Lecture 2: Complete algorithms and proof systems for SAT: DPLL and lookahead. Resolution proof systems.
Jan 21, 2016
On This Lecture

- Background (continued from first lecture): Refresher on computational complexity / NP
- The DPLL SAT solving algorithm
- Propositional proof systems
  - Resolution and its refinements
  - Resolution and SAT solvers
Refresher on Computational Complexity
A computation problem consists of
- an *instance* of the problem (the input instance)
- a *question* applicable to any instance of the problem

- *Decision problems*:
  - the question has a yes/no answer for any instance of the problem.
  - An algorithm that can provide the correct answer to any instance of a decision problem $B$ is called a *decision procedure* for $B$.
    - Such an algorithm is said to *decide* $B$.

Fundamentally, only problems with *an infinite number of instances* are interesting (Why?)
Decision Problems: Examples

**k-Coloring**

INSTANCE: A graph $G = (V, E)$ and a positive integer $k$.
QUESTION: Is $G$ $k$-colorable?

**SAT**

INSTANCE: A propositional formula $F$ in conjunctive normal form (CNF).
QUESTION: Is $F$ satisfiable?
Other Types of Computational Problems

Instance $I$ of a search / optimization / counting / enumeration problem.

**Search problems**
Find a *solution* to $I$:
- a witness for the “yes” answer of the decision problem.
- answer “no” if there is no solution.

Also known as *function problems*.

**Optimization problems**
Find a *best solution* to $I$,
minimizing or maximizing some *cost function* over all solutions.

**Counting problems**
*Count* the number of solutions to $I$.

**Enumeration problems**
*List all solutions* to $I$. 
Types of Computational Problems related to SAT

Given a CNF formula $F$:

- **Search:**
  Find a *satisfying assignment* to $F$, or prove that none exist.

- **Optimization:**
  Find assignment that satisfies the maximum number of clauses in $F$.
  (MaxSAT)

- **Counting:**
  Count the number of satisfying assignments to $F$.

- **Enumeration:**
  List all satisfying assignments to $F$.

SAT solvers (solving the search problem) form a basis for procedures for optimization, counting, and enumeration.
Some Problems are Easy, Some are Hard

Problem $P$ is *computationally easy*:

There is a polynomial time algorithm for $P$ (i.e., an algorithm whose running time is *polynomially-bounded* w.r.t. the input instance size.

Problem $P$ is (seems) *computationally hard*:

No polynomial-time algorithm is known.

⇒ For any known algorithm $A$:
there is an infinite number of instances of increasing size on which the running time of $A$ increases *super-polynomially* w.r.t. instance size.
Categorization of problems into *problem classes*

The most famous and often practically most relevant distinction is between the problem classes $\mathbf{P}$ and $\mathbf{NP}$

- $\mathbf{P}$ contains all decision problems for which there are *polynomial-time algorithms*.
- $\mathbf{NP}$ contains decision problems for which there are *small (polynomial-size) certificates*, i.e., any possible solution candidate can be checked in polynomial time.

**Example**

$\text{SAT} \in \mathbf{NP}$.

$2\text{-SAT} \in \mathbf{P}$.
Special Cases of Hard Problems

*Special cases* of hard problems can be easy.

**Example**

- In the $k$-SAT problem, each clause of the input instance (CNF formula) contains at most $k$ variables.
- There is a known $O(n^2)$ algorithm for 2-SAT, where $n$ denotes the number of variables in the input instance. [See tutorial 1!]
- There is no known polynomial-time algorithm for $k$-SAT for any $k > 2$. 
P vs NP

P
The class of problems decided by \textit{deterministic polynomial-time Turing machines}.

NP
The class of problems decided by \textit{non-deterministic polynomial-time Turing machines}.

