CSC2512
Algorithms for Solving Propositional Theories
Proof Systems.

A proof system for a language \( L \) is a polynomial time algorithm \( PC \) s.t.

- For all inputs \( F \)
  \( F \in L \) iff there exists a string \( P \) s.t. \( PC \) accepts input \( (F, P) \)

EXAMPLE

\( L \) is the set of unsatisfiable CNF formulas. \( F \) is a sample CNF, and we want to test if \( F \) is unsatisfiable.

\( P \) is a proof that \( F \) is UNSAT, this proof is valid if there is a proof-checking algorithm (verifier) \( PC \) that runs in time polynomial in the size of \( P \) and \( F \)

The string \( P \) is a proof, e.g., a resolution refutation. But other proof systems exist that verify other type of proofs.
Proof Systems.

- The **complexity** of a proof system, $PC$ for a language $L$ is a function

$$f(n) = \max_{F \in L, |F| = n} \min_{P \text{ s.t. } PV \text{ accepts } (F, P)} |P|$$

- The smallest proof of any $F$ that is accepted by the proof system. $f(n)$ is how the maximum smallest proof grows as the length of $F$ grows.
Proof Systems.

- Given two proof systems $\mathrm{PC}_1$ and $\mathrm{PC}_2$ we say that $\mathrm{PC}_1$ $\text{p-simulates}$ $\mathrm{PC}_2$ if there is a polynomially computable function $f$ such that for any proof $P_2$ of $\mathrm{PC}_2$ (i.e., proof accepted by $\mathrm{PC}_2$) $f(P_2)$ is a proof of $\mathrm{PC}_1$.

- In other words any proof of $\mathrm{PC}_2$ can be converted to a proof of $\mathrm{PC}_1$ with at most a polynomial increase in size (if the size increased non-polynomially, $f$ could not be computed in polynomial time).
Resolution Proof system

- Resolution is a proof system. Given an unsatisfiable CNF $F$ a proof $P$ of $F$ is a sequence of clauses as defined before.
- The proof system (or checker) can check that every step of $P$ is a valid step (a clause from $F$ or the result of resolving two prior clauses in $P$), and that $P$ ends in the empty clause.
- Clearly this check can be done in time polynomial in the length of $P$.
- However, the complexity of resolution is $O(2^n)$. That is, formulas exist of length $n$ that require exponentially long proofs.
Resolution “Refinements”

• A number of special cases of resolution have been defined and studied empirically and theoretically.

• These special cases are called refinements, although they are actually restrictions of the general case not improvements.

• Each refinement forms a new proof system:

  For a refinement the proof checker will accept only resolution proofs of a certain structure.
Resolution DAGS

Represent Resolution proofs as DAGs:
1. Arcs go from clauses to the two clauses whose resolution yielded it.
2. The only source node is the empty clause ()
3. Each clause of the input formula F has out-degree 0 (these are sink nodes)
4. Every other clause has out-degree 2 (points to the two clauses that produced it via a resolution step)
5. The arcs pointing to (A,x) and (B,-x) are labeled with the literals x and –x. (Represents the literal that was removed from the clause the arc points to).
Resolution “Refinements”

1. Negative Resolution
A resolution step \( R[(A,x), (-x, B)] = (A,B) \) is negative whenever \( B \) contains only negative literals. Negative Resolution requires that all resolution steps be negative.

2. Semantic Resolution
Given a truth assignment \( \pi \) to the variables of \( F \) a \( \pi \)-refutation of \( F \) is a resolution refutation such that when two clauses are resolved at least one of them must be falsified by \( \pi \). A refutation of \( F \) is called semantic if it is a \( \pi \)-refutation for some truth assignment.

– Are negative resolutions semantic?
Resolution “Refinements”

3. Linear Resolution
Each refutation must have a linear underlying DAG:
- The proof consists of a sequence of clauses $c_1, c_2, \ldots, c_m = ()$, such that either $c_i$ is a clause of $F$ or $c_i$ is derived from $c_{i-1}$ and $c_j$ for some $j < i-1$.

DAG looks like?

4. Regular Resolution
In the DAG of each refutation each path from the source empty clause node to a clause of $F$ (sink node) has the property that no variable appears more than once as an arc-label.
Resolution “Refinements”

5. Ordered Resolution
   In the DAG of each refutation the sequence of variables labeling each path from the source node to a sink node respects some total ordering of the variables.

