Algorithms for Computational Logic

Introduction

Emmanuel Hebrard (adapted from **João Marques Silva**, Inês Lynce and Vasco Manquinho)

Outline

1. Introduction to Boolean Satisfaction
2. Boolean Reasoning

Outline

1. **Introduction to Boolean Satisfaction**
   - Propositional Logic
   - The Satisfiability Problem
   - Some Fragments of Propositional Logic
2. Boolean Reasoning
   - Unit Propagation
   - Resolution
   - Proof Systems

Propositional Logic

Proposition

A *proposition* is an assertion that can be:

- assigned a truth value (**true** or **false**)
- written using *atomic propositions* (or *atoms*) and *logic connectors*

An atom is a proposition written using a unique symbol.

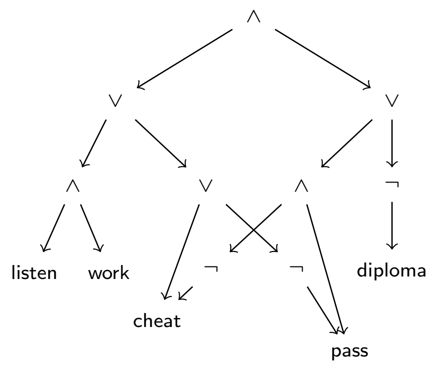
- Atomic propositions:
  - "Adam follows the lecture", "Adam works at home", "Adam cheats at the exam", "Adam passes the exam"
- Propositions:
  - "**if** Adam does not listen the lecture **and** does not work at home **then** he will not pass the exam **unless** he cheats

Formulae (syntax)

A non-atomic proposition (*Formula*) φ is either:

- an atom
- the negation $\neg\psi$ of another proposition ψ
- the concatenation of two or more propositions φ_1 and φ_2 by a logical connector $\{\wedge, \vee, \rightarrow, \oplus, \ldots\}$

$$((\text{“listen lecture”} \wedge \text{“work at home”}) \vee \text{“cheat”} \vee \neg\text{“pass exam”}) \quad \wedge$$
$$((\neg\text{“cheat”} \wedge \text{“pass exam”}) \vee \neg\text{“get diploma”})$$



Models (interpretations)

A *model* $\mathcal{A}$ is a mapping from atoms in $\mathcal{X}$ to $\{\mathbf{true}, \mathbf{false}\}$. We write $\mathcal{A} \models x$ for "Atom x is true in model $\mathcal{A}$"

A proposition φ written using atoms in $\mathcal{X}$ can be interpreted (given a truth value) using a model $\mathcal{A}$ on $\mathcal{X}$:

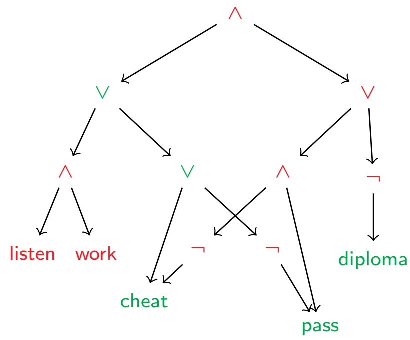
- if φ is the negation of a proposition ψ, then $\mathcal{A} \models \varphi$ if and only if $\mathcal{A} \not\models \psi$
- if φ is a conjunction $\varphi_1 \wedge \varphi_2$, then $\mathcal{A} \models \varphi$ if and only if $\mathcal{A} \models \varphi_1$ **and** $\mathcal{A} \models \varphi_2$
- if φ is a disjunction $\varphi_1 \vee \varphi_2$, then $\mathcal{A} \models \varphi$ if and only if $\mathcal{A} \models \varphi_1$ **or** $\mathcal{A} \models \varphi_2$

- Ex: $\text{“listen lecture”} \wedge \text{“work at home”} \wedge \neg\text{“cheat”} \wedge \neg\text{“pass exam”} \wedge \neg\text{“get diploma”}$

Models: examples

$$((\text{“listen lecture”} \land \text{“work at home”}) \lor \text{“cheat”} \lor \neg\text{“pass exam”}) \quad \land$$
$$((\neg\text{“cheat”} \land \text{“pass exam”}) \lor \neg\text{“get diploma”})$$

	“listen lecture”	“work at home”	“cheat”	“pass exam”	“get diploma”
$\mathcal{A}_1$	**false**	**false**	**true**	**true**	**true**



The Satisfiability Problem

SAT

- **data**: A Boolean formula ϕ
- **question**: Does there exist an interpretation that satifies ϕ?