- The “\(P = \text{NP}\)” question is still unresolved.
  - If the verification of a solution is easy, finding a solution may still not be easy.
- \textbf{NP} contains a vast number of hard decision problems that have a lot of practical relevance.
- Counting problems are often even harder.
P, NP, and Beyond

- There are infinitely many problems which may be harder than \textbf{NP} — \textit{polynomial hierarchy}
- $L \subseteq P \subseteq \textbf{NP} \subseteq \textbf{NP}^\text{NP} \subseteq \cdots \subseteq \text{PSPACE}$

On this course the main focus is on SAT and SAT-based \textit{practical approaches} to solving problems in \textbf{NP} (and beyond).
NP v coNP

Problems in NP
short certificates (solutions) that are easy to verify
- SAT: the language of satisfiable CNF formulas

Problems in coNP
short counterexamples that are easy to verify
- UNSAT: the language of unsatisfiable CNF formulas

Practical Perspective
SAT solvers can provide
- Solutions to instances of problems in NP
- Counterexamples to instances of problems in coNP
- ... and: proofs of unsatisfiability (not necessary short)
→ SAT solvers as a basis for counterexample-guided abstraction refinement procedures for problems beyond NP
The relationship between two decision problems $A$ and $B$ can be studied via reductions.

- **$B$ reduces to $A$:**
  There is a transformation (reduction) $R$ which, given any instance $x$ of $B$, produces an input instance $R(x)$ of $A$ for which the following holds: $R(x)$ is a "yes"-instance of $A$ if and only if $x$ is a "yes"-instance of $B$.

- Typically reductions are required to be computable in polynomial time.
Example: Reducing 3-Coloring to SAT

Recall \( k \)-coloring for \( k = 3 \):

INSTANCE: A graph \( G = (V, E) \).

QUESTION: Is \( G \) 3-colorable?

A reduction from 3-COLORING to (3-)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 \( (v, u) \in E \):

\[
\begin{align*}
& \neg v_r \lor \neg u_r \\
& \neg v_g \lor \neg u_g \\
& \neg v_b \lor \neg u_b
\end{align*}
\]

Has the properties of a reduction:

(i) Can be computed efficiently.

(ii) For any 3-COLORING instance, produces a SAT instance \( \text{SAT} \): the graph is 3-colorable if and only if the produced CNF formula is satisfiable.
Reductions in Practice

Reductions are useful in (at least) two ways:

- A reduction $R$ from $B$ to $A$ implies that $A$ is computationally at least as hard as $B$.
  
  For example, any decision problem in $\textbf{NP}$ can be reduced to SAT.

- An algorithm for $B$ can be build from $R$ and an algorithm for $A$.

In practice

- Reductions are often called encodings.

- The goal is to develop encodings such that $R(x)$ is linear, or close to linear, in the size of $x$ for any syntactically valid input $x$. 
Complete algorithms for SAT: DPLL
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
DPLL: Depth-First Search for Satisfiability

**Davis–Putnam–Logemann–Loveland Procedure** [DP’60, DLL’62]
- Classical complete search algorithm for SAT
- Nowadays often dominated by *conflict-driven clause learning* (CDCL) solvers in real-world problem domains
- Still competitive at times on certain types of hard problems

**DPLL**
- Backtracking depth-first search
  - **Branch** by assigning $x = 0$ and $x = 1$ for a currently unassigned variable
  - **Unit propagation**: central pruning technique in SAT solvers
  - **Backtrack** if there is a clause with all literals assigned to 0 (*conflict*)
- Unit propagation applied after every branching step
- Backtracking: undo last branching step
Unit Propagation

One unit propagation step

Each literal \( l_i \) in a clause \((l \lor l_1 \lor \cdots l_k)\) assigned to 0.
\[ \Rightarrow \text{Assign } l \text{ to } 1. \]

Resulting formula: \( F[l = 1] \).

Alternative view

Unit propagating a unit clause \((l)\) over a CNF formula \( F \):
- Remove each clause with \( l \) from \( F \).
- Remove \( \neg l \) from each clause in \( F \).

Resulting formula: \( F[l = 1] \).

