarrow p_1, p_2, p_3 \\
p_2 & \leftarrow p_1, p_3 \\
p_4 & \leftarrow p_5, p_6 \\
p_5 & \leftarrow p_0
\end{align*}
\]
Horn Satisfiability: The Graph Structure

- **Horn clause**: clause with at most one positive literal
- \( p \leftarrow q_1, \ldots, q_n \) stands for \( p \lor \neg q_1 \lor \cdots \lor \neg q_n \)
- Restriction allows for efficient Horn-SAT algorithms (e.g., based on graphs)

\[
\begin{align*}
p_1 & \leftarrow \\
p_3 & \leftarrow \\
p_6 & \leftarrow \\
p_0 & \leftarrow p_1, p_2, p_3 \\
p_2 & \leftarrow p_1, p_3 \\
p_4 & \leftarrow p_5, p_6 \\
p_5 & \leftarrow p_0 \\
p_0 & \uparrow p_2 \\
p_3 & \leftarrow p_1
\end{align*}
\]
Horn Satisfiability: The Graph Structure

- **Horn clause**: clause with at most one positive literal
- \[ p \leftarrow q_1, \ldots, q_n \] stands for \( p \lor \neg q_1 \lor \cdots \lor \neg q_n \)
- Restriction allows for efficient Horn-SAT algorithms (e.g., based on graphs)

\[
\begin{align*}
p_1 & \leftarrow \\
p_3 & \leftarrow \\
p_6 & \leftarrow \\
p_0 & \leftarrow p_1, p_2, p_3 \\
p_2 & \leftarrow p_1, p_3 \\
p_4 & \leftarrow p_5, p_6 \\
p_5 & \leftarrow p_0
\end{align*}
\]
Horn Satisfiability: The Graph Structure

- **Horn clause**: clause with at most one positive literal
- \( p \leftarrow q_1, \ldots, q_n \) stands for \( p \lor \neg q_1 \lor \cdots \lor \neg q_n \)
- Restriction allows for efficient Horn-SAT algorithms (e.g., based on graphs)

\[ p_1 \leftarrow \]
\[ p_3 \leftarrow \]
\[ p_6 \leftarrow \]
\[ p_0 \leftarrow p_1, p_2, p_3 \]
\[ p_2 \leftarrow p_1, p_3 \]
\[ p_4 \leftarrow p_5, p_6 \]
\[ p_5 \leftarrow p_0 \]
Horn Satisfiability: The Graph Structure

- **Horn clause**: clause with at most one positive literal
  - $p \leftarrow q_1, \ldots, q_n$ stands for $p \lor \neg q_1 \lor \cdots \lor \neg q_n$
  - Restriction allows for efficient Horn-SAT algorithms (e.g., based on graphs)

```
p_1 \leftarrow 
p_3 \leftarrow 
p_6 \leftarrow 
p_0 \leftarrow p_1, p_2, p_3
p_2 \leftarrow p_1, p_3
p_4 \leftarrow p_5, p_6
p_5 \leftarrow p_0
```
Horn Satisfiability: The Graph Structure

- **Horn clause**: clause with at most one positive literal
- $p \leftarrow q_1, \ldots, q_n$ stands for $p \lor \neg q_1 \lor \cdots \lor \neg q_n$
- Restriction allows for efficient Horn-SAT algorithms (e.g., based on graphs)
- Clause set is now represented as a graph

\[
\begin{align*}
p_1 & \leftarrow \\
p_3 & \leftarrow \\
p_6 & \leftarrow \\
p_0 & \leftarrow p_1, p_2, p_3 \\
p_2 & \leftarrow p_1, p_3 \\
p_4 & \leftarrow p_5, p_6 \\
p_5 & \leftarrow p_0
\end{align*}
\]
Horn Satisfiability: The Propagation Algorithm

- All red-marked atoms are true and derivable by punit res (i.e., res, where one parent is a positive unit clause (atom))
- Propagate consequences of true atoms through the graph

\[
\begin{align*}
p_1 & \leftarrow \\
p_3 & \leftarrow \\
p_6 & \leftarrow \\
p_0 & \leftarrow p_1, p_2, p_3 \\
p_2 & \leftarrow p_1, p_3 \\
p_4 & \leftarrow p_5, p_6 \\
p_5 & \leftarrow p_0
\end{align*}
\]
Horn Satisfiability: The Propagation Algorithm

- All red-marked atoms are true and derivable by punit res (i.e., res, where one parent is a positive unit clause (atom))
- Propagate consequences of true atoms through the graph

\[ p_1 \leftarrow \]
\[ p_3 \leftarrow \]
\[ p_6 \leftarrow \]
\[ p_0 \leftarrow p_1, p_2, p_3 \]
\[ p_2 \leftarrow p_1, p_3 \]
\[ p_4 \leftarrow p_5, p_6 \]
\[ p_5 \leftarrow p_0 \]
Horn Satisfiability: The Propagation Algorithm

- All **red-marked atoms** are true and derivable by punit res (i.e., res, where one parent is a positive unit clause (atom))
- Propagate consequences of true atoms through the graph

\[ p_1 \leftarrow \]
\[ p_3 \leftarrow \]
\[ p_6 \leftarrow \]
\[ p_0 \leftarrow p_1, p_2, p_3 \]
\[ p_2 \leftarrow p_1, p_3 \]
\[ p_4 \leftarrow p_5, p_6 \]
\[ p_5 \leftarrow p_0 \]
Horn Satisfiability: The Propagation Algorithm

