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

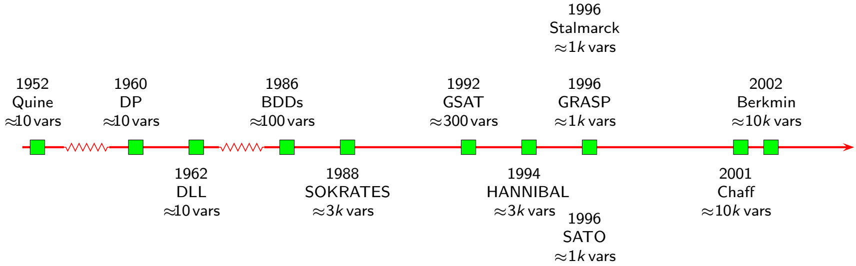
Why is SAT Important?

- Theoretical importance:
    - First NP-Complete problem discovered by S. A. Cook (The complexity of theorem proving procedures, Proc. 3rd ACM Symp. on the Theory of Computing,1971, 151-158)
- 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
≈3*k* vars

1992
GSAT
≈300 vars

1994
HANNIBAL
≈3*k* vars

1996
Stalmarck
≈1*k* vars

1996
GRASP
≈1*k* vars

1996
SATO
≈1*k* vars

2001
Chaff
≈10*k* vars

2002
Berkmin
≈10*k* vars

Transformation of BFs into Normal Forms

- 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 $\wedge$ and $\vee$
  - NNF of ϕ (denoted by $nnf(\phi)$) and ϕ 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 ϕ and translate it to NNF
- Take *nnf*(ϕ) and replace the left side of the following equivalences by the right side (order does not matter!)

$$\begin{array}{lrcl} 1 & \phi \vee (\psi \wedge \chi) & \equiv & (\phi \vee \psi) \wedge (\phi \vee \chi) \\ 2 & (\psi \wedge \chi) \vee \phi & \equiv & (\psi \vee \phi) \wedge (\chi \vee \phi) \end{array}$$

- Observe that *nnf*(ϕ) $\equiv$ *cnf*(ϕ) $\equiv \phi$ holds

Exa: Transform ϕ: $(p \land q \to r) \to (q \to r)$ to CNF

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

Formula	Rule
$(p \land q \land \neg r) \lor (\neg q \lor r)$	2
$(p \lor \neg q \lor r) \land (q \land \neg r \lor (\neg q \lor r))$	2
$(p \lor \neg q \lor r) \land (q \lor \neg q \lor r) \land (\neg r \lor \neg q \lor r)$	

Two disadvantages:

1. Disruption of the formula's structure
2. $cnf(\phi)$ can be exponentially longer than ϕ
   Take a formula in disjunctive NF and translate it to CNF!

Important concept for later: polarities of subformulas

Polarity Labels $(+, -, \pm)$ of Subformulas

$$
\begin{array}{lcll}
(\neg\phi_1)^{+} & \rightsquigarrow & (\neg\phi_1^{-})^{+} & \\
(\neg\phi_1)^{-} & \rightsquigarrow & (\neg\phi_1^{+})^{-} & \\
(\neg\phi_1)^{\pm} & \rightsquigarrow & (\neg\phi_1^{\pm})^{\pm} & \\
(\phi_1 \circ \phi_2)^{q} & \rightsquigarrow & (\phi_1^{q} \circ \phi_2^{q})^{q} & \text{for } q \in \{+,-,\pm\} \text{ and } \circ \in \{\vee, \wedge\} \\
(\phi_1 \rightarrow \phi_2)^{+} & \rightsquigarrow & (\phi_1^{-} \rightarrow \phi_2^{+})^{+} & \\
(\phi_1 \rightarrow \phi_2)^{-} & \rightsquigarrow & (\phi_1^{+} \rightarrow \phi_2^{-})^{-} & \\
(\phi_1 \rightarrow \phi_2)^{\pm} & \rightsquigarrow & (\phi_1^{\pm} \rightarrow \phi_2^{\pm})^{\pm} & \\
(\phi_1 \leftrightarrow \phi_2)^{q} & \rightsquigarrow & (\phi_1^{\pm} \leftrightarrow \phi_2^{\pm})^{q} & \text{for } q \in \{+,-,\pm\}
\end{array}
$$

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

For simplicity, we restrict ourselves to input formulas without ↔

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 ϕ as a tree
- Label each subformula occurrence (SFO) with a new atom
  (atom neither occurs in ϕ nor is it introduced before)
- Construct equivalences of the form

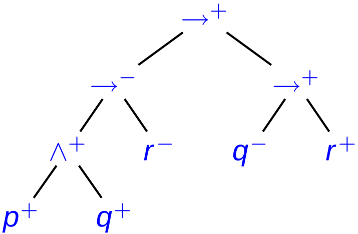
  $$l_\phi \leftrightarrow (l_{\phi_1} \circ l_{\phi_2}) \quad \text{for SFOs} \quad \phi_1 \circ \phi_2$$

  where l_ψ is the label for SF(O) ψ.
- Translate each $l_\phi \leftrightarrow (l_{\phi_1} \circ l_{\phi_2})$ to CNF using the NFT above

Example for a Translation: ϕ: $(p \wedge q \to r) \to (q \to r)$

Step 1: Label the Formula Tree

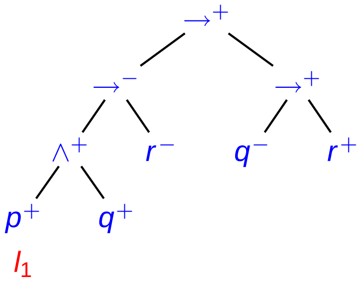
Tree of ϕ^+ with polarities



Example for a Translation: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities

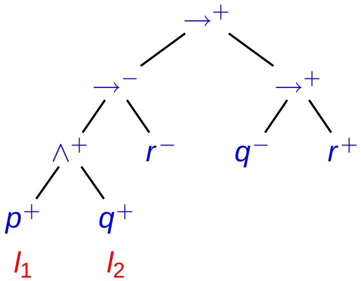


$l_1 \leftrightarrow p$

Example for a Translation: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



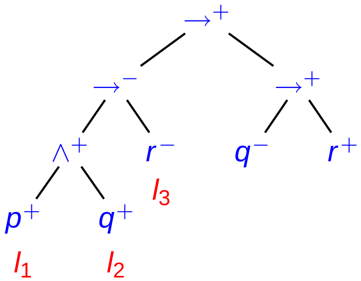
$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