6. Tree-Like Resolution
   In the DAG of each refutation is a tree (we can use the input clauses multiple times)

Each of these refinements of resolution is sound and complete.
CSC2512: Satisfiability

What Type?

\[\neg X, X, \neg Y, Y, \neg Z, \neg X, Q, \neg Z, \neg X, \neg Q, X\]
### Known P-Simulation Results

<table>
<thead>
<tr>
<th></th>
<th>Neg</th>
<th>Sem</th>
<th>Lin</th>
<th>Order</th>
<th>Reg</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Sem</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Lin</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Order</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Reg</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Tree</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Cell (i,j) = does refinement of row i p-simulate refinement of column j

No means further that Refinement i requires on some formulas an exponentially long proof while Refinement j has polynomial sized proofs for the formula.

**Source:** “The Complexity of Resolution Refinements” by Buresh-Oppenheim and Pitassi
CSC2512: Satisfiability

Known P-Simulation Results

• Some key results:
  • **Regular** is a generalization of Tree and Ordered:
    – Both Tree and Ordered proofs are regular proofs.
  • **Tree** and **ordered** are very weak. They both require exponential sized proofs for formulas that other systems can prove with polynomial sized proofs.
  • **Regular** also not that powerful
CSC2512: Satisfiability

Not Regular

Regular
CSC2512: Satisfiability

Tree Resolution

C1, C2, C3 → C4, C5 → C1, C2, C3
DP produces ordered resolution proofs.

- DP was developed prior to resolution, but every DP run that yields the empty clause contains an ordered proof.

<table>
<thead>
<tr>
<th></th>
<th>[a]</th>
<th>[b]</th>
<th>[c]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a,b,c)</td>
<td>(b,c)</td>
<td>(c)</td>
<td>()</td>
</tr>
<tr>
<td>(¬a,b,c)</td>
<td>(¬b, c)</td>
<td>(¬c)</td>
<td></td>
</tr>
<tr>
<td>(¬b, c)</td>
<td>(¬b,¬c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a,¬b,¬c)</td>
<td>(b,¬c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(¬a,¬b,¬c)</td>
<td>(b,¬c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b,¬c)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DP produces ordered resolution proofs.

