Satisfiability, Boolean Modeling and Computation
Spring 2016

Matti Järvisalo

Lecture 1: Practical Arrangements,
Introduction and Basic Concepts
January 19, 2016
On This Lecture

- Course information
- Motivation: what, why?
- Course outline
- Basics/refresher on propositional logic
Course Information

Lectures: Tuesdays and Thursdays 12–14
during Jan 19 – Mar 1 (≈ 10 lectures)
  ▶ No lectures during Feb 11–18
  ▶ March 3 on reserve

Lecturer: Dr. Matti Järvisalo
  matti.jarvisalo@cs.helsinki.fi

Reception: During lectures / contact lecturer by email
for an appointment

Tutorials: Tuesdays 10–12
during Jan 26 – Mar 1 (6 sessions)

Assistant: M.Sc. Jeremias Berg,
  jeremias.berg@cs.helsinki.fi

Course code: 582742

Credit units: 5 ECTS

WWW:
  https://www.cs.helsinki.fi/en/courses/582742/2016/k/k/1
Course Requirements and Grading

- Final exam: Mar 9 at 9 AM
- Active participation in lectures
- Active participation in tutorials
- Weekly hand-in exercises, must obtain $> 50\%$ points
- Final grade: final exam grade

Active participation in lectures and tutorials is recommended

Final exam covers lectures, lecture slides, and tutorials
Weekly hand-in exercises

- Solve a set of weekly assignments (2–3 per week)
- Tutorial sheet with assignments made available by Thursday the week before
- Return to course assistant at the tutorials
- Deadline: beginning of the weekly tutorial session
- Each solution is graded on the scale 0-2
- To pass the course, you must obtain at least 50 % of the total number of available points

Tutorials consist of

- discussing solutions the weekly home assignments (if you want)
- additional tutorial exercises (similar to problems in final exam)
Prerequisites

Background:

- Design and Analysis of Algorithms (algorithmic thinking)
- Introduction to Artificial Intelligence (search)
- Models of Computation (complexity of problems)
- Basics of propositional logic

Either you have the necessary formal background, or are willing to do the extra work to figure things out yourself.
Materials and Further Reading

- Course materials = lecture slides + tutorials
- Since the lecture slides are new, there will likely be mistakes in the slides presented at lectures
- A revised slide package, including all lectures, will be made available after the last lecture
- *Not based on any single book*
- Possible further reading
  - See “Further Reading” list on the course webpage
- *Not necessary to purchase a book*

Acknowledgements

Lectures at times rely on material from my collaborators, including Fahiem Bacchus, Anton Belov, Armin Biere, Marijn Heule, Joao Marques-Silva, and others. Many thanks!
Materials and Further Reading

- Course materials = lecture slides + tutorials
- Since the lecture slides are new, there will likely be mistakes in the slides presented at lectures
- A revised slide package, including all lectures, will be made available after the last lecture
- *Not based on any single book*
- Possible further reading
  - See “Further Reading” list on the course webpage
- *Not necessary to purchase a book*

Acknowledgements

Lectures at times rely on material from my collaborators, including Fahiem Bacchus, Anton Belov, Armin Biere, Marijn Heule, Joao Marques-Silva, and others. Many thanks!
What This Course is about

- A computer science perspective to solving hard problems
- Focus on *practical* algorithmic methods
- Based on automated logical reasoning
  - Efficient implementations will allow you to solve *real* problem instances
- Focusing on the declarative approach
  - Constraint satisfaction and optimization
  - Linear and integer programming
  - Modelling and solving
Motivation
Combinatorial Problems in the Real World

- Automated planning
  - Planning for executing tasks
    (e.g., optimizing packet delivery services)
- Scheduling
  - Given N tasks with earliest start times, completion deadlines, and set of M machines on which they can execute, schedule them so that they all finish by their deadlines
- Model checking
  - Does a (formal model of a) hardware or software design satisfy a formal specification (e.g., “an operating system driver will never cause the computer to deadlock”)

Interesting problems are often characterized by computational intractability — \textbf{NP}-complete or even harder

\textit{no general polynomial algorithms known}
Algorithm Design Techniques

- There are several approaches to developing specialized efficient algorithms for computationally difficult problems
  - Specialized exact algorithms with provably good *(but exponential)* worst-case behavior
  - Fast (polynomial-time) approximation algorithms
  - Randomized algorithms