Example for a Translation: ϕ: $(p \land q \to r) \to (q \to r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



$l_1 \leftrightarrow p$

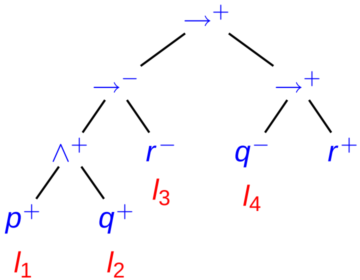
$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

Example for a Translation: ϕ: $(p \wedge q \to r) \to (q \to r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

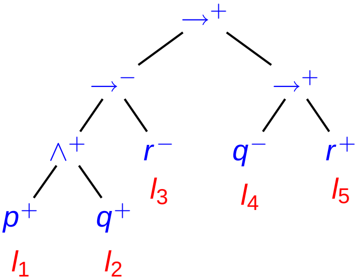
$l_3 \leftrightarrow r$

$l_4 \leftrightarrow q$

Example for a Translation: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

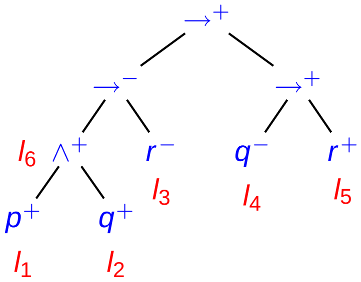
$l_4 \leftrightarrow q$

$l_5 \leftrightarrow r$

Example for a Translation: ϕ: $(p \wedge q \to r) \to (q \to r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

$l_4 \leftrightarrow q$

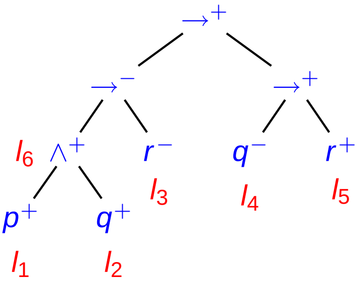
$l_5 \leftrightarrow r$

$l_6 \leftrightarrow p \wedge q$

Example for a Translation: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

$l_4 \leftrightarrow q$

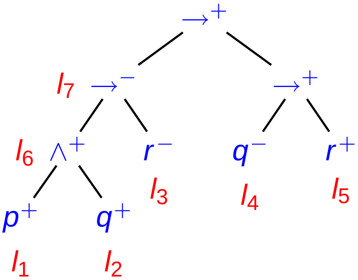
$l_5 \leftrightarrow r$

$l_6 \leftrightarrow l_1 \wedge l_2$

Example for a Translation: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

$l_4 \leftrightarrow q$

$l_5 \leftrightarrow r$

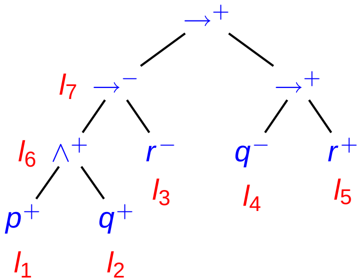
$l_6 \leftrightarrow l_1 \wedge l_2$

$l_7 \leftrightarrow p \wedge q \rightarrow r$

Example for a Translation: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

$l_4 \leftrightarrow q$

$l_5 \leftrightarrow r$

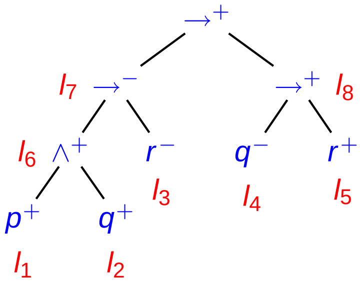
$l_6 \leftrightarrow l_1 \wedge l_2$

$l_7 \leftrightarrow l_6 \rightarrow l_3$

Example for a Translation: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



$l_1 \leftrightarrow p$

$l_2 \leftrightarrow q$

$l_3 \leftrightarrow r$

$l_4 \leftrightarrow q$

$l_5 \leftrightarrow r$

$l_6 \leftrightarrow l_1 \wedge l_2$

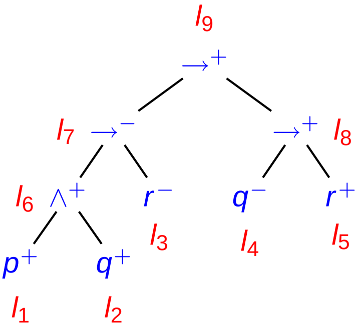
$l_7 \leftrightarrow l_6 \rightarrow l_3$

$l_8 \leftrightarrow l_4 \rightarrow l_5$

Example for a Translation: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 1: Label the Formula Tree

Tree of ϕ^+ with polarities



l_1	$\leftrightarrow$	p
l_2	$\leftrightarrow$	q
l_3	$\leftrightarrow$	r
l_4	$\leftrightarrow$	q
l_5	$\leftrightarrow$	r
l_6	$\leftrightarrow$	$l_1 \wedge l_2$
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: ϕ: $(p \wedge q \rightarrow r) \rightarrow (q \rightarrow r)$

Step 2: Translate the “Labeling Formulas” to Clauses

Equivalences for SFOs in ϕ	Associated Clauses		
	$C_1(\phi)$	$C_2(\phi)$	$C_3(\phi)$
$l_1 \leftrightarrow p$	$\neg l_1 \vee p$	$l_1 \vee \neg p$	
$l_2 \leftrightarrow q$	$\neg l_2 \vee q$	$l_2 \vee \neg q$	
$l_3 \leftrightarrow r$	$\neg l_3 \vee r$	$l_3 \vee \neg r$	
$l_4 \leftrightarrow q$	$\neg l_4 \vee q$	$l_4 \vee \neg q$	
$l_5 \leftrightarrow r$	$\neg l_5 \vee r$	$l_5 \vee \neg r$	
$l_6 \leftrightarrow l_1 \wedge l_2$	$\neg l_6 \vee l_1$	$\neg l_6 \vee l_2$	$l_6 \vee \neg l_1 \vee \neg l_2$
$l_7 \leftrightarrow l_6 \rightarrow l_3$	$\neg l_7 \vee \neg l_6 \vee l_3$	$l_7 \vee l_6$	$l_7 \vee \neg l_3$
$l_8 \leftrightarrow l_4 \rightarrow l_5$	$\neg l_8 \vee \neg l_4 \vee l_5$	$l_8 \vee l_4$	$l_8 \vee \neg l_5$
$l_9 \leftrightarrow l_7 \rightarrow l_8$	$\neg l_9 \vee \neg l_7 \vee l_8$	$l_9 \vee l_7$	$l_9 \vee \neg l_8$