- Every DP run that yields the empty clause contains an ordered proof.

```
<table>
<thead>
<tr>
<th></th>
<th>[a]</th>
<th>[b]</th>
<th>[c]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a,b,c)</td>
<td>(b,c)</td>
<td>(c)</td>
<td>()</td>
</tr>
<tr>
<td>(¬a,b,c)</td>
<td>(¬b, c)</td>
<td>(¬c)</td>
<td></td>
</tr>
<tr>
<td>(¬b, c)</td>
<td>(¬b, ¬c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(a,¬b,¬c)</td>
<td>(¬a,¬b,¬c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(¬a,¬b,¬c)</td>
<td>(b,¬c)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b,¬c)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(¬b, c)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(¬b,¬c)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b,¬c)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(¬c)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Potentially many redundant clauses are generated, but an ordered resolution is contained in these clauses.
DP is not very effective at determining SAT.
1. Let $F$ be a formula that requires an exponentially sized ordered refutation.

What will be the run time of DP on $F$?

2. Also DP has high space complexity—the updated sets of clauses $C$ become exponentially large.

Two years later Davis, Logemann and Loveland developed a new procedure for testing SAT (also predating resolution).

This procedure required only linear space. The algorithm became known as DPLL (although Putnam didn’t play a role).

DPLL was a backtracking search algorithm (backtracking originated much earlier)

DPLL(\(\pi\), F) \ // Initially F is the input formula. 
\(\pi\) is an empty set of literals (truth assignment)

If F is empty
  return SAT (\(\pi\) is a satisfying assignment)
else choose a variable v in F \ //choose a v appearing
  //in a unit clause if one exists

F' = F|_v \ //Reduce F
if F' does not contain an empty clause
  DPLL(\(\pi + v\), F')

F' = F|_{-v}
if F' does not contain an empty clause
  DPLL(\(\pi + v\), F')
return UNSAT
CSC2512: Satisfiability

Reduction:

\[ F|_l \]  F reduced by literal \( l \)

Remove all clauses of \( F \) that contain \( l \)
Remove \(-l\) from all remaining clauses.

\[ \{(a, b, -d), (d, c, e), (g, h, e)\}|_d \]
\[ = \{(c, e), (g, h, e)\} \]

Note \((F|_a)|_b = (F|_b)|_a\), so we write \( F|_{a,b} \)
CSC2512: Satisfiability

Example:
F = (a, -b), (-a, b), (-b, c), (a, c), (a, -c), (-b, -c)
A tree resolution can be extracted from DP whenever it runs on an unsatisfiable formula

Example:
\[ F = (a, -b), (-a, b), (-b, c), (a, c), (a, -c), (-b, -c) \]
CSC2512: Satisfiability

DPLL is not very effective at determining SAT.
1. Let $F$ be a formula that requires an exponentially sized tree refutation.

What will be the run time of DPLL on $F$?
CSC2512: Satisfiability

DP is equivalent to ordered resolution
DPLL is equivalent to tree resolution

1. Any ordered resolution proof can be generated by some run of DP: just follow the same order of the variables
2. Any tree resolution proof can be generated by some run of DPLL: just branch on the variables in the same order as a pre-order traversal of the DAG (starting at the root)
CSC2512: Satisfiability

Modern SAT solvers

1. Based on DPLL
2. More efficient implementation methods.
3. Additional inference allowing them to move beyond tree resolution.
Sources of Inefficiency

**DPLL(π, F)**

If $F$ is empty
   return SAT ($π$ is a satisfying assignment)
else choose a variable $v$ in $F$ //choose a $v$ appearing
   //in a unit clause if one exists
\[
F' = F|_v \quad \text{ //How Implemented? Copy? Modify/restore?}
\]
if $F'$ does not contain an empty clause //How to test?
   DPLL($π + v, F'$)
\[
F' = F|_{-v}
\]
if $F'$ does not contain an empty clause
   DPLL($π + -v, F'$)
return UNSAT
Unit Preference vs. Unit Propagation

E.g. (a, b) (-b, a, d) (-d, -b, e) (-d, -e, c, g) (-a)

DPLL would choose a: a = True yields an empty clause
a = False yields (b) (-b, d) (-d, -b, e) (-d, -e, c, g)

Now have to choose b: b = False yields an empty clause
b = True yields (d) (-d, e) (-d, -e, c, g)

Now have to choose d: d = False yields an empty clause
d = True yields (e) (-e, c, g)
Unit Preference vs. Unit Propagation

One Unit clause yields new unit clauses via reduction. DPLL would choose a sequence of the variables exploring the path that makes all units (and generated units) true (the other side always yields a false clause). Each new unit requires a new recursion of the algorithm.
On large industrial problems this chaining of units might yield hundreds of new units. Need efficient way of doing it.

Don’t implement as a sequence of choices of unit variables. Instead choose a variable and then find all variables that would recursively appear in unit clauses:
set the value of all of these variables right after the choice, so as to satisfy all units before choosing the next variable.

This process is called **Unit Propagation**.
DPLL(\(\pi, F\))  //Perform UP(F) before invoking DPLL

If \(F\) is empty
  return SAT (\(\pi\) is a satisfying assignment)
else choose a variable \(v\) in \(F\)  //no unset variable appears  
  //in a unit clause

\[ F' = UP(F|_v) \]

if \(F'\) does not contain an empty clause
  DPLL(\(\pi + v, F'\))

\[ F' = UP(F|_{-v}) \]

if \(F'\) does not contain an empty clause
  DPLL(\(\pi + \neg v, F'\))

return UNSAT
Recursion, how do we compute \( F|_l \) and then restore \( F \) so that we can compute \( F|_{-l} \)?

Make a copy of \( F \) then reduce?
   Large instances can require more than 5 Mbytes

Make changes then restore?
   Unit propagation can force hundreds of variables forcing extensive changes.
Modern technique:

Why do we need $F|_i$?

Only two reasons in the algorithm
   Find units, and perform unit propagation.
   Find empty clauses—but empty clauses must first become unit.

Can do these two things without computing $F|_i$