- All of the above require expertise in developing algorithms for the *specific problem* at hand
  - Changing the problem even in a small way may require one to develop a new specialized algorithm from scratch

Focus on this course: the *declarative programming* approach

*Modelling + search*

- *Generic* approaches for attacking hard computational problems
Declarative Programming

Two-step approach to solving hard combinatorial problems:

1. **Encoding**: *Domain-specific* declarative formulation of problem using chosen *(constraint) modelling language*
   - Given any problem instance, formulate the instance in terms of *mathematical constraints*

2. **Solving**: A *generic* solver—a search algorithm—for the chosen modelling language, which can find a *solution* (or determine that none exist) to any formulation in the modelling language
   - Found solution mapped back to a solution of the original problem instance

Various approaches based on *different modelling languages*

integer programming, linear programming, constraint programming, *Boolean satisfiability*
Boolean Satisfiability

- In general, SAT the question of whether a given *propositional logic formula is satisfiable*
- Typically SAT refers to CNF SAT
  - The satisfiability problem of propositional (Boolean) formulas in *conjunctive normal form*, CNF formulas
- Very simple, low-level modelling language

Constraint language that provides a highly efficient approach to solving various hard computational problems
Speaker Dress Code as SAT

- Variables: tie, shirt
- Three conditions / clauses:
  - clearly one should not wear a tie without a shirt
    \[ \neg\text{tie} \lor \text{shirt} \]
  - not wearing a tie nor a shirt is impolite
    \[ \text{tie} \lor \text{shirt} \]
  - wearing a tie and a shirt is overkill
    \[ \neg(\text{tie} \land \text{shirt}) \equiv \neg\text{tie} \lor \neg\text{shirt} \]
- Is the formula
  \[ (\neg\text{tie} \lor \text{shirt}) \land (\text{tie} \lor \text{shirt}) \land (\neg\text{tie} \lor \neg\text{shirt}) \]
satisfiable?
CNF Formulas: Syntax

- **Literal**: a Boolean variable $x$, or the *negation* $\neg x$ of $x$
  - $\neg x$ is the *negative literal* of $x$, $x$ the positive literal

- **Clause**: $l_1 \lor \cdots \lor l_k$, where each $l_i$ is a literal
  - $\lor$ is called *disjunction*, i.e., logical OR
  - Short-hand: $\lor_{i=1}^{k} l_i$
  - $k$: *length* of the clause
    - $k = 1$: unit clause
    - $k = 2$: binary clause

- **CNF formula**: $C_1 \land \cdots \land C_m$, where each $C_i$ is a clause
  - $\land$ is called *conjunction*, i.e., logical AND
  - Short-hand: $\land_{i=1}^{m} C_m$
  - Often viewed as a set of clauses
A truth assignment $\tau$ maps Boolean variables to $\{0, 1\}$
- 0: false, 1: true

$\tau$ satisfies a literal $l$, $\tau(l) = 1$, iff
- $l$ is a positive literal $x$ and $\tau(x) = 1$, or
- $l$ is a negative literal $\neg x$ and $\tau(x) = 0$

Satisfying a clause $C = l_1 \lor \cdots \lor l_k$:
$\tau(C) = 1$ iff $\tau(l_i) = 1$ for some $i = 1..k$.

Satisfying a CNF formula $F = C_1 \land \cdots \land C_m$:
$\tau(F) = 1$ iff $\tau(C_i) = 1$ for each $i = 1..m$.

The Boolean Satisfiability (SAT) problem

Input: A CNF formula $F$.

Question: Is $F$ satisfiable?

Model of $F$: an assignment that satisfies $F$
Example: 3-Coloring as SAT

- INSTANCE: A graph \( G = (V, E) \).
- QUESTION: Is \( G \) 3-colorable?
- Encoding 3-COLORING as SAT:

Clauses for each node \( v \in V \):
\[
\begin{align*}
&v_r \lor v_g \lor v_b \\
&\neg v_r \lor \neg v_g \\
&\neg v_r \lor \neg v_b \\
&\neg v_g \lor \neg v_b
\end{align*}
\]

Clauses for each edge \( (v, u) \in E \):
\[
\begin{align*}
&\neg v_r \lor \neg u_r \\
&\neg v_g \lor \neg u_g \\
&\neg v_b \lor \neg u_b
\end{align*}
\]