Defining the Translations

Formula ϕ^q	Associated Clauses		
	$C_1(\phi)^q$	$C_2(\phi)^q$	$C_3(\phi)^q$
p^+	$\neg l_p \vee p$		
p^-		$l_p \vee \neg p$	
$(\neg\phi_1)^+$	$\neg l_\phi \vee \neg l_{\phi_1}$		
$(\neg\phi_1)^-$		$l_\phi \vee l_{\phi_1}$	
$(\phi_1 \wedge \phi_2)^+$	$\neg l_\phi \vee l_{\phi_1}$	$\neg l_\phi \vee l_{\phi_2}$	
$(\phi_1 \wedge \phi_2)^-$			$l_\phi \vee \neg l_{\phi_1} \vee \neg l_{\phi_2}$
$(\phi_1 \vee \phi_2)^+$	$\neg l_\phi \vee l_{\phi_1} \vee l_{\phi_2}$		
$(\phi_1 \vee \phi_2)^-$		$l_\phi \vee \neg l_{\phi_1}$	$l_\phi \vee \neg l_{\phi_2}$
$(\phi_1 \rightarrow \phi_2)^+$	$\neg l_\phi \vee \neg l_{\phi_1} \vee l_{\phi_2}$		
$(\phi_1 \rightarrow \phi_2)^-$		$l_\phi \vee l_{\phi_1}$	$l_\phi \vee \neg l_{\phi_2}$

For any ϕ, C_ϕ^q denotes the clauses for ϕ^q ($q \in \{+, -\}$)

The Structure-preserving Normal Forms

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

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

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

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

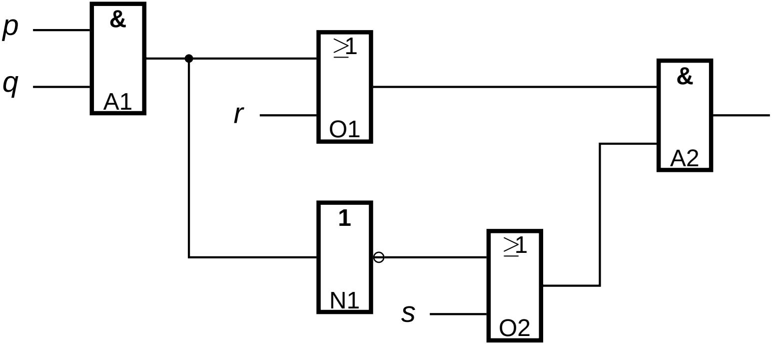
where $\{r\} = \{+, -\} \setminus \{q\}$)

- Sat-equivalence: ϕ 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.

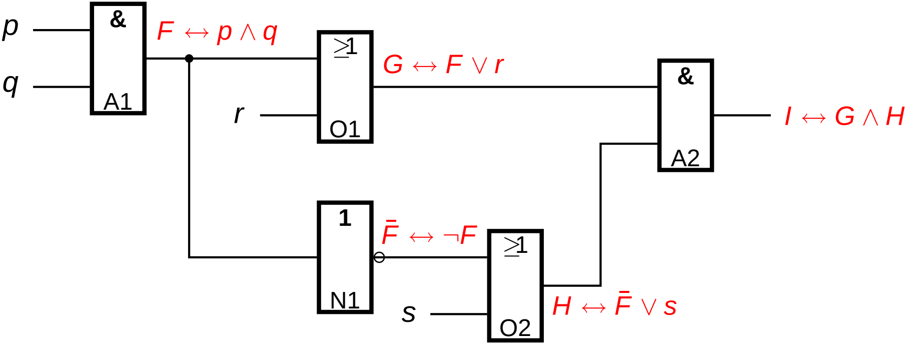
Properties of the Translation

- 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
- ϕ and its definitional translation not logically equivalent
  (new labels change signature $\leadsto$ logical equivalence lost)
- ϕ has a model iff its definitional translation has one holds
  (this is the important prop.; cf 1st order ATP and Skolemization)
- Models of ϕ 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



An Application of the Translation: Circuits to CNF



$F \leftrightarrow p \wedge q$

$G \leftrightarrow F \vee r$

$\bar{F} \leftrightarrow \neg F$

$H \leftrightarrow \bar{F} \vee s$

$I \leftrightarrow G \wedge H$

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)
  - A. del Val. Simplifying Binary Propositional Theories into Connected Components Twice as Fast. In R. Nieuwenhuis and A. Voronkov (Eds.): LPAR 2001, LNAI 2250, pp. 392-406, 2001.

Implications Associated to a 2-CNF

Clause C	Implication(s) $I(C)$	
p	$\neg p \rightarrow p$	
$\neg p$	$p \rightarrow \neg p$	
$p \vee q$	$\neg p \rightarrow q$	$\neg q \rightarrow p$
$p \vee \neg q$	$\neg p \rightarrow \neg q$	$q \rightarrow p$
$\neg p \vee q$	$p \rightarrow q$	$\neg q \rightarrow \neg p$
$\neg p \vee \neg q$	$p \rightarrow \neg q$	$q \rightarrow \neg p$

$u \rightarrow v$ means: "If literal u is true then literal q is also true"

Construction of the Implication Graph (IG)

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

$$\begin{aligned} V_\phi &= \{p, \neg p \mid \text{atom } p \text{ occurs in } \phi\} \\ E_\phi &= \{I(C) \mid C \in \phi\} \end{aligned}$$

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

Construction of the Implication Graph (IG)

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

$$\begin{aligned} V_\phi &= \{p, \neg p \mid \text{atom } p \text{ occurs in } \phi\} \\ E_\phi &= \{I(C) \mid C \in \phi\} \end{aligned}$$

