VL Logik (LVA-Nr. 342208)
Winter Semester 2014/2015

Propositional Logic: Evaluating the Formulas
Version 2014.2

Armin Biere (biere@jku.at)
Martina Seidl (martina.seidl@jku.at)
Satisfiability Checking

Definition (Satisfiability Problem of Propositional Logic (SAT))

Given a formula $\phi$, is there an assignment $\nu$ such that $[\phi]_{\nu} = 1$?

- oldest NP-complete problem (see next slides)
  - checking a solution (is an assignment satisfying a formula?) is easy (polynomial effort)
  - finding a solution is difficult (probably exponential in the worst case, what is easy compared to satisfiability checking in other logics)
- many practical applications (used in industry)
- efficient SAT solvers (solving tools) are available
- other problems can be translated to SAT:

<table>
<thead>
<tr>
<th>problem</th>
<th>formulation in propositional logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi$ is valid</td>
<td>$\neg \phi$ is unsatisfiable</td>
</tr>
<tr>
<td>$\phi$ is refutable</td>
<td>to $\neg \phi$ is satisfiable</td>
</tr>
<tr>
<td>$\phi \iff \psi$</td>
<td>to $\neg (\phi \iff \psi)$ is unsatisfiable</td>
</tr>
<tr>
<td>$\phi_1, \ldots, \phi_n \models \psi$</td>
<td>$\phi_1 \land \ldots \land \phi_n \land \neg \psi$ is unsatisfiable</td>
</tr>
</tbody>
</table>
A Glimpse of Complexity Theory

classification of computational hardness of a problem

Turing Machine: machine model for abstract “run time” or “memory usage”
allows more abstract versions of “run time”, “memory usage”
the focus is on worst-case asymptotic time and space usage

Definition

problem is in $O(f(n))$ iff exists constant $c$ and an algorithm which needs $c \cdot f(n)$ steps (in the worst case on a Turing machine) for an input of size $n$

- logarithmic $O(\log n)$, e.g. binary search on sorted array of size $n$
- linear $O(n)$, e.g. linear search in list with $n$ elements
- quadratic $O(n^2)$, e.g. generate list of pairs of $n$ elements
- exponential $O(2^n)$, e.g. produce all subsets of a set of $n$ elements

Definition

polynomial problems: exists fixed $k$ such that worst-case run time is in $O(n^k)$
the class of polynomial problems is called $\mathcal{P}$
SAT and the Complexity Class **NP**

**Definition**
A decision problem asks whether an input belongs to a certain class.

**Prime**: decide whether a number given as input is prime.

**SAT**: decide whether formula given as input is satisfiable.

Basic idea of **NP** is to use a “guess” and “check” approach, where “guessing” is non-deterministic, e.g. just a “good” choice has to exist.

**Definition**
The class NP contains all decision problems which can be decided by a “guessing” and “checking” algorithm in polynomial time in the input size.

Clearly both **Prime** and **SAT** belong to **NP**.

**Theorem (Cook’71)**
Any decision problem in **NP** can be reduced (encoded) polynomially into **SAT**.

Actually, **Prime** can also be solved polynomially (rather complicated).

More on this topic in the “complexity” course.
Complexity Hierarchy

- **P** polynomial time
- **NP** non-deterministic polynomial time
- **PSPACE** polynomial space
- **EXP** exponential time
- **NEXP** non-deterministic exponential time

except for $P \neq \text{EXP}$ and $NP \neq \text{NEXP}$, nothing is known about strict inclusion
One Simple Algorithm for Satisfiability Checking

1 **Algorithm**: evaluate

   **Data**: formula $\phi$
   **Result**: 1 iff $\phi$ is satisfiable

2 if $\phi$ contains a variable $x$ then
   3 pick $v \in \{\top, \bot\}$
   4 /* replace $x$ by truth constant $v$, evaluate resulting formula */
   5 if evaluate($\phi[x|v]$) then return 1;
   6 else return evaluate($\phi[x|\neg v]$);