Polynomial-size encoding of the NP-complete 3-Coloring problem

- Given any graph \( G \):
  1. Generate the clauses
  2. Input to a SAT solver
- SAT solver answers
  - “No”: we have proven that \( G \) cannot be 3-colored
  - “Yes”: a coloring can be read from the satisfying truth assignment output by the solver.
Example: 3-Coloring as SAT

- INSTANCE: A graph $G = (V, E)$.
- QUESTION: Is $G$ 3-colorable?
- Encoding 3-COLORING as SAT:

  **Clauses for each node $v \in V$:**
  \[
  v_r \lor v_g \lor v_b \\
  \neg v_r \lor \neg v_g \\
  \neg v_r \lor \neg v_b \\
  \neg v_g \lor \neg v_b
  \]

  **Clauses for each edge $(v, u) \in E$:**
  \[
  \neg v_r \lor \neg u_r \\
  \neg v_g \lor \neg u_g \\
  \neg v_b \lor \neg u_b
  \]

Polynomial-size encoding of the NP-complete 3-Coloring problem

- Given any graph $G$:
  1. Generate the clauses
  2. Input to a SAT solver

- SAT solver answers
  - “No”: we have proven that $G$ cannot be 3-colored
  - “Yes”: a coloring can be read from the satisfying truth assignment output by the solver.
A central research area in theoretical computer science

- Cook-Levin Theorem
  - Polynomial-time reductions
- P vs NP
  - One of Clay Institute’s Millenium Problems
  - Resolution worth $1 Million
Hundreds of practical applications:

- **Hardware model checking, Automated Planning**
  
  Software model checking; Termination analysis of term-rewrite systems; Test pattern generation (testing of software & hardware); Model finding; Symbolic trajectory evaluation; Knowledge representation; Games (n-queens, sudoku, etc.); Haplotype inference; Pedigree checking; Equivalence checking; Delay computation; Fault diagnosis; Digital filter design; Noise analysis; Cryptanalysis; Inversion attacks on hash functions; Graph coloring; Traveling salesperson; van der Waerden numbers; itemset mining; etc. etc.

- **Polynomial-time reductions meet the real world!**
Hundreds of practical applications:

- **Hardware model checking, Automated Planning**

  Software model checking; Termination analysis of term-rewrite systems; Test pattern generation (testing of software & hardware); Model finding; Symbolic trajectory evaluation; Knowledge representation; Games (n-queens, sudoku, etc.); Haplotype inference; Pedigree checking; Equivalence checking; Delay computation; Fault diagnosis; Digital filter design; Noise analysis; Cryptanalysis; Inversion attacks on hash functions; Graph coloring; Traveling salesperson; van der Waerden numbers; itemset mining; etc. etc.

**Polynomial-time reductions meet the real world!**
SAT — a Success Story of CS

- Remarkable improvements since mid 90s in SAT solvers: practical decision procedures for SAT

From 100 variables, 200 constraints (early 90s) up to >10,000,000 vars. and 40,000,000 cls. in 20 years.

Beyond Satisfiability

SAT solvers provide highly efficient NP-solvers for tackling real-world search and optimization problems
SAT — a Success Story of CS

- Remarkable improvements since mid 90s in SAT solvers: practical decision procedures for SAT

From 100 variables, 200 constraints (early 90s) up to >10,000,000 vars. and 40,000,000 cls. in 20 years.

Beyond Satisfiability

SAT solvers provide highly efficient NP-solvers for tackling real-world search and optimization problems
Industrial Applications: Examples

SAT solvers a central in various types of software or hardware verification tasks in the industry
  ▶ Indirectly impacting our everyday lives

Examples:
  ▶ Intel core i7 processor designed with the help of SAT solvers [Kaivola et al, CAV 2009]
  ▶ Windows 7 device drivers verified using SAT related technology (Z3, SMT solver) [De Moura and Bjorner, IJCAR 2010]
  ▶ The Eclipse open platform uses SAT technology for resolving dependencies between components [Le Berre and Rapicault, IWOCE 2009]
SAT Solver Competitions