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

p q

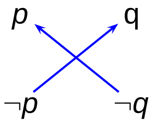
$\neg p$ $\neg q$

Construction of the Implication Graph (IG)

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

$$\begin{aligned} V_\phi &= \{p, \neg p \mid \text{atom } p \text{ occurs in } \phi\} \\ E_\phi &= \{I(C) \mid C \in \phi\} \end{aligned}$$

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

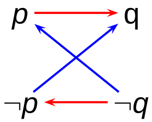


Construction of the Implication Graph (IG)

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

$$\begin{aligned} V_\phi &= \{p, \neg p \mid \text{atom } p \text{ occurs in } \phi\} \\ E_\phi &= \{I(C) \mid C \in \phi\} \end{aligned}$$

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



Next: Construct all strongly connected components of $\mathrm{G}(\phi)$

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 ϕ is SAT iff no p and $\neg p$ belong to the same SCC

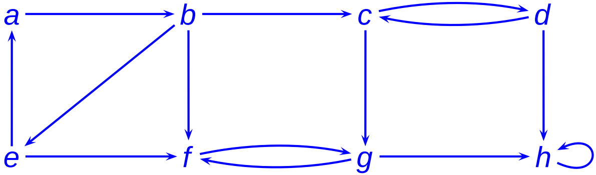
2-SAT is solvable in linear time

SCC(*G*): Compute all SCCs of Directed Graph *G*

Input: Directed graph G
Output: All SCCs of G

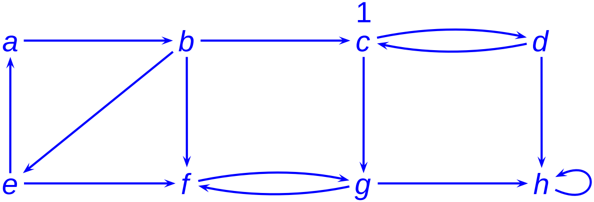
1. Call DFS(G) to compute finishing time $f(u)$ for each vertex u
2. Compute transpose G^t of G (reverse edge arrows)
3. Call DFS(G^t), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)
4. Output the vertices of each tree in DFS-forest obtained by the second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

SCC(G) on Example Graph *G*: Part 1



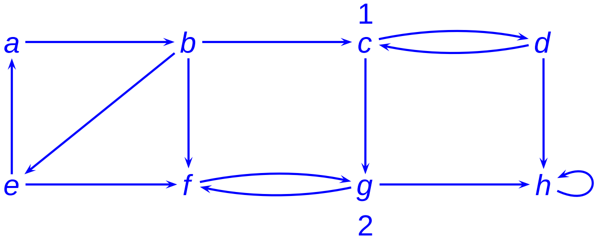
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex u



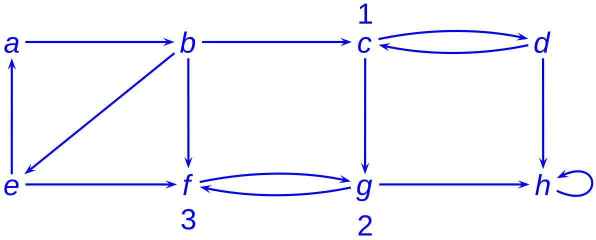
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



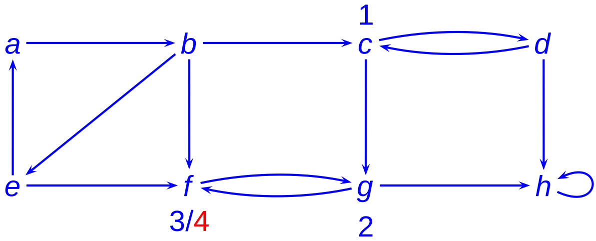
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



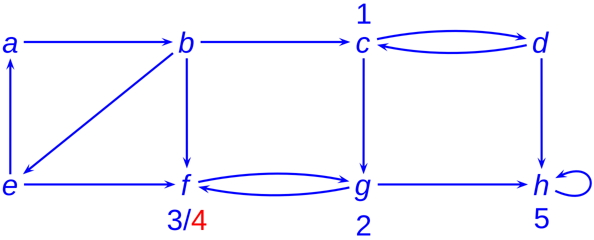
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



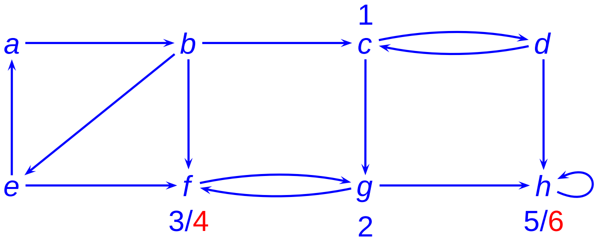
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



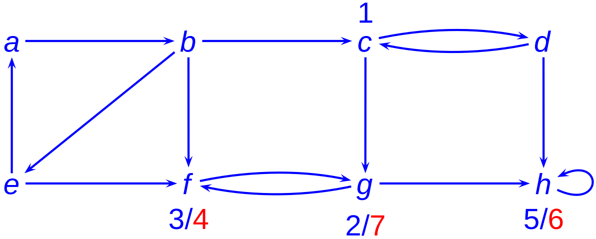
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time *f*(*u*) for each vertex *u*



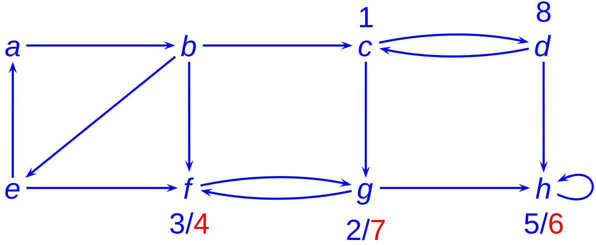
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