7 else
   8 switch $\phi$ do
   9     case $\top$ return 1;
   10    case $\bot$ return 0;
   11    case $\neg \psi$ return ! evaluate($\psi$) /* true iff $\psi$ is false */;
   12    case $\psi' \land \psi''$
   13        | return evaluate($\psi'$) && evaluate($\psi''$) /* true iff both $\psi'$ and $\psi''$ are true */
   14    case $\psi' \lor \psi''$
   15        | return evaluate($\psi'$) || evaluate($\psi''$) /* true iff $\psi'$ or $\psi''$ is true */
Reasoning with (Propositional) Calculi

- **goal**: automatically reason about (propositional) formulas, i.e., mechanically show validity/unsatisfiability

- **basic idea**: use syntactical manipulations to prove/refute a formula

- **elements of a calculus**:
  - **axioms**: trivial truths/trivial contradictions
  - **rules**: inference of new formulas

- **approach**: construct a *proof/refutation*, i.e., apply the rules of the calculus until only axioms are inferred. If this is not possible, then the formula is not valid/unsatisfiable.

- **examples of calculi**:
  - sequence calculus: shows validity
  - resolution calculus: shows unsatisfiability
Sequent Calculus: Sequent

Definition

A *sequent* is an expression of the form

\[ \phi_1, \ldots, \phi_n \vdash \psi \]

where \( \phi_1, \ldots, \phi_n, \psi \) are propositional formulas.

The formulas \( \phi_1, \ldots, \phi_n \) are called *assumptions*, \( \psi \) is called *goal*.

*Remarks:*

- **intuitively** \( \phi_1, \ldots, \phi_n \vdash \psi \) means goal \( \psi \) follows from \( \{ \phi_1, \ldots, \phi_n \} \)

- **special case** \( n = 0 \):
  - written as \( \vdash \psi \)
  - meaning: we have to prove that \( \psi \) is valid

- **notation**: for sequent \( \phi_1, \ldots, \phi_n \vdash \psi \), we write \( K \ldots \phi_i \vdash \psi \) if we are only interested in assumption \( \phi_i \)

- the assumptions are *orderless* not ordered
Sequent Calculus: Axiom and Structural Rules

- **axiom "goal in assumption":**
  If the goal is among the assumptions, the goal can be proved.

  \[
  \text{GoalAssum} \quad K \ldots, \psi \vdash \psi
  \]

- **axiom "contradiction in assumptions":**
  If the assumptions are contradicting, anything can be proved.

  \[
  \text{ContrAssum} \quad K \ldots, \phi, \neg \phi \vdash \psi
  \]

- **rule "add valid assumption":**
  If \( \phi \) is valid

  \[
  \text{ValidAssum} \quad K \ldots, \phi \vdash \psi \quad \text{if } \phi \text{ is valid}
  \]
Sequent Calculus: Negation Rules

**rules "contradiction":**

\[ \text{A-}\neg \quad K \ldots \neg \psi \vdash \bot \quad \text{K-} \quad K \ldots \vdash \psi \]

\[ \text{P-}\neg \quad K \ldots \vdash \neg \phi \quad \text{K-} \quad K \ldots , \phi \vdash \bot \]

**rules "elimination of double negation":**

\[ \text{P-}\neg \neg \quad K \ldots \vdash \psi \quad \text{K-} \quad K \ldots \vdash \neg \neg \psi \]

\[ \text{A-}\neg \neg \quad K \ldots , \neg \neg \phi \vdash \psi \quad \text{K-} \quad K \ldots , \phi \vdash \psi \]
Sequent Calculus: Binary Connective Rules

- **rules "conjunction":**

  \[
  \frac{K \ldots, \phi_1, \phi_2 \vdash \psi}{K \ldots, \phi_1 \land \phi_2 \vdash \psi}
  \]

- **rules "disjunction":**

  \[
  \frac{K \ldots, \neg \psi_1 \vdash \psi_2}{K \ldots \vdash \psi_1 \lor \psi_2}
  \]

Rules for other connectives like implication “→” and equivalence “↔” are constructed accordingly.
Some Remarks on Sequent Calculus

- **premises** of a rule: sequent(s) above the line
- **conclusion** of a rule: sequent below the line
- **axiom**: rule without premises
- **non-deterministic rule**: $P \lor$
- **further non-determinism**: decision which rule to apply next
- **rules with case split**: $P \land$, $A \lor$