- A formula is *satisfiable* iff there exists an interpretation that satisfies it
- A formula φ is *unsatisfiable* iff there is no interpretation that satisfies it
  - Write it $\text{UNSAT}(\varphi)$
- A formula is *valid / a tautology* iff all interpretations satisfy it
  - Equivalent to $\text{UNSAT}(\neg\varphi)$
- A formula ψ is an *implicate* of φ iff all interpretations satisfying φ also satisfy ψ
  - Equivalent to $\text{UNSAT}(\varphi \land \neg\psi)$
- A formula ψ is an *implicant* of φ iff φ is an *implicate* of ψ

Examples of Applications

- Linux package upgrade
  - The Eclipse foundation uses Daniel le Berre's SAT solver **SAT4j** to solve this problem
  - Equinox/p2/CUDFResolver
- (Re-)Attribution of the TV radiospectrum by the Federal Communications Commission (FCC) in 2017
  - The radiofrequency allocation problem corresponds to *Graph Coloring*
    - Vertices are broadcasters, colors are frequencies
    - Easy to encode as SAT
  - Reverse auction: the FCC buys frequencies and starts with high quotes that decrease at each round
    - Stops when it is *not* possible to assign frequencies to broadcasters who opted out
  - Critical to *prove* unsatisfiability (the auction yielded $20 billion)

Cook-Levin Theorem

- SAT is in $\mathbf{NP}$, the interpretation σ that satisfies it is a polynomial certificate

Théorème de Cook-Levin

SAT is $\mathbf{NP}$-complete

  - At least as hard as any problem in $\mathbf{NP}$
  - If SAT is in $\mathbf{P}$ then $\mathbf{P} = \mathbf{NP}$

Fragments of SAT

- Fragments of SAT are particular case defined by the *language*
  - Using only negation ($\neg$), disjunction ($\vee$) and conjunction ($\wedge$) is not restrictive

Disjunctive Normal Form

- Disjunctive normal form:
  - Disjunction of conjunctions (sum) of literals (products)
  - Ex: $(\neg a \wedge b \wedge c) \vee (\neg b \wedge \neg c) \wedge (a \wedge \neg b)$
- Every product is an *implicant*, and corresponds to an interpretation
- Satisfiability of a DNF is easy

Conjunctive Normal Form

- Conjunctive normal form:
  - Conjunction of disjunctions of literals (clauses)
  - Ex: $(\neg a \vee b \vee c) \wedge (\neg b \vee \neg c) \wedge (a \vee \neg b)$
- For any formula φ, there is a CNF formula φ' such that
  - $\mathrm{SAT}(\varphi) \iff \mathrm{SAT}(\varphi')$
  - $|\varphi'| \in \mathcal{O}(|\varphi|^c)$ for some constant c
- Every clause is an *implicate*
- Validity of a CNF is easy

Horn Clauses

- Horn clause:
  - Clause with at most one positive literal
  - Ex: $(\neg a \vee \neg c \vee b) \wedge (\neg b \vee \neg c) \wedge (\neg b \vee a)$
  - Equivalent to implications
    - ★ $(a \wedge c \Rightarrow b) \wedge (b \wedge c \Rightarrow \textbf{false}) \wedge (b \Rightarrow a)$

The DIMACS format

- Comments
- Header [#variables(=5)] [#clauses(=7)]
- Variables are numbered 1 to n
- One line per clause '0' is a delimiter
- positive (negative) numbers are positive (negative) literals
  - $(\neg x_1 \vee x_3 \vee \neg x_5 \vee x_4)$

```
c This line is a comment.
p cnf 5 7
-1 3 -5 4 0
2 -3 0
1 5 0
-3 -4 0
-1 2 4 0
-2 0
2 -3 -5 0
```


Basic definitions

- Typename/classes
  - **Variable**: used for indexing → e.g., `int` from 0 to $n-1$
  - **Literal**: used for indexing → e.g., `int` from 0 to $2n-1$
  - **TruthValue**: three possibility (**true**, **false**, **undef**) → $\{1, 0, -1\}$
  - **Clause**: iterable list of literals
- Functions on variables
  - pos(**Variable**:x) ↦ **Literal** x (e.g., $2x+1$)
  - neg(**Variable**:x) ↦ **Literal** $\neg x$ (e.g., $2x$)
- Functions on literals
  - sign(**Literal**:l) ↦ $\{\mathbf{false}, \mathbf{true}\}$ (e.g., $l\%2$)
  - not(**Literal**:l) ↦ $\neg l$ (e.g., l^1)
  - var(**Literal**:l) ↦ x (e.g., $l/2$)