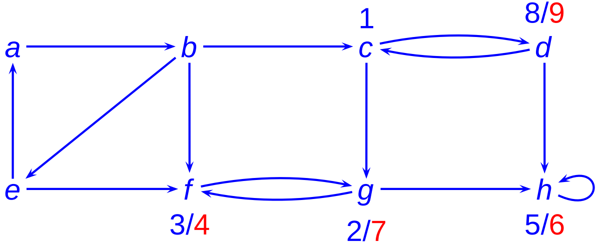
SCC(G) on Example Graph G: Part 1

1. Call DFS(G) to compute finishing time $f(u)$ for each vertex u



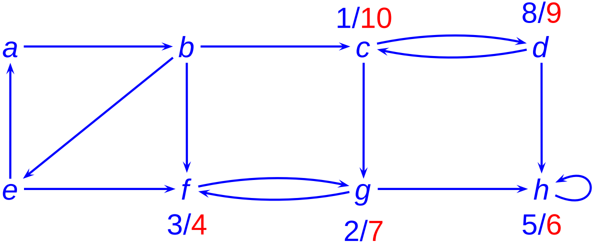
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



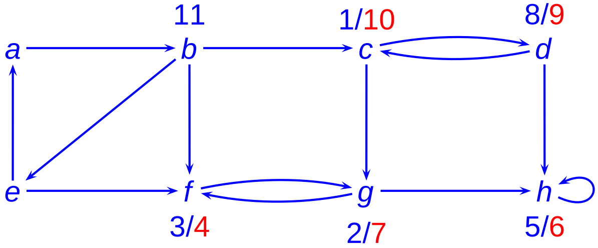
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



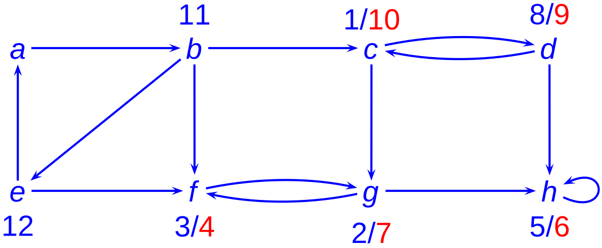
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



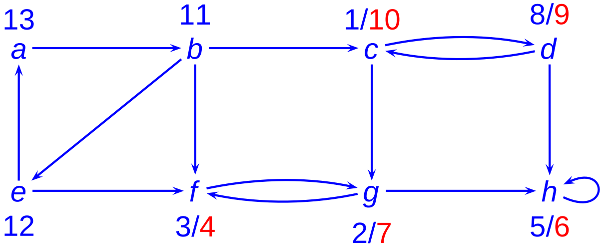
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



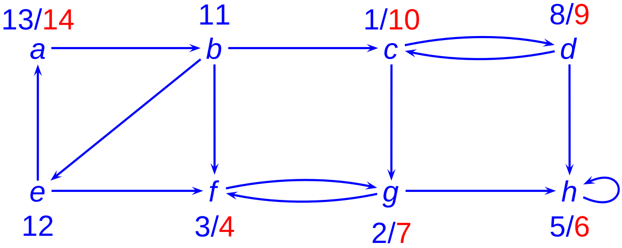
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex u



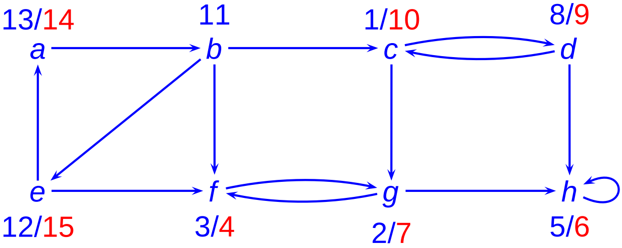
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



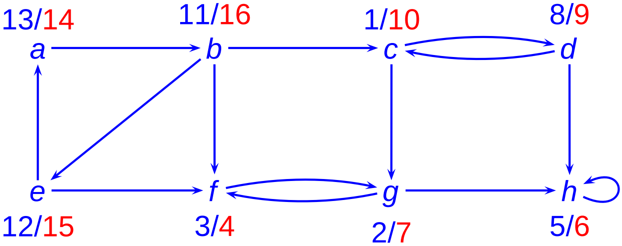
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time $f(u)$ for each vertex *u*