**proof of formula $\psi$**

1. start with $\vdash \psi$
2. apply rules from bottom to top as long as possible, i.e., for given conclusion, find suitable premise(s)
3. if finally all sequents are axioms then $\psi$ is valid

- note: there are many variants of the sequent calculus
One Algorithm for Calculating with Sequent Calculus

1. Algorithm: entails
   
   **Data:** set of assumptions $A$, formula $\psi$
   
   **Result:** 1 iff $A$ entails $\psi$, i.e., $A \models \psi$

2. if $\psi = \neg\neg\psi'$ then return $\text{entails} (A, \psi')$;
3. if $\neg\neg\phi \in A$ then return $\text{entails} (A \setminus \{\neg\neg\phi\} \cup \{\phi\}, \psi)$;
4. if $\phi_1 \land \phi_2 \in A$ then return $\text{entails} (A \setminus \{\phi_1 \land \phi_2\} \cup \{\phi_1, \phi_2\}, \psi)$;
5. if ($\psi \in A$) or ($\phi, \neg\phi \in A$) then return 1;
6. if $A \cup \{\psi\}$ contains only literals then return 0;
7. switch $\psi$ do
8.     case $\bot$
9.         if $\neg\phi \in A$ then return $\text{entails} (A \setminus \{\neg\phi\}, \phi)$;
10.        if $\phi_1 \lor \phi_2 \in A$ then
11.            if $! \text{entails} (A \setminus \{\neg\phi_1 \lor \phi_2\} \cup \{\phi_1\}, \bot)$ then return 0;
12.            else return $\text{entails} (A \setminus \{\neg\phi_1 \lor \phi_2\} \cup \{\phi_2\}, \bot)$;
13.     case $x$ where $x$ is a variable return $\text{entails} (A \cup \{\neg\phi\}, \bot)$;
14.     case $\neg\psi'$ return $\text{entails} (A \cup \{\psi'\}, \bot)$;
15.     case $\psi_1 \lor \psi_2$ return $\text{entails} (A \cup \{\neg\psi_1\}, \psi_2)$;
16.     case $\psi_1 \land \psi_2$ return $\text{entails} (A, \psi_1) \& \& \text{entails} (A, \psi_2)$;
Proving XOR stronger than OR with the Sequent Calculus

Proof direction:

GoalAssum:

\[
\begin{align*}
\text{A-\lor} & \quad b, (\neg a \lor \neg b), \neg a \vdash b \\
\text{P-\lor} & \quad (a \lor b), (\neg a \lor \neg b), \neg a \vdash b \\
\text{A-\land} & \quad (a \lor b), (\neg a \lor \neg b) \vdash a \land b \\
\text{A-\neg} & \quad (a \lor b) \land (\neg a \lor \neg b) \vdash a \land b \\
\text{P-\lor} & \quad \neg((a \lor b) \land (\neg a \lor \neg b)) \vdash a \lor b \\
\end{align*}
\]
Refuting XOR stronger than AND with the Sequent Calculus

\[
\begin{align*}
\text{GAss} & & b, \neg b \vdash a & & b, \neg a \vdash a \\
\text{A-\lor} & & b, (\neg a \lor \neg b) \vdash a \\
\text{P-\land} & & (a \lor b), (\neg a \lor \neg b) \vdash a
\end{align*}
\]

\[
\begin{align*}
\text{CAss} & & b, \neg b \vdash a & & b, \neg a \vdash a \\
\text{A-\lor} & & b, (\neg a \lor \neg b) \vdash a \\
\text{P-\land} & & (a \lor b), (\neg a \lor \neg b) \vdash a
\end{align*}
\]

\[
\begin{align*}
\text{A-\land} & & (a \lor b), (\neg a \lor \neg b) \vdash a \land b \\
\text{A-\neg d} & & (a \lor b) \land (\neg a \lor \neg b) \vdash a \land b \\
\text{P-\lor} & & \neg ((a \lor b) \land (\neg a \lor \neg b)) \vdash a \land b \\
\end{align*}
\]

\[
\begin{align*}
\text{A-\lor} & & (a \lor b), (\neg a \lor \neg b) \vdash a \lor b \\
\text{A-\land} & & (a \lor b), (\neg a \lor \neg b) \vdash a \land b \\
\text{A-\neg d} & & (a \lor b) \land (\neg a \lor \neg b) \vdash a \land b \\
\text{P-\lor} & & \neg ((a \lor b) \land (\neg a \lor \neg b)) \lor (a \land b)
\end{align*}
\]

counter example to validness: \[ a = \bot, b = \top \]
Soundness and Completeness