See http://satcompetition.org/

<table>
<thead>
<tr>
<th>SAT 2014 competition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Organizing committee</td>
</tr>
<tr>
<td>Judges</td>
</tr>
<tr>
<td>Proceedings</td>
</tr>
<tr>
<td>Benchmarks</td>
</tr>
<tr>
<td>Solvers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Application</th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
<th>Hard combinatorial</th>
<th>Gold</th>
<th>Silver</th>
<th>Bronze</th>
<th>Random</th>
<th>Gold</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAT+UNSAT</td>
<td>Lingeling</td>
<td>SWDiA5BY</td>
<td>Riss BlackBox</td>
<td>glueSplit_clasp</td>
<td>Lingeling</td>
<td>SparrowToRiss</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAT</td>
<td>minisat_blbd</td>
<td>Riss BlackBox</td>
<td>SWDiA5BY</td>
<td>SparrowToRiss</td>
<td>CCA+glucose</td>
<td>SGSeq</td>
<td>Dimetheus</td>
<td>BalancedZ</td>
<td>CSCCSat2014</td>
</tr>
<tr>
<td>Certified UNSAT</td>
<td>Lingeling (druplig)</td>
<td>glucose</td>
<td>SWDiA5BY</td>
<td>Riss BlackBox</td>
<td>Lingeling (druplig)</td>
<td>glucose</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAT+UNSAT</td>
<td>Plingeling</td>
<td>PeneLoPe</td>
<td>Treengeling</td>
<td>Treengeling</td>
<td>Plingeling</td>
<td>pmcSAT 2.0</td>
<td>pprobSAT</td>
<td>Plingeling</td>
<td>CSCCSat2014</td>
</tr>
<tr>
<td>SAT</td>
<td>MiniSat HACK_999ED</td>
<td>minisat_blbd</td>
<td>ROKKminisat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Core solvers

Core solvers, Parallel

Minisat hack
Improvements in SAT Solvers

Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout

CPU Time (in seconds) vs. Number of problems solved

- Limmat (2002)
- Zchaff (2002)
- Berkmin (2002)
- Forklift (2003)
- Siege (2003)
- SatELite (2005)
- Minisat 2 (2006)
- Picosat (2007)
- Rsat (2007)
- Minisat 2.1 (2008)
- Precosat (2009)
- Glucose (2009)
- Clasp (2009)
- Cryptominisat (2010)
- Lingeling (2010)
- Minisat 2.2 (2010)
- Glucose 2 (2011)
- Glueminisat (2011)
- Contrasat (2011)
Modern SAT Solvers

- Black-box, no command line parameters necessary
- Input: CNF formula, in the standard DIMACS CNF file format
- Output:
  - “UNSATISFIABLE” (+ proof), or
  - “SATISFIABLE” + solution → complete solvers

Freely available open-source implementations

- Minisat
  - http://minisat.se/
- Lingeling
  - http://fmv.jku.at/lingeling/
- Glucose
  - http://www.labri.fr/perso/lsimon/glucose/
- ...

- Implement variants of the CDCL algorithm
Preprocessing, Inprocessing, and Search: Example

Instance: aaai10-planning-ipc5-TPP-21-step11.cnf

c Lingeling SAT Solver
c Copyright (C) 2010-2014 Armin Biere JKU Linz Austria.
c read 99736 variables, 783991 clauses, 1708562 literals in 0.00 seconds
...
s UNSATISFIABLE
...
c 2.344 1% preprocessing 2%
c 140.829 30% inprocessing 98%
c ============
c 143.173 31% simplifying
c 320.824 69% search
c ============
c 464.001 100% all
c 4456232 decisions, 9603.9 decisions/sec
c 1181243 conflicts, 2545.8 conflicts/sec
c 2443039996 propagations, 5.3 megaprops/sec
c 464.0 seconds, 42.1 MB
Categorizing SAT Instances

- Random $k$-SAT instances:
  - Fixed clause-length $k$
  - Clauses generated uniformly at random
- Real-world (application) instances
  - (Some examples already discussed)
- “Crafted” instances
  - “In-between” application instances and random instances
  - Often non-random instances which are hard to solve

No clear categorization

Consider the following examples:
- Graph coloring of randomly generated graphs
- Cryptanalysis

Our focus on this course will be on real-world SAT
Categorizing SAT Instances

- Random $k$-SAT instances:
  - Fixed clause-length $k$
  - Clauses generated uniformly at random
- Real-world (application) instances
  - (Some examples already discussed)
- "Crafted" instances
  - "In-between" application instances and random instances
  - Often non-random instances which are hard to solve

No clear categorization

Consider the following examples:

- Graph coloring of randomly generated graphs
- Cryptanalysis

Our focus on this course will be on real-world SAT
Categorizing SAT Instances

- Random $k$-SAT instances:
  - Fixed clause-length $k$
  - Clauses generated uniformly at random