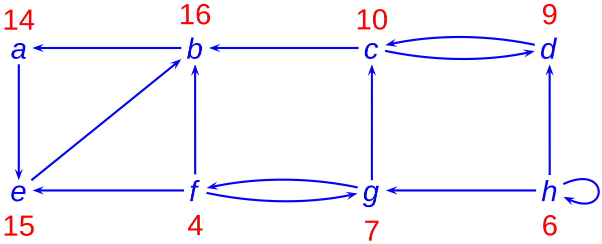
SCC(G) on Example Graph *G*: Part 1

1. Call DFS(*G*) to compute finishing time *f*(*u*) for each vertex *u*



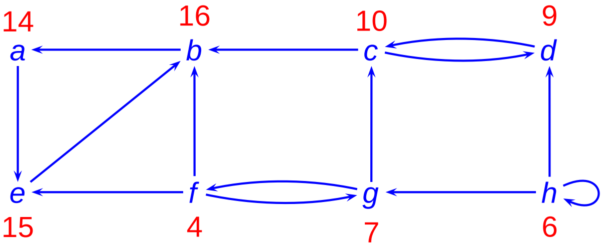
SCC(G) on Example Graph *G*: Part 2

2. Compute transpose G^t of G (reverse edge arrows)



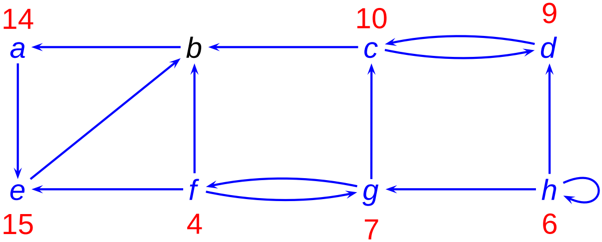
SCC(G) on Example Graph *G*: Part 2

3. Call DFS(G^t), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)



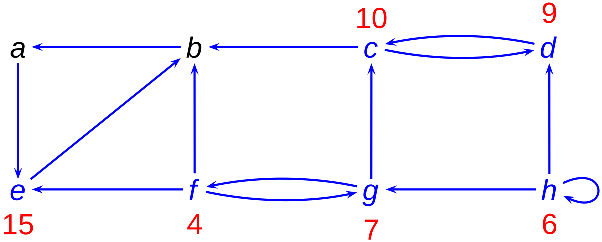
SCC(G) on Example Graph *G*: Part 2

3. Call DFS(G^t), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)



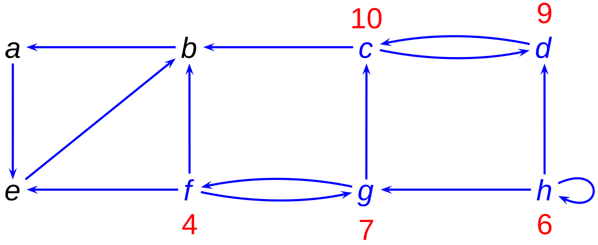
SCC(G) on Example Graph G: Part 2

3. Call DFS(G^t), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)



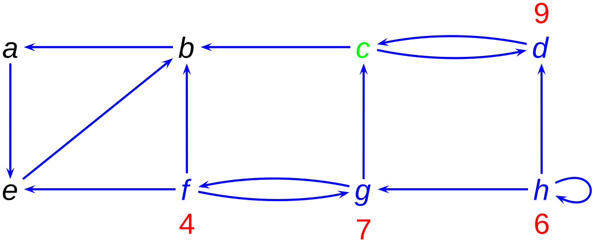
SCC(G) on Example Graph G: Part 2

3. Call DFS(G^t), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)



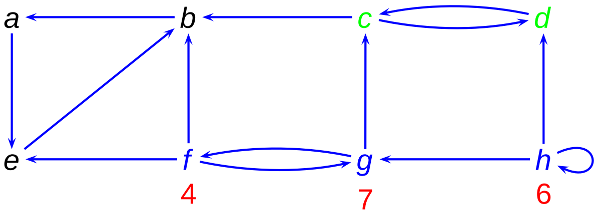
SCC(G) on Example Graph *G*: Part 2

3. Call DFS(G^t), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)



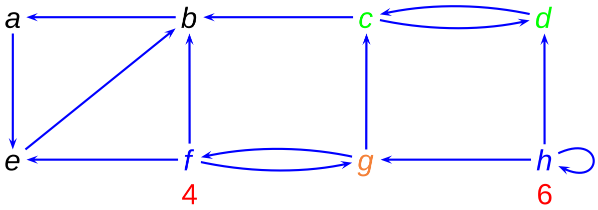
SCC(G) on Example Graph *G*: Part 2

3. Call DFS(G^t), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)



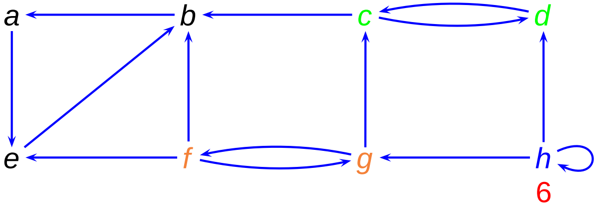
SCC(G) on Example Graph G: Part 2

3. Call DFS(G^t), but in the main loop, consider vertices in order of decreasing $f(u)$ (as computed in first DFS)



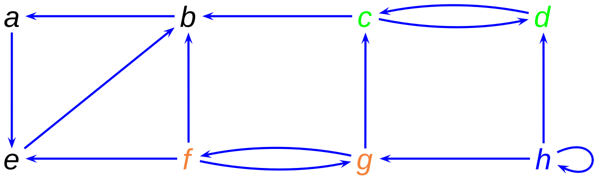
SCC(G) on Example Graph *G*: Part 2

3. Call DFS(G^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

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 tempval(x) = false then /* temporary conflict: $S \models \bar{x} \rightarrow x$ */
    Set $S$ := PropUnit($S \cup \{x\}$); return;
end
tempval($x$) := true; tempval($\bar{x}$) := false;
foreach $y\bar{x} \in S$ do
    if $\square \in S$ or permval(x) $\neq$ NIL then return;
    if permval(y) = NIL and tempval(y) $\neq$ true then TempPropUnit(y);
end
```

Algorithm: BinSAT

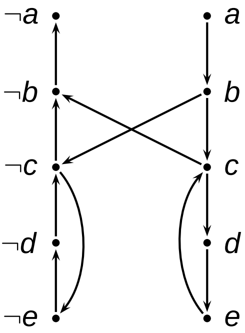
```
Input: A set $S$ of binary clauses
Result: Unsatisfiable or a model of $S$
foreach literal x in S do  tempval(x) := permval(x) := Nil;
$S$ := PropUnit(S);
while $\square \notin S$ and there exists a literal x with tempval(x) = permval(x) = Nil do
    TempPropUnit(x);
end
if $\square \in S$ then  return Unsatisfiable;
else  return GetModel();
```

BinSAT: Some Comments

- PropUnit: any implementation of unit resolution which
  - reports forced assignements in permval and
  - generates $\Box$ 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

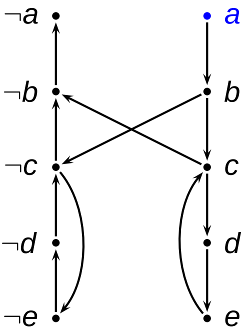
$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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 continoue the dfs . . .
- List exhausted ➥ bt and continoue from *b*
- Continouing this way yields the final result:
  permval($\neg a$) = permval($\neg b$) = true,
  tempval(c) = tempval(d) = tempval(e) = true

BinSAT: Example

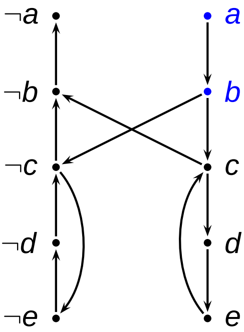
$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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 continoue the dfs . . .
- List exhausted ➥ bt and continoue from *b*
- Continouing this way yields the final result:
  permval($\neg a$) = permval($\neg b$) = true,
  tempval(*c*) = tempval(*d*) = tempval(*e*) = true

BinSAT: Example

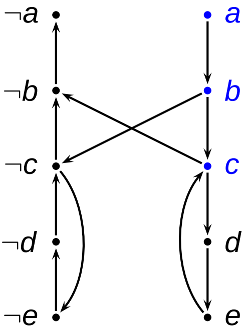
$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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 continoue the dfs . . .
- List exhausted ➥ bt and continoue from b
- Continouing this way yields the final result:
  permval($\neg a$) = permval($\neg b$) = true,