- All **red-marked atoms** are true and derivable by punit res (i.e., res, where one parent is a positive unit clause (atom))
- Propagate consequences of true atoms through the graph

\[
\begin{align*}
p_1 & \leftarrow \\
p_3 & \leftarrow \\
p_6 & \leftarrow \\
p_0 & \leftarrow p_1, p_2, p_3 \\
p_2 & \leftarrow p_1, p_3 \\
p_4 & \leftarrow p_5, p_6 \\
p_5 & \leftarrow p_0
\end{align*}
\]
Horn Satisfiability: The Propagation Algorithm

- All red-marked atoms are true and derivable by punit res (i.e., res, where one parent is a positive unit clause (atom))
- Propagate consequences of true atoms through the graph

\[ p_1 \leftarrow \]
\[ p_3 \leftarrow \]
\[ p_6 \leftarrow \]
\[ p_0 \leftarrow p_1, p_2, p_3 \]
\[ p_2 \leftarrow p_1, p_3 \]
\[ p_4 \leftarrow p_5, p_6 \]
\[ p_5 \leftarrow p_0 \]
Horn Satisfiability: The Propagation Algorithm

- All red-marked atoms are true and derivable by punit res (i.e., res, where one parent is a positive unit clause (atom))
- Propagate consequences of true atoms through the graph

\[
\begin{align*}
p_1 & \leftarrow \\
p_3 & \leftarrow \\
p_6 & \leftarrow \\
p_0 & \leftarrow p_1, p_2, p_3 \\
p_2 & \leftarrow p_1, p_3 \\
p_4 & \leftarrow p_5, p_6 \\
p_5 & \leftarrow p_0
\end{align*}
\]
Horn Satisfiability: The Propagation Algorithm

- All red-marked atoms are true and derivable by punit res (i.e., res, where one parent is a positive unit clause (atom))
- Propagate consequences of true atoms through the graph

\[ p_1 \leftarrow \]
\[ p_3 \leftarrow \]
\[ p_6 \leftarrow \]
\[ p_0 \leftarrow p_1, p_2, p_3 \]
\[ p_2 \leftarrow p_1, p_3 \]
\[ p_4 \leftarrow p_5, p_6 \]
\[ p_5 \leftarrow p_0 \]
Horn Satisfiability: The Propagation Algorithm

- All red-marked atoms are true and derivable by punit res (i.e., res, where one parent is a positive unit clause (atom))
- Propagate consequences of true atoms through the graph

\[ p_1 \leftarrow \]
\[ p_3 \leftarrow \]
\[ p_6 \leftarrow \]
\[ p_0 \leftarrow p_1, p_2, p_3 \]
\[ p_2 \leftarrow p_1, p_3 \]
\[ p_4 \leftarrow p_5, p_6 \]
\[ p_5 \leftarrow p_0 \]
Horn Satisfiability: Handling Negative Clauses

- So far, all our Horn clauses were non-negative (and the clause set was SAT)
- UNSAT clause sets: at least one purely negative clause
- Clause set is UNSAT if, for a negative clause $q_1, \ldots, q_n$, all $q_i$ are true
A \textit{k-XOR clause}, $C$, is a linear equation over the finite field $GF(2)$ using exactly $k$ distinct variables, i.e.,

$$C = ((x_1 \oplus \ldots \oplus x_k) = \varepsilon) \quad \text{where } \varepsilon = 0 \text{ or } 1$$

A \textit{k-XOR formula (k-XOR clause set)}, $\varphi$, is a conjunction of not necessarily distinct $k$-XOR clauses
Truth Assignments for $k$-XOR Clauses

A truth assignment $I$ is a mapping that assigns 0 or 1 to each variable in its domain

$I$ satisfies an XOR clause $C = ((x_1 \oplus \ldots \oplus x_k) = \varepsilon)$ if and only if

$$I(C) := \left( \sum_{i=1}^{k} I(x_i) \right) \mod 2 = \varepsilon.$$

$I$ satisfies a formula $\varphi$ if and only if it satisfies every clause in $\varphi$

XOR-SAT problem: Given an XOR formula $\varphi$, is there an assignment $I$ which satisfies $\varphi$?
How to Solve XOR-SAT?

Consider $\varphi$ as a system of linear equations and write it as

$$S := (C \mid \vec{e})$$

- $C$ is the coefficient matrix from the lhs of the clauses, $\vec{e}$ is the column vector from the clauses' rhs
- Entries in $S$ are 0 or 1
- Bring $S$ to echelon form (by applying Gaussian elimination) (Coefficient arithmetic is performed in $GF(2)$!)
- Compute the rank of $S$
- Answer NO if rank of $S >$ rank of $C$; answer YES otherwise
- The runtime is in $O(l^2 n)$ ($l$ no of clauses, $n$ no of variables)
Concluding Remarks

- Horn-SAT, 2-SAT, XOR-SAT are easy (polynomial)
- Naive algorithms for Horn-, 2-SAT yield quadratic runtime
- For linear runtime, sophisticated data structures (and some restrictions on the input format) are necessary
- Extensions of the problems lead to NP-complete problems
  - Allow more than one positive literal in Horn clauses
  - Allow more than two literals per clause (yields $k$-CNF)
  - Mix Horn and 2-CNF
  - etc.
The Evolution of SAT Solving Algorithms

- 1952: Quine, \( \approx 10 \) vars
- 1960: DP, \( \approx 10 \) vars
- 1962: DLL, \( \approx 10 \) vars
- 1986: BDDs, \( \approx 100 \) vars
- 1988: SOKRATES, \( \approx 3k \) vars
- 1992: GSAT, \( \approx 300 \) vars
- 1994: HANNIBAL, \( \approx 3k \) vars
- 1996: STALMARCK, \( \approx 1k \) vars
- 1996: GRASP, \( \approx 1k \) vars
- 1996: SATO, \( \approx 1k \) vars
- 2001: Chaff, \( \approx 10k \) vars
- 2002: Berkmin, \( \approx 10k \) vars