- Real-world (application) instances
  - (Some examples already discussed)

- “Crafted” instances
  - “In-between” application instances and random instances
  - Often non-random instances which are hard to solve

No clear categorization

Consider the following examples:

- Graph coloring of randomly generated graphs
- Cryptanalysis

Our focus on this course will be on real-world SAT
Categorizing SAT Solvers

Complete
- Given enough time, will give correct answer (UNSAT or SAT)
- Modern SAT solvers: DPLL, CDCL
- Best for:
  - proving unsatisfiability
  - real-world applications

Incomplete
- Modern SAT solvers: stochastic local search
- Heuristically walk around the space of truth assignments
- Unable to determine unsatisfiability
- Best for:
  - Random SAT

\[ x^3 = 0 \]
\[ x^8 = 0 \]
\[ x^3 = 1 \]
\[ x^8 = 1 \]
\[ x^{56} = 0 \]
\[ x^{56} = 1 \]
\[ x^5 = 1 \]

\[ 2^n \] possible solutions

Solution found!

\[ n \text{ variables} \]
Course Outline

Lecture 1: Motivation and Basic Concepts
Lecture 2: The DPLL search procedure and Resolution
  - SAT solvers and proof systems
Lecture 3: Conflict-driven clause learning (CDCL)
  - The most important SAT solving algorithm
Lecture 4: Preprocessing
  - Most important SAT preprocessing techniques
  - Interleaving CDCL search and additional reasoning
Lecture 5: Modelling and Encoding
  - How to represent (encode, model) problems in SAT
Course Outline

Lectures 6–9: Incremental SAT solving.
- How to use SAT-solvers incrementally for developing complex search procedures
  - Lectures 6–7: Extracting minimal unsatisfiable cores
    - Analysing sources of inconsistency
  - Lectures 7–8: Maximum Satisfiability
    - SAT-based Boolean optimization
    - Modern MaxSAT algorithms
  - Lecture 9: Counterexample guided abstraction refinement (CEGAR)
    - Going beyond NP
    - Solving satisfiability of quantified Boolean formulas (QBFs)
Basic Concepts
Propositional Logic

The alphabet of propositional logic:

- Propositional (Boolean) variables $x, y, z, \ldots$
- Parentheses (, )
- Logical connectives (basic Boolean functions)
  - $\neg$: negation (NOT)
  - $\land$: conjunction (AND)
  - $\lor$: disjunction (OR)
  - $\rightarrow$: implication (IF-THEN)
  - $\leftrightarrow$: equivalence (IF AND ONLY IF)
  - $\oplus$: exclusive-or (XOR)
  - $\ldots$
Connectives

Typical ones:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$\top$</th>
<th>$\bot$</th>
<th>$\neg x$</th>
<th>$x \land y$</th>
<th>$x \lor y$</th>
<th>$x \rightarrow y$</th>
<th>$x \leftrightarrow y$</th>
<th>$x \oplus y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Overall, there are 16 different binary connectives.
### Connectives

#### Different notations:

<table>
<thead>
<tr>
<th>operator</th>
<th>“standard”</th>
<th>“alternative”</th>
<th>C/C++/Java/...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>⊤</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>⊥</td>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>negation</td>
<td>¬x</td>
<td>¯x</td>
<td>!x</td>
</tr>
<tr>
<td>conjunction</td>
<td>x ∧ y</td>
<td>x · y</td>
<td>x &amp;&amp; y</td>
</tr>
<tr>
<td>disjunction</td>
<td>x ∨ y</td>
<td>x + y</td>
<td>x</td>
</tr>
<tr>
<td>equivalence</td>
<td>x ↔ y</td>
<td>x = y</td>
<td>x == y</td>
</tr>
</tbody>
</table>
Propositional variables are propositional formulas

If $\phi$ and $\psi$ are propositional formulas, then 

$\neg \phi$,

$(\phi \land \psi)$,

$(\phi \lor \psi)$,

(... and the other connectives...)

are propositional formulas

There are no other propositional formulas
Precedence Rules

How strongly connective bind variables — when to use parenthesis?