Data structures

- Data structures
  - model [**Variable** : x] $\mapsto$ **TruthValue** — stores the current truth value of x
  - clauses [**Literal** : l] $\mapsto$ [**Clause**,...] — list of clauses containing literal l
  - unit-literals — stack of true literals (efficient push(**Literal**:l) and **Literal**:back() and pop-back())
- Functions
  - val(**Variable**:x) $\mapsto$ **TruthValue** — truth value of variable x
  - falsified(**Literal**:l) $\mapsto$ Boolean — literal is falsified in model
  - satisfied(**Literal**:l) $\mapsto$ Boolean — literal is satisfied in model
- IN/OUT
  - Functions from-dimacs(**int**:d) $\mapsto$ **Literal** and to-dimacs(**Literal**:l) $\mapsto$ **int**
  - Functions read-dimacs() and write-dimacs()

Outline

Unit Propagation

- A clause forbids exactly one tuple

$$(\bar{x} \vee y \vee z \vee \bar{v} \vee \bar{w}) \iff \neg(x \wedge \bar{y} \wedge \bar{z} \wedge v \wedge w)$$

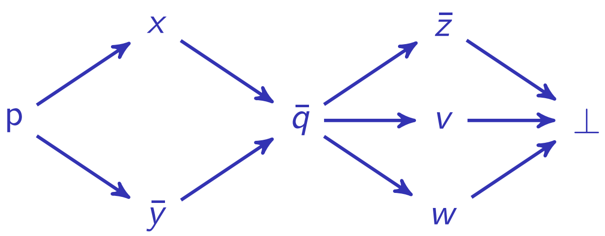
- What can we deduce by looking at just *one* clause?
- Nothing unless it is a unit clause (p): then we deduce that the literal p is true
  - x is true if $p = x$
  - x is false if $p = \bar{x}$
- If the clause has two (independent) literals, any one can be false, providing that the other is true
- Incomplete proof system (e.g. $(x \vee a) \wedge (\bar{x} \vee a) \wedge (\bar{y} \vee \bar{a}) \wedge (y \vee \bar{a})$)

Unit Propagation

- However it *propagates*: if we have the unit literal p, a clause containing $\bar{p}$ can be reduced, and maybe become unit, triggering more unit propagation

$$(\bar{x} \vee y \vee z \vee \bar{v} \vee \bar{w}) \wedge (\bar{p} \vee x) \wedge (\bar{p} \vee \bar{y}) \wedge (q \vee \bar{z}) \wedge (q \vee v) \wedge (p) \wedge (q \vee w) \wedge (\bar{q} \vee \bar{x} \vee y)$$

- (p) is a unit clause
- (x) and $(\bar{y})$
- $(\bar{q})$ is a unit clause
- $(\bar{z})$, (v) and (w) are unit clauses
- Unit propagation produces an empty clause

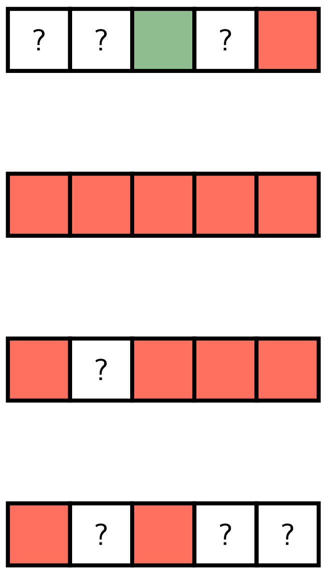


Unit Propagation

- Unit propagation solves *Horn*-SAT
- If a *Horn*-SAT formula has no unit clause, then every clause has at least one negative literal
  - The model with all variables false satisfies the formula
- Otherwise, unit propagate until reaching an inconsistency or a subformula without unit clauses

Implementing Unit Propagation

- A clause can either be:
  - Satisfied iff it contains at least one true literal
  - Falsified iff it contains *only* false literals
  - Unit iff it contains a single unknown literal, and $n-1$ false literals
  - Unresolved iff it contains no true literal and at least two unknown literals



Unit Propagation with counters

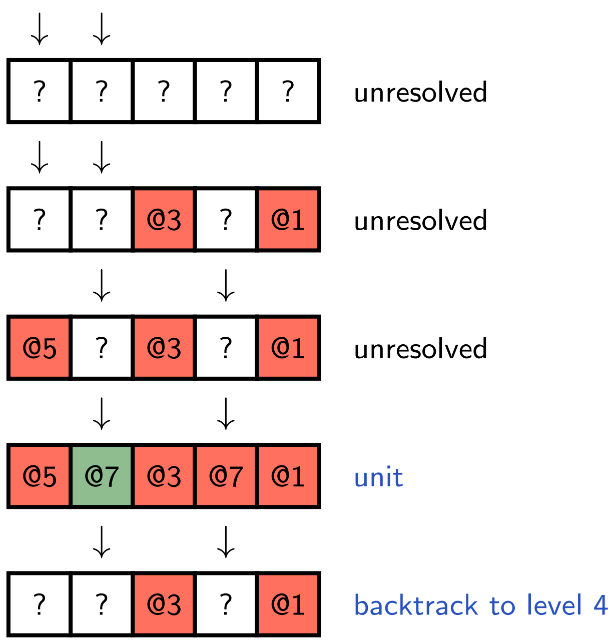
Unit propagation algorithm (counters)