For any calculus important properties are, first soundness, i.e. the question “Can only valid formulas be shown as valid?” and second completeness, i.e. the question ”Is there a proof for every valid formula?”.

Soundness

If a formula is shown to be valid in the Gentzen Calculus, then it is valid.

*Proof sketch:* consider each rule individually and show that from valid premises only valid conclusions can be drawn.

Completeness

Every valid formula can be proven to be valid in the Gentzen Calculus.

*Proof sketch:* Show that the algorithm terminates and that there is at least one case where it returns false if the formula is not valid.
In practice, formulas of arbitrary structure are quite challenging to handle.
- tree structure
- simplifications affect only subtrees

We have seen that CNF and DNF are able to represent every formula.
- so why not use them as input for SAT?

**Conjunctive Normal Form**
- refutability is easy to show
- CNF can be efficiently calculated (polynomial)

**Disjunctive Normal Form**
- satisfiability is easy to show
- complexity is in getting the DNF

CNF and DNF can be obtained from the *truth tables*.
- exponential many assignments have to be considered

**alternative approach**
- *structural rewritings* which are (satisfiability) equivalence preserving
Transformation to Normal Form

1. Remove $\leftrightarrow$, $\rightarrow$, $\oplus$ as follows:
   $\phi \leftrightarrow \psi \Leftrightarrow (\phi \rightarrow \psi) \land (\psi \rightarrow \phi)$, $\phi \rightarrow \psi \Leftrightarrow \neg \phi \lor \psi$, $\phi \oplus \psi \Leftrightarrow (\phi \lor \psi) \land (\neg \phi \lor \neg \psi)$

2. Transform formula to negation normal form (NNF) by application of laws of De Morgan and elimination of double negation

3. Transform formula to CNF (DNF) by laws of distributivity

Example

Transform $\neg (a \leftrightarrow b) \rightarrow \neg (c \land d) \land e$ to an equivalent formula in CNF.

1. a) remove equivalences: $\Leftrightarrow \neg ( (a \rightarrow b) \land (b \rightarrow a) ) \rightarrow \neg ( c \land d ) \land e$
   
   b) remove implications: $\Leftrightarrow \neg \neg ( (\neg a \lor b) \land (\neg b \lor a) ) \lor \neg ( c \land d ) \land e$

2. NNF: $\Leftrightarrow ( (\neg a \lor b) \land (\neg b \lor a) ) \lor ( (\neg c \lor \neg d) \land e )$

3. $\Leftrightarrow ( (\neg a \lor b) \lor ( (\neg c \lor \neg d) \land e ) ) \land ( (\neg b \lor a) \lor ( (\neg c \lor \neg d) \land e ) )$
   $\Leftrightarrow ( \neg a \lor b \lor \neg c \lor \neg d ) \land ( \neg a \lor b \lor e ) \land ( \neg b \lor a \lor \neg c \lor \neg d ) \land ( \neg b \lor a \lor e )$
Some Remarks on Normal Forms

- The presented transformation to CNF/DNF is exponential in the worst case (e.g., transform \((a_1 \land b_1) \lor (a_2 \land b_2) \lor \cdots \lor (a_n \land b_n)\) to CNF).

- For DNF transformation, there is probably no better algorithm.

- For CNF transformation, there are polynomial algorithms.
  - Basic idea: introduce labels for subformulas.
  - Also works for formulas with sharing (circuits).
  - Also known as “Tseitin Encoding”.

- CNF is usually not easier to solve, but easier to handle:
  - compact data structures: a CNF is simply a list of lists of literals.