Standard:

1. \( \neg \) is stronger than \( \lor \) and \( \land \)
2. \( \lor \) and \( \land \) are stronger than \( \rightarrow \) and \( \leftrightarrow \)

Examples

- \( ((\neg x) \lor y) \lor (\neg z)) \) can be written as \( \neg x \lor y \lor \neg z \)
- \( \neg x \lor y \land z \) not ok — either \( (\neg x \lor y) \land z \) or \( \neg x \lor (y \land z) \)

When in doubt, use parentheses!
Syntax Trees

Example: \((((x \lor y) \lor \neg z)) \leftrightarrow (\neg x \to (\neg y \to z))\)

Subformulas

The set of propositional formulas rooted at the different nodes in the tree form the set of subformulas of the formula.

Example: \((((x \lor y) \lor \neg z)) \leftrightarrow (\neg x \to (\neg y \to z)), (x \lor y) \lor \neg z), \neg x \to (\neg y \to z), (y \lor \neg z, \neg y \to z, \neg x, \neg y, \neg z, x, y, z\)
**Syntax Trees**

Example: \(((x \lor y) \lor \neg z)\) \iff (\neg x \rightarrow (\neg y \rightarrow z))

**Subformulas**

The set of propositional formulas rooted at the different nodes in the tree form the set of *subformulas* of the formula.

Example: \(((x \lor y) \lor \neg z)\) \iff (\neg x \rightarrow (\neg y \rightarrow z)), (x \lor y) \lor \neg z), \neg x \rightarrow (\neg y \rightarrow z), (y \lor \neg z), \neg y \rightarrow z, \neg x, \neg y, \neg z, x, y, z
Propositional Logic: Semantics

- A truth assignment \( \tau : X \rightarrow \{0, 1\} \) assigns truth values to each Boolean variable in a variable set \( X \).
- Semantics of propositional logic is defined recursively:
  - \( \tau(\top) = 1 \)
  - \( \tau(\bot) = 0 \)
  - For propositional formulas \( \phi \) and \( \psi \):
    - \( \tau(\neg \phi) = 1 \) iff \( \tau(\phi) = 0 \).
    - \( \tau(\phi \lor \psi) = 1 \) iff \( \tau(\phi) = 1 \) or \( \tau(\psi) = 1 \).
    - \( \tau(\phi \land \psi) = 1 \) iff \( \tau(\phi) = 1 \) and \( \tau(\psi) = 1 \).
Syntactic Sugar

Semantics for additional connectives:

- For example, the connectives $\neg$ and $\land$ sufficient
- However, other connectives are convenient:
  - $\phi \lor \varphi$ is equivalent to $\neg(\neg\phi \land \neg\varphi)$
  - $\phi \rightarrow \varphi$ is equivalent to $\neg\phi \lor \varphi$
  - $\phi \leftrightarrow \varphi$ is equivalent to $(\phi \rightarrow \varphi) \land (\varphi \rightarrow \phi)$
  - $\phi \oplus \varphi$ is equivalent to $\neg(\phi \leftrightarrow \varphi)$
  - ...

SAT (Lecture 1)

Spring 2016
Properties of Propositional Formulas

- **Satisfiability:**
  - $\tau(\phi) = 1$: truth assignment $\tau$ satisfies formula $\phi$
  - Formula $\phi$ is satisfiable if there is a truth assignment $\tau$ that satisfies it

- **Unsatisfiability:**
  - $\phi$ is unsatisfiable if there is no truth assignment that satisfies it

- **Validity:** $\models \phi$
  - $\phi$ is valid if every truth assignment over the variables in $\phi$ satisfy $\phi$
    - $\phi$ is a tautology

- **Logical equivalence:**
  - Two propositional formulas $\phi$ and $\varphi$ are *logically equivalent* iff they are satisfied by the same set of truth assignments

- **Entailment:** $\phi \models \varphi$
  - $\varphi$ is *logically entailed* by $\phi$ iff any truth assignment that satisfies $\phi$ also satisfies $\varphi$.