Unit propagation: Fixed-point computation

- Successful unit propagation steps produce new assignments / units
- Iterate until no new units are produced
DPLL Pseudocode

\[
\text{DPLL}(F) = \\
F \leftarrow \text{UnitPropagation}(F) \\
\text{if } (F = \emptyset) \text{ return } "\text{satisfiable}" \\
\text{if } (\emptyset \in F) \text{ return } "\text{unsatisfiable}" \\
x \leftarrow \text{ChooseVariable}(F) \\
\epsilon \leftarrow \text{ChooseValue}(F, x) \\
\text{if } (\text{DPLL}(F[x = \epsilon]) = "\text{satisfiable}" \text{ ) return } "\text{satisfiable}" \\
\text{if } (\text{DPLL}(F[x = 1 - \epsilon]) = "\text{satisfiable}" \text{ ) return } "\text{satisfiable}" \\
\text{return } "\text{unsatisfiable}" \\
\]

Can also produce a satisfying assignment
DPLL: Example

Unsatisfiable CNF formula $F$:
$$(x_1 \lor x_2) \land (x_4 \lor \neg x_2) \land (x_5 \lor \neg x_4) \land (x_3 \lor \neg x_4) \land (\neg x_5 \lor \neg x_3) \land (\neg x_1 \lor x_4 \lor x_6) \land (\neg x_6)$$

DPLL search tree (proof) for $F$
DPLL Branching Heuristics

How to

- ... choose the next variable \( x \) to branch on?
- ... choose which value to assign first?

Classical heuristics

Based on counting variable occurrences in clauses. Intrinsically greedy

- Bohm
- MOMS
  - Branch on variable that has the maximum number of occurrences in clauses of minimum size
- Jeroslaw-Wang
  - Counting literal occurrences, inverse exponential weighting
  - Two-sided: both literals of variable, choose which branch to search first based the counts
Further Propagation/Inference Mechanisms

Pure literal elimination:
If a variable $x$ occurs only positively ($x$) or only negatively ($\neg x$) in the formula:
Assign $x$ according to the polarity ($x = 1$ or $x = 0$).

Failed literal elimination:
If assigning $x = 0$ (or $x = 1$) results in a conflict after unit propagation:
Assign $x = 1$ (or $x = 0$).
Lookahead

Lookahead on variable \( x \)
1. Assign \( x = 0 \) (or \( x = 1 \)).
2. Unit propagate.
3. Evaluate the effects \( \Delta(F, x = 0) \) (or \( \Delta(F, x = 1) \)) of 1) and 2).

Lookahead-based branching heuristics for DPLL
1. Lookahead on the variables in the formula.
2. Choose “a best” variable to branch on based on the effects of looking ahead.
Lookahead: Practical Considerations

A concrete heuristic

- Let $\Delta(F, x = \epsilon) =$ the number of additional variable assignment by unit propagation after assigning $x = \epsilon \in \{0, 1\}$.
- Branch on a variable in $\arg \max_x \Delta(F, x = 0) \cdot \Delta(F, x = 1)$.

Pitfalls of looking ahead

- With millions of variables, too costly to perform on all variables
  - Possible solution: restrict lookahead to a subset of “promising” variables
- Implementation requires care
  - Tree-based lookahead in the March lookahead solver  
  [HeuleJB CPAIOR’14]
  - Can at the same time detect
    - Failed literals
    - Equivalent literals
    - …
Propositional Proof Systems: Resolution
Resolution

Classical complete proof system for SAT

The Resolution rule

\[
\frac{(x \lor C) \land (\neg x \lor D)}{(C \lor D)}
\]

- \(C \lor D\) is the resolvent of \(x \lor C\) and \(\neg x \lor D\), obtained by resolving on 
  \(x\).

- Resolution derivation of a clause \(C\) from a CNF formula \(F\):
  Sequence of clauses \(\pi = (C_1, C_2, \ldots, C_k = C)\) such that for each \(C_i\), either
  - \(C_i \in F\), i.e., \(C_i\) is an original clause in \(F\); or
  - \(C_i\) is a resolvent of two clauses \(C_j, C_{j'}\) with \(j, j' < i\) in \(\pi\).