- CNF very popular in practice: standard input format DIMACS

- For solving satisfiability of CNF formulas, there are many optimization techniques as well as dedicated algorithms.
Resolution

- the *resolution calculus* consists of the single resolution rule

\[
\frac{x \lor C \quad \neg x \lor D}{C \lor D}
\]

- \(C\) and \(D\) are (possibly empty) clauses
- the clause \(C \lor D\) is called *resolvent*
- variable \(x\) is called *pivot*
- usually antecedent clauses \(x \lor C\) and \(\neg x \lor D\) are assumed not to be tautological, that is \(x \not\in C\) and \(x \not\in D\).

- in other words:
  \((\neg x \rightarrow C), (x \rightarrow D) \models C \lor D\)

- resolution is *sound* and *complete*.
- the resolution calculus works only on formulas in CNF
- if the empty clause can be derived then the formula is *unsatisfiable*
- if no new clause can be generated by application of the resolution rule then the formula is *satisfiable*
Resolution Example

We *prove unsatisfiability* of

\[\{ (\neg x_1 \vee \neg x_5), (x_4 \vee x_5), (x_2 \vee \neg x_4), (x_3 \vee \neg x_4), (\neg x_2 \vee \neg x_3), (x_1 \vee x_4 \vee \neg x_6), (x_6) \}\]

as follows:
DPLL Overview

The DPLL algorithm is ...

- old (invented 1962)
- easy (basic pseudo-code is less than 10 lines)
- popular (well investigated; also theoretical properties)
- usually realized for formulas in CNF
- using binary constraint propagation (BCP)
- in its modern form as conflict drive clause learning (CDCL) basis for state-of-the-art SAT solvers
Binary Constraint Propagation (BCP)

Definition (Binary Constraint Propagation (BCP))

Let $\phi$ be a formula in CNF containing a unit clause $C$, i.e., $\phi$ has a clause $C = (l)$ which consists only of literal $l$. Then $BCP(\phi, l)$ is obtained from $\phi$ by

- removing all clauses with $l$
- removing all occurrences of $\overline{l}$

- one application of $BCP$ can trigger other applications of $BCP$
- $BCP(\phi)$ denotes all possible applications of $BCP(\phi, l)$ until fixpoint
- if $BCP(\phi)$ produces the empty clause, then the formula $\phi$ is unsatisfiable
- if $BCP(\phi)$ produces the empty CNF, then the formula $\phi$ is satisfiable

Example

$\phi = \{ (\neg a \lor b \lor \neg c), (a \lor b), (\neg a \lor \neg b), (a) \}$

1. $\phi' = BCP(\phi, a) = \{ (b \lor \neg c), (\neg b) \}$
2. $\phi'' = BCP(\phi', \neg b) = \{ (\neg c) \}$
3. $\phi''' = BCP(\phi', c) = \{ \} = \top$
DPLL Algorithm

1 **Algorithm:** evaluate
   
   **Data:** formula \( \phi \) in CNF
   
   **Result:** 1 iff \( \phi \) satisfiable

2 while 1 do
3     \( \phi = \text{BCP}(\phi) \)
4     if \( \phi == \top \) then return 1;
5     if \( \phi == \bot \) then
6         if stack.isEmpty() then return 0;
7         \((l, \phi) = \text{stack.pop}()\)
8         \( \phi = \phi \land l \)
9     else
10        select literal \( l \) occurring in \( \phi \)
11        stack.push(\( \overline{l}, \phi \))
12        \( \phi = \phi \land l \)
Some Remarks on DPLL

- DPLL is the basis for most state-of-the-art SAT solvers
  - like Lingeling  [http://fmv.jku.at/lingeling](http://fmv.jku.at/lingeling)
  - simpler or more established solvers: MiniSAT, PicoSAT, Cleaneling, ...

- DPLL alone is not enough - powerful optimizations required for efficiency:
  - learning and non-chronological back-tracking (CDCL)
  - reset strategies and phase-saving
  - compact lazy data-structures
  - variable selection heuristics
  - usually combined with preprocessing before search
  - and inprocessing algorithms interleaved with search

- variants of DPLL are also used for other logics:
  - quantified propositional logic (QBF)
  - satisfiability modulo theories (SMT)

- challenge to parallelize
  - some succesfull attempts: ManySAT, Plingeling, Penelope, Treengeling, ...