- **Equisatisfiability:**
  - $\phi$ and $\varphi$ are equisatisfiable if both are either satisfiable or unsatisfiable.
Properties of Propositional Formulas

- $\phi$ is valid iff $\neg \phi$ is unsatisfiable
- $\phi$ is satisfiable iff $\neg \phi$ is not valid
- $\phi$ and $\psi$ are equivalent iff
  - $\phi \leftrightarrow \psi$ is valid; or, equivalently
  - $\phi \oplus \psi$ is unsatisfiable.
- $\phi$ is satisfiable if $\phi \leftrightarrow \bot$ is not valid.
- ...
CNF: Conjunctive normal form

Every propositional formula can be represented as a logically equivalent propositional formula in CNF

- De facto input format for SAT solvers

Essential: every propositional formula can be compactly represented as a logically equivalent CNF formulas

However:
- “Brute-force” CNF translation of propositional formulas can result in exponential blow-up in the formula size
  - Using e.g. de Morgan’s law
  - Without introducing auxiliary variables
Standard “Tseitin” CNF Encoding for Propositional Formulas

- By introducing auxiliary variables, any propositional formula $\phi$ can be translated into a logically equivalent linear-size CNF in linear-time
  - Often referred to as the standard or Tseitin encoding

**Standard linear-size CNF encoding**

Given a propositional formula $\phi$:

1. For each subformula that is not a literal:
   - Take an auxiliary variable $x_{\phi'}$ for each subformula $\phi'$ of $\phi$
   - Write $x_{\phi'} \leftrightarrow \phi'$ as clauses
     - Clauses for $x_{\phi'} \rightarrow \phi'$ and $\phi' \rightarrow x_{\phi'}$

2. Add unit clause ($x_{\phi}$) to enforce that the formula should be satisfied
Standard CNF Encoding: Example

\[ \phi : \ a \lor (b \land \neg c) \]

Syntax tree of \( \phi \):

- \( \lor \)
  - \( a \)
  - \( \land \)
    - \( b \)
    - \( \neg c \)

CNF Encoding:

1. \( x_\phi \leftrightarrow a \lor x_{b \land \neg c} \)
2. \( x_{b \land \neg c} \leftrightarrow b \land \neg c \)
3. \( (x_\phi) \)
Standard CNF Encoding: Example

\[ \phi : a \lor (b \land \neg c) \]

Syntax tree of \( \phi \):

\[
\begin{array}{c}
\lor \\
a \\
\land \\
b \quad \neg c
\end{array}
\]

CNF Encoding:

1. \( x_\phi \leftrightarrow a \lor x_{b \land \neg c} \)
2. \( x_{b \land \neg c} \leftrightarrow b \land \neg c \)
3. \( (x_\phi) \)
Standard CNF Encoding: Example

\[ \phi : a \lor (b \land \neg c) \]

Syntax tree of \( \phi \):

- \( x_\phi \)
- \( a \)
- \( b \land \neg c \)
- \( b \land \neg c \)

CNF Encoding:

1. \( x_\phi \leftrightarrow a \lor x_{b \land \neg c} \)
2. \( x_{b \land \neg c} \leftrightarrow b \land \neg c \)
3. \( (x_\phi) \)
Standard CNF Encoding: Example

\[ \phi: \quad a \lor (b \land \neg c) \]

Syntax tree of \( \phi \):

\begin{align*}
\lor \quad & x_{\phi} \\
\land \quad & x_{b \land \neg c} \\
\quad & \\
\land \quad & b \\
\quad & \\
\neg \quad & c
\end{align*}

CNF Encoding:

1. \[ x_{\phi} \leftrightarrow a \lor x_{b \land \neg c} \]
2. \[ x_{b \land \neg c} \leftrightarrow b \land \neg c \]
3. \( (x_{\phi}) \)
Standard CNF Encoding: Example

\( \phi : a \lor (b \land \neg c) \)

Syntax tree of \( \phi \):

CNF Encoding:

1. \( x_\phi \iff a \lor x_{b \land \neg c} \)

2. \( x_{b \land \neg c} \iff b \land \neg c \)

3. \( (x_\phi) \)
Standard CNF Encoding: Example

\[ \phi : \quad a \lor (b \land \neg c) \]

Syntax tree of \( \phi \):

CNF Encoding:

1. \( x_\phi \leftrightarrow a \lor x_{b \land \neg c} \)
2. \( x_{b \land \neg c} \leftrightarrow b \land \neg c \)
3. \( (x_\phi) \)
Standard CNF Encoding: Example

\[ \phi : (a \lor (b \land \neg c)) \]

Syntax tree of \( \phi \):

- **CNF Encoding:**
  1. \( x_\phi \leftrightarrow a \lor x_{b \land \neg c} \)
  2. \( x_{b \land \neg c} \leftrightarrow b \land \neg c \)
  3. \( (x_\phi) \)
Standard CNF Encoding: Example

\( \phi : \ a \vee (b \wedge \neg c) \)