- Resolution refutation (i.e., proof for the unsatisfiability) of a CNF formula \(F\): Derivation of the empty clause \(\emptyset\) from \(F\).
Resolution Proof DAGs

Any resolution derivation $\pi = (C_1, C_2, \ldots, C_k)$ can be represented as a directed acyclic graph (DAG):

- Each $C_i$ in $\pi$ is represented as a unique node.
- $C_i$ is a resolvent of two clauses $C_j$, $C_j'$ with $j, j' < i$ in $\pi$: directed edges $(C_j, C_i)$ and $(C_{j'}, C_i)$

Note:

- Each $C_i \in F$ is a source node
- The empty clause $\emptyset$ is the unique sink node.
Resolution Proofs: Example

Unsatisfiable CNF formula $F$:

$$(x_1 \lor x_2) \land (x_4 \lor \neg x_2) \land (\neg x_5 \lor \neg x_4) \land (x_3 \lor \neg x_4) \land (x_5 \lor \neg x_3) \land (\neg x_1 \lor x_4 \lor x_6) \land (\neg x_6)$$

Resolution proof for $F$:

```
\text{Resolution Proofs: Example}

Unsatisfiable CNF formula $F$:

$$(x_1 \lor x_2) \land (x_4 \lor \neg x_2) \land (\neg x_5 \lor \neg x_4) \land (x_3 \lor \neg x_4) \land (x_5 \lor \neg x_3) \land (\neg x_1 \lor x_4 \lor x_6) \land (\neg x_6)$$

Resolution proof for $F$

```

![Resolution Proof Diagram]

\[\text{Resolution Proofs: Example} \]

Unsatisfiable CNF formula $F$:

$$(x_1 \lor x_2) \land (x_4 \lor \neg x_2) \land (\neg x_5 \lor \neg x_4) \land (x_3 \lor \neg x_4) \land (x_5 \lor \neg x_3) \land (\neg x_1 \lor x_4 \lor x_6) \land (\neg x_6)$$

Resolution proof for $F$:

```
\text{Resolution Proofs: Example}

Unsatisfiable CNF formula $F$:

$$(x_1 \lor x_2) \land (x_4 \lor \neg x_2) \land (\neg x_5 \lor \neg x_4) \land (x_3 \lor \neg x_4) \land (x_5 \lor \neg x_3) \land (\neg x_1 \lor x_4 \lor x_6) \land (\neg x_6)$$

Resolution proof for $F$

```

![Resolution Proof Diagram]
Soundness and Completeness

Important properties of proof systems

- Practical perspective: all generated proofs (and satisfying truth assignments) are correct

**Completeness:** A propositional proof system is *complete* if any unsatisfiable CNF formula can be refuted in the system.

**Soundness:** A proof system is *sound* if any CNF formula, that can refuted in the system, is unsatisfiable.

Resolution is sound and complete
Resolution Refinements

Various *refinements* of Resolution have been proposed and studied.

- Refinements are defined via imposing restrictions on (eg) the structure of available proofs.

**Some important sound and complete refinements:**

- **Treelike Resolution:**
  Resolution proof DAGs that are trees.
  - Resolvents must be re-derived each time they are used as antecedents.
Resolution Refinements

Some important sound and complete refinements:

- **Regular Resolution:**
  Proof DAGs in which each variable is resolved on at most ones on each path in the DAG.

- **Linear Resolution:**
  Resolution proof DAGs that are linear.
  - Every clause $C_i$ in the proof has to be derived from $C_{i-1}$ and some other clause $C_j$, $j < i - 1$.

- **Ordered Resolution / DP Resolution:**
  Eliminate variables by producing all possible resolvents obtained by resolving on a variable
  - Davis-Putnam Resolution [DP’60]
Complexity Measures

Assume a Resolution proof $\pi = (C_1, C_2, \ldots, C_k = \emptyset)$.