  tempval(c) = tempval(d) = tempval(e) = true

BinSAT: Example

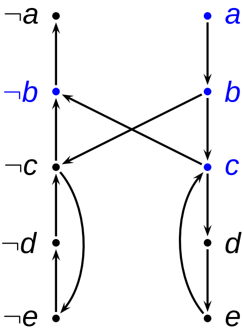
$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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 continoue the dfs …
- List exhausted ➥ bt and continoue from b
- Continouing this way yields the final result:
  permval($\neg a$) = permval($\neg b$) = true,
  tempval(c) = tempval(d) = tempval(e) = true

BinSAT: Example

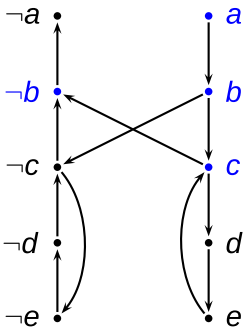
$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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 continoue the dfs . . .
- List exhausted ➥ bt and continoue from b
- Continouing this way yields the final result: permval($\neg a$) = permval($\neg b$) = true, tempval(c) = tempval(d) = tempval(e) = true

BinSAT: Example

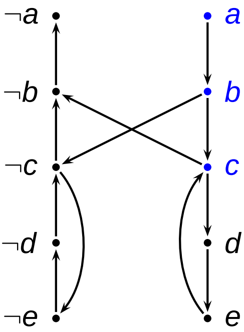
$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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 continoue the dfs …
- List exhausted ➥ bt and continoue from b
- Continouing this way yields the final result: permval($\neg a$) = permval($\neg b$) = true, tempval(c) = tempval(d) = tempval(e) = true

BinSAT: Example

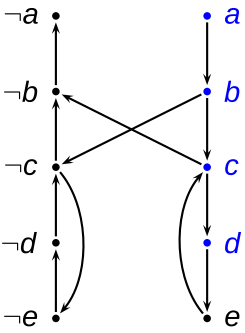
$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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 continoue the dfs . . .
- List exhausted ➥ bt and continoue from b
- Continouing this way yields the final result: permval($\neg a$) = permval($\neg b$) = true, tempval(c) = tempval(d) = tempval(e) = true

BinSAT: Example

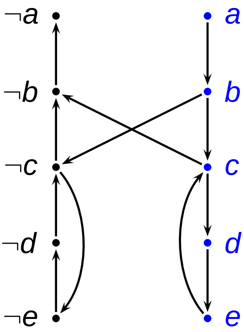
$$S = \{\neg a \lor b, \neg b \lor c, \neg c \lor \neg b, \neg c \lor d, \neg d \lor e, \neg e \lor c\}$$



- 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 continoue the dfs . . .
- List exhausted ➥ bt and continoue from b
- Continouing this way yields the final result:
  permval($\neg a$) = permval($\neg b$) = true,
  tempval(c) = tempval(d) = tempval(e) = true

BinSAT: Example

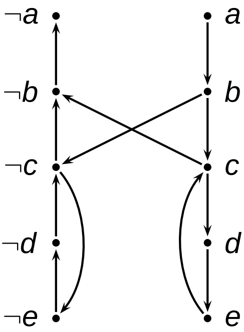
$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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 continoue the dfs …
- List exhausted ➥ bt and continoue from b
- Continouing this way yields the final result: permval($\neg a$) = permval($\neg b$) = true, tempval(c) = tempval(d) = tempval(e) = true

BinSAT: Example

$$S = \{\neg a \vee b, \neg b \vee c, \neg c \vee \neg b, \neg c \vee d, \neg d \vee e, \neg e \vee c\}$$



- 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:
  ¬*b* is true and ¬*a* is true
- Backtrack (bt) and continoue the dfs ...
- List exhausted ➥ bt and continoue from *b*
- Continouing this way yields the final result:
  permval(¬*a*) = permval(¬*b*) = true,
  tempval(*c*) = tempval(*d*) = tempval(*e*) = true

Horn Satisfiability: The Graph Structure

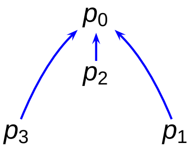
- Horn clause: clause with at most one positive literal
- $p \leftarrow q_1, \ldots, q_n$ stands for $p \vee \neg q_1 \vee \cdots \vee \neg q_n$
- Restriction allows for efficient Horn-SAT algorithms (e.g., based on graphs)

$$
\begin{array}{lll}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}
$$

Horn Satisfiability: The Graph Structure

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

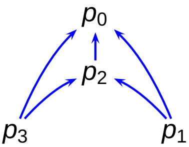
$$
\begin{array}{lll}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}
$$



Horn Satisfiability: The Graph Structure

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

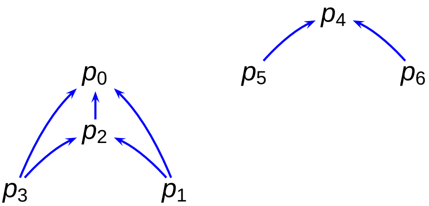
$$
\begin{array}{lll}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}
$$