Syntax tree of \( \phi \):

CNF Encoding:

1. \( x_\phi \leftrightarrow a \vee x_{b \wedge \neg c} \)
   \( \sim (x_\phi \rightarrow a \vee x_{b \wedge \neg c}) \land (a \vee x_{b \wedge \neg c} \rightarrow x_\phi) \)

2. \( x_{b \wedge \neg c} \leftrightarrow b \wedge \neg c \)

3. \((x_\phi)\)
Standard CNF Encoding: Example

\[ \phi : \overbrace{a \lor (b \land \neg c)}^{x_\phi} \land x_{b \land \neg c} \]

Syntax tree of \( \phi \):

<table>
<thead>
<tr>
<th>CNF Encoding:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ( x_\phi \iff a \lor x_{b \land \neg c} )</td>
</tr>
<tr>
<td>( \sim \quad (x_\phi \rightarrow a \lor x_{b \land \neg c}) \land (a \lor x_{b \land \neg c} \rightarrow x_\phi) )</td>
</tr>
<tr>
<td>( \sim \quad (\neg x_\phi \lor a \lor x_{b \land \neg c}) \land (\neg a \lor x_\phi) \land (\neg x_{b \land \neg c} \lor x_\phi) ) in CNF</td>
</tr>
<tr>
<td>2 ( x_{b \land \neg c} \iff b \land \neg c )</td>
</tr>
<tr>
<td>3 ( (x_\phi) )</td>
</tr>
</tbody>
</table>
Standard CNF Encoding: Example

\[ \phi : \quad \left\{ \begin{array}{l} \quad a \vee (b \land \neg c) \\ \quad x_b \land \neg c \end{array} \right. \]

Syntax tree of \( \phi \):

CNF Encoding:

1. \( x_\phi \iff a \lor x_b \land \neg c \)
   \[ \sim (x_\phi \to a \lor x_b \land \neg c) \land (a \lor x_b \land \neg c \to x_\phi) \]
   \[ \sim (\neg x_\phi \lor a \lor x_b \land \neg c) \land (\neg a \lor x_\phi) \land (\neg x_b \land \neg c \lor x_\phi) \quad \text{in CNF} \]

2. \( x_b \land \neg c \iff b \land \neg c \)
   \[ \sim (x_b \land \neg c \to b \land \neg c) \land (b \land \neg c \to x_b \land \neg c) \]

3. \( (x_\phi) \)
Standard CNF Encoding: Example

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

Syntax tree of \( \varphi \):

\[
\begin{array}{c}
\lor \\
x_\varphi \\
\land \\
a \\
b \\
\neg c
\end{array}
\]

CNF Encoding:

1. \[ x_\varphi \leftrightarrow a \lor x_{b \land \neg c} \]
   \[ \sim \rightarrow \left( x_\varphi \rightarrow a \lor x_{b \land \neg c} \right) \land \left( a \lor x_{b \land \neg c} \rightarrow x_\varphi \right) \]
   \[ \sim \rightarrow \left( \neg x_\varphi \lor a \lor x_{b \land \neg c} \right) \land \left( \neg a \lor x_\varphi \right) \land \left( \neg x_{b \land \neg c} \lor x_\varphi \right) \text{ in CNF} \]

2. \[ x_{b \land \neg c} \leftrightarrow b \land \neg c \]
   \[ \sim \rightarrow \left( x_{b \land \neg c} \rightarrow b \land \neg c \right) \land \left( b \land \neg c \rightarrow x_{b \land \neg c} \right) \]
   \[ \sim \rightarrow \left( \neg x_{b \land \neg c} \lor b \right) \land \left( \neg x_{b \land \neg c} \lor \neg c \right) \land \left( \neg b \lor c \lor x_{b \land \neg c} \right) \text{ in CNF} \]

3. \[ (x_\varphi) \]
## Summary

### Take-home message
- SAT is a major research field, both theoretical and applied aspects
- SAT solvers are generic NP-solvers with

### Study goals
- Propositional logic: syntax, semantics, properties
- CNF: conjunctive normal form
  - Standard DIMACS input format of SAT solvers
  - Standard “Tseitin” encoding: linear-size CNF representation of propositional formulas

### Next time
- Refresher on computational complexity: NP, reductions
- The classical DPLL SAT solving algorithm
- Lookahead solvers
- SAT solvers and Resolution proofs systems