**Length:** length of $\pi$ is $k$.  
*The most studied measure.*

**Width:** the width of $\pi$ is  
$$\max_{i=1..k} |C_i|,$$  
i.e., the length of a longest clause in $\pi$.

**Space:** maximum number of clauses needed in memory while verifying the correctness of a proof.
Resolution Refinements and SAT Solvers

Although partially coincidental:
Proofs available for Resolution refinements and proofs produced by SAT solvers have tight connections.

- Proofs produced by DPLL essentially coincide with Treelike Resolution proofs.
- Proofs produced by CDCL SAT solvers are tightly (polynomially) related with (unrestricted) Resolution proofs.
- Clauses produced by the clause learning mechanism of CDCL solvers can be derived with “Trivial” Resolution

These connections can be formalized via the concept of polynomial simulation.
Comparing Proof Systems: Simulation

Polynomial simulation

Showing that proof system $P$ polynomially simulates proof system $P'$

Show that, given any CNF formula $F$, there is a $P$ proof $p$ for $F$ such that $p$ has polynomial length wrt the length of the shortest $P'$ proofs for $F$.

$P$ and $P'$ are polynomially equivalent if they polynomially simulate each other.

Practical perspective: assuming optimal search heuristics
Comparing Proof Systems: Separation

Superpolynomial/Exponential Separation

Showing that proof system $P$ does not polynomially simulate $P'$

Show that there is an infinite family $\{F_n\}_n$ of CNF formulas such that for any $i$, there is no $P$ proof for $F_i$ of polynomial length wrt the length of the shortest $P'$ proofs for $F_i$.

$P$ is superpolynomially stronger than $P'$ if

- $P$ polynomially simulates $P'$; and
- $P'$ does not polynomially simulate $P$.

- Practical perspective: assuming optimal search heuristics
Relative Power of Resolution Refinements

$P \rightarrow P'$: $P$ is stronger than $P'$.

On the relative power of Resolution refinements

Resolution $\rightarrow$ Regular $\rightarrow$ Treelike
Resolution and SAT Solvers
Resolution and SAT Solvers

\[ P \rightarrow P': \] \quad P \text{ is stronger than } P'.
\[ P \approx P': \] \quad P \text{ and } P' \text{ are polynomially equivalent.}

Res. \approx \text{CDCL}^* \quad \rightarrow \quad \text{Regular Res.} \quad \rightarrow \quad \text{DPLL} \approx \text{Treelike Res.}

CDCL*: CDCL with relatively realistic assumptions

Next lecture focuses on CDCL — more details then
Treelike Resolution and DPLL are Polynomially Equivalent

Idea, from DPPL to Treelike Resolution:

- View unit propagation steps as branching steps
  - “other” branch with give a direct conflict with a unit clause
- Search tree root: empty formula $\emptyset$
- For every branch in the tree:
  - assignment in branch falsifies an original clause
  - attach such a clause in the leaf

Whiteboard example, for unsatisfiable CNF formula $F$:

$$(x_1 \lor x_2) \land (x_4 \lor \neg x_2) \land (x_5 \lor \neg x_4) \land (x_3 \lor \neg x_4) \land (\neg x_5 \lor \neg x_3) \land (\neg x_1 \lor x_4 \lor x_6) \land (\neg x_6)$$

$x_6 = 0$

$x_1 = 0, x_2 = 1, x_4 = 1, x_5 = 1, x_3 = 1$
Resolution and CDCL

- Due to CDCL SAT solvers being more complex than DPLL, arguing about the power of CDCL is more challenging
  - CDCL solvers include a variety of refined search techniques:
    - Clause learning + learning heuristics
    - Decision heuristics
    - Clause forgetting heuristics
    - Restart policies
    - Capturing behavior of CDCL in an abstract proof system is a challenge

Under relatively realistic assumptions:
CDCL SAT solvers are polynomially equivalent to Resolution.
[PipatsrisawatD’11]

Next lecture focuses on CDCL — more details then
Transferring Results for Resolution to SAT Solvers

If smallest Resolution proofs for a family of CNF formulas $\{F_n\}_n$ are of superpolynomial length, then the running time of CDCL SAT solvers is bound to scale exponentially on these formulas.