Horn Satisfiability: The Graph Structure

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

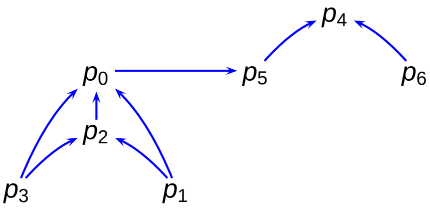
$$\begin{array}{lll}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}$$



Horn Satisfiability: The Graph Structure

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

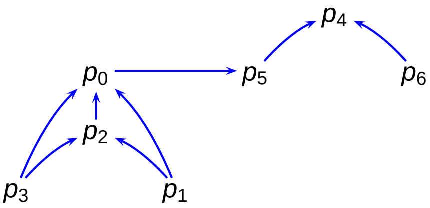
$$
\begin{array}{lll}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}
$$



Horn Satisfiability: The Graph Structure

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

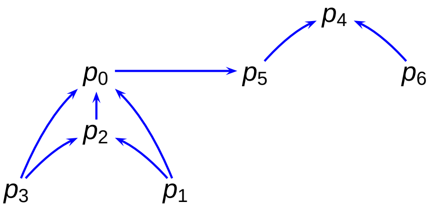
$$\begin{array}{lll} p_1 & \leftarrow & \\ p_3 & \leftarrow & \\ p_6 & \leftarrow & \\ p_0 & \leftarrow & p_1, p_2, p_3 \\ p_2 & \leftarrow & p_1, p_3 \\ p_4 & \leftarrow & p_5, p_6 \\ p_5 & \leftarrow & p_0 \end{array}$$



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

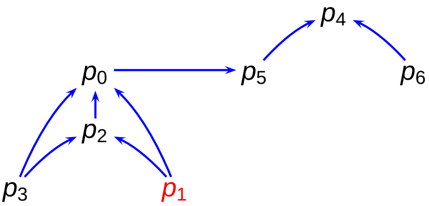
$$\begin{array}{lll} p_1 & \leftarrow & \\ p_3 & \leftarrow & \\ p_6 & \leftarrow & \\ p_0 & \leftarrow & p_1, p_2, p_3 \\ p_2 & \leftarrow & p_1, p_3 \\ p_4 & \leftarrow & p_5, p_6 \\ p_5 & \leftarrow & p_0 \end{array}$$



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

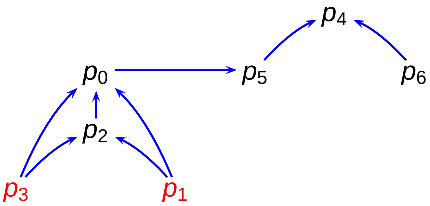
$$\begin{array}{lll} p_1 & \leftarrow & \\ p_3 & \leftarrow & \\ p_6 & \leftarrow & \\ p_0 & \leftarrow & p_1, p_2, p_3 \\ p_2 & \leftarrow & p_1, p_3 \\ p_4 & \leftarrow & p_5, p_6 \\ p_5 & \leftarrow & p_0 \end{array}$$



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

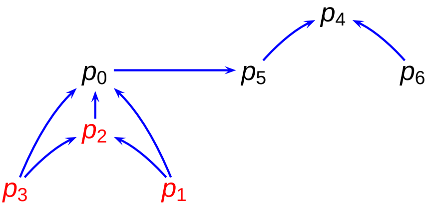
$$
\begin{array}{lll}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}
$$



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

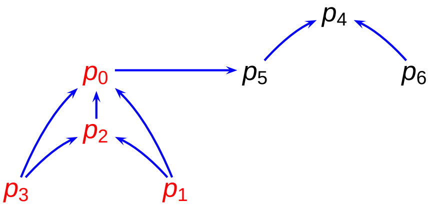
$$\begin{array}{lll} p_1 & \leftarrow & \\ p_3 & \leftarrow & \\ p_6 & \leftarrow & \\ p_0 & \leftarrow & p_1, p_2, p_3 \\ p_2 & \leftarrow & p_1, p_3 \\ p_4 & \leftarrow & p_5, p_6 \\ p_5 & \leftarrow & p_0 \end{array}$$



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

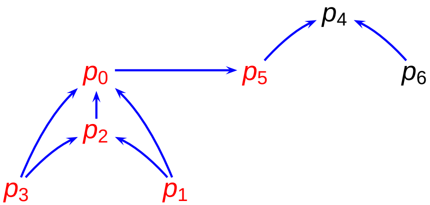
$$\begin{array}{lll}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}$$



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

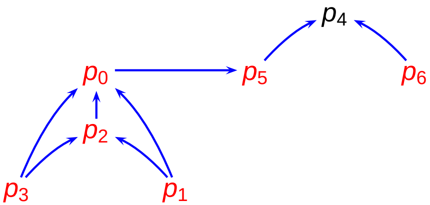
$$\begin{array}{lcl}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}$$



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

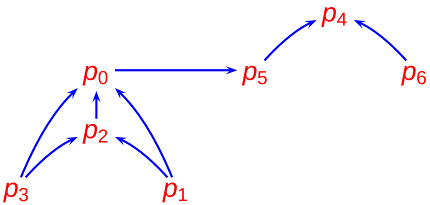
$$\begin{array}{lll} p_1 & \leftarrow & \\ p_3 & \leftarrow & \\ p_6 & \leftarrow & \\ p_0 & \leftarrow & p_1, p_2, p_3 \\ p_2 & \leftarrow & p_1, p_3 \\ p_4 & \leftarrow & p_5, p_6 \\ p_5 & \leftarrow & p_0 \end{array}$$



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

$$\begin{array}{lll}
p_1 & \leftarrow & \\
p_3 & \leftarrow & \\
p_6 & \leftarrow & \\
p_0 & \leftarrow & p_1, p_2, p_3 \\
p_2 & \leftarrow & p_1, p_3 \\
p_4 & \leftarrow & p_5, p_6 \\
p_5 & \leftarrow & p_0
\end{array}$$



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 $\leftarrow q_1, \ldots, q_n$, all q_i are true

The XOR-SAT Problem

k-XOR Clauses and Clause Sets

A *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 *k*-XOR formula (*k*-XOR clause set), φ, 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

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

I satisfies a formula φ if and only if it satisfies every clause in φ

XOR-SAT problem: Given an XOR formula φ, is there an assignment I which satisfies φ?

How to Solve XOR-SAT?

- Consider φ as a system of linear equations and write it as

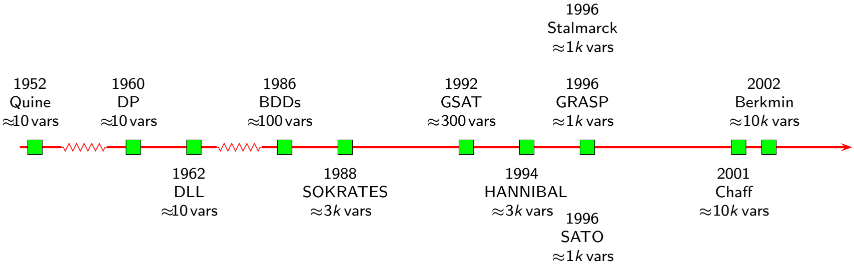
$$\mathbf{S} := (\mathbf{C} \mid \vec{e})$$

- **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