Understanding what is provable hard to solve for state-of-the-art SAT solvers may give hints towards developing new solving techniques.
Hard Examples for Resolution

Examples:

**Pigeon-Hole Problem PHP**

**PHP**$_n$: Cannot put $n + 1$ pigeons into $n$ holes so that each pigeon gets its own hole.

Haken'85: There are no polynomial-length Resolution proofs for PHP$_n$.

**Random $k$-CNF**

For example:

Any Resolution proof of a randomly chosen 3-CNF formula with at most $n^{6/5-\epsilon}$ clauses is of exponential length. [BeameP’96]

- First exponential lower bounds for random formulas by Chvatal and Szemeredi ’88

**Practical Perspective**

The running times of DPLL and CDCL SAT solvers are bound to scale exponentially on such families of formulas.
Hard Examples for Resolution: PHP

Variables:

\[ p_{i,j} = \text{“pigeon } i \text{ sits in hole } j \text{”} \quad \text{for } i = 1..n+1, \quad j = 1..n \]

Clauses:

- Each pigeon gets at least one hole:

\[
\bigwedge_{i=1}^{n+1} \bigvee_{j=1}^{n} p_{i,j}
\]

- No two pigeons get the same hole:

\[
\bigwedge_{j=1}^{n} \bigwedge_{i=1}^{n} \bigwedge_{i<i'\leq n+1} (\neg p_{i,j} \lor \neg p_{i',j})
\]
Beyond CDCL: Extended Resolution

Extension Rule

Given a CNF formula $F$, let $x, y$ be variables in $F$.

The extension rule of Extended Resolution allows adding to $F$ the clauses

$$(\neg e \lor x), (\neg e \lor y), (e \lor \neg x \lor \neg y),$$

where $x, y$ are variables in $F$, and $e$ is a new variable.

- The clause represent $e \leftrightarrow x \land y$.

Extended Resolution

- Extension rule
- Resolution rule

There are polynomial-length Extended Resolution for PHP$_n$.

- In fact: Extended Treelike Resolution is enough.
Short Extended Resolution Proofs for PHP

Cook'76: There are polynomial-length Extended Resolution for PHP\(_n\).

Basic idea:
- Build an extension which allows to simulate the induction proof for refuting PHP
  - “If PHP\(_n\) holds, then PHP\(_{n-1}\) holds.”
  - That is: remove one pigeon and hole at a time.

Lower bounds for Extended Resolution?

*Extended Resolution has not been shown to have superpolynomial proofs*
- If there is a polynomially-bounded propositional proof system, then NP=coNP!
Extended Treelike Resolution proofs are “enough”

For every extended resolution proof of length $l$, there is a Extended Treelike Resolution proof of length $O(l)$.

- Assume Extended Resolution proof $(C_1, \ldots, C_i, C_{i+1}, \ldots, C_k)$:
  - $C_j$ for $j \leq i$ either original clauses or clauses produced with the extension rule
  - $C_j$ for $j > i$ derived with the resolution rule

- Treelike proof:
  - Start with $C_1, \ldots, C_i$.
  - For each $C_j$ for $j > i$, add extension $e_j \leftrightarrow C_j$.
  - Apply treelike resolution by resolving on the $e_j's$ from $j = i + 1$ to $j = k$.

Viewed as DPLL (lookahead!) proof of the extended formula

- Branch according to $e_{i+1}, e_{i+2}, \ldots, e_k$

- Branch with $e_j = 0$ (given $e_{j'} = 1$ for all $j' < j$) gives a conflict by unit propagation (why?)

- The branch with $e_j = 1$ for all $j$ gives a conflict (why?)
Summary

Take-home message
- Reductions allow for solving various NP problems via SAT solvers
- SAT solvers form a basis for optimization, counting, and enumeration
- DPLL: classical complete search procedure for SAT
- Connections between SAT solvers and Resolution refinements

Study goals
- DPLL and unit propagation
- Resolution proofs

Next time
- The Conflict-Driven Clause Learning (CDCL) algorithm