Organise clauses per literals (*Clauses*(l) is the set of clauses containing literal l)
keep an initially null counter $\#f_i$ of false literals for each clause c_i
Put all unit clauses (*true literals*) in a list
while *There is a non-processed true literal l* **do**
    mark l as processed
    foreach $c_i \in Clauses(l)$ **do**
        increment $\#f_i$ // at most once per literal: $O(s)$
        if $\#f_i = |c_i|$ **then** return *FAIL*
        if $\#f_i = |c_i| - 1$ **then**
            find the last literal and add it to the list of true literals // $\Theta(|c_i|)$ at most once per clause: $O(s)$

- Let φ have n variables and m clauses, and let s be the total number of literals $s = \sum_{i=1}^{m} |c_i|$
- Worst case: every variable x is unit propagated (x if $|Clauses(x)| \geq |Clauses(\bar{x})|$, and $\bar{x}$ otherwise)
- Overall linear time $\Theta(s)$ amortized down a branch

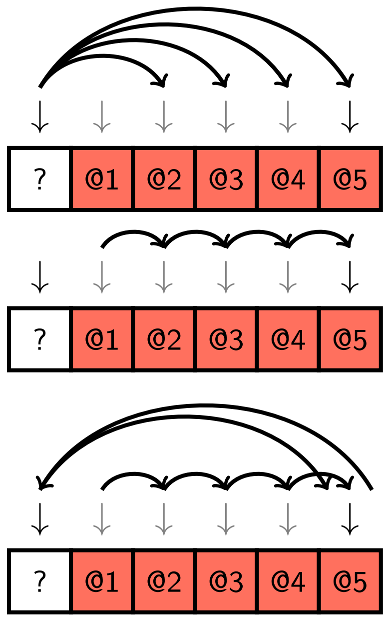
Watched Literals

- Invariant *Watch* only two *non-false* literals per clause
  - *Watch*(l) is the list of clauses that *watches* literal l
- Non-watched literals can become false, it *cannot make the clause unit or falsified* as long as two unknown literals remain
- When a watched literal become false, a replacement must be found
- When no replacement can be found, the clause is either unit or falsified
- *Nothing to do when backtracking*: the literals watched at level i cannot be false at level $i - 1$



Finding a new watched literals

- Scan the clause from first to last literal: possibly $\Theta(|c_i|)$ scans each costing $\Theta(|c_i|)$
  - Quadratic
- Store the initial position of the watch and scan *forward*
  - Linear but we must update the position of the watchers when *backtracking*
- Circular list: scan *forward*, but past the end and back to the *current* position
  - The clause is scanned at most twice: linear and no need to do anything when backtracking!



Average Complexity

- Let n be the number of variables, m be the number of clauses, $s = \sum_{i=1}^{m} |c_i|$ be the overall size of the formula, k be the number of true literals after unit propagation
- Consider first the clauses that unit propagated
  - They contain only variables among the k true literals
  - In order to propagate them, every literal must be explored (to increment the counter of find a new watched): it takes linear time in both cases call that $O(K)$
- Consider now the m' clauses that did not unit propagate (and let s' be their total size)
  - The counters algorithm increments the counters of every clause containing one of the k true literals
    - ★ The average number of clauses per literal is $\frac{s'}{n}$ so $\Theta\left(\frac{ks'}{n}\right)$ time in average
  - Overall: $\Theta(O(K) + \frac{ks'}{n})$ time
  - The watched algorithm increments finds a new wathed literal for each of the clauses that watch it
    - ★ A literal is watched by $\frac{m'}{n}$ of these clauses in average
    - ★ The probability that a random literal is not false is $\frac{n-k}{n}$, so the expected number of literals to scan to find a valid one to watch is $\frac{n}{n-k}$
  - Overall: $\Theta(O(K) + \frac{km'}{n-k})$ time

Additionnal structures

- Structure
  - watches [**Literal** : l] $\mapsto$ [**Clause**,...] — list of clauses watching literal l
  - **int**:to-propagate — the first non-unit-propagated literal in unit-literals
- Functions
  - get-rank(**Clause**:c, **Literal**:l) $\mapsto \{0,1\}$ — 0 if l is the first watched in c, 1 otherwise
  - get-index(**Clause**:c, $\{0,1\}$:r) $\mapsto$ **int** — index of the $(r+1)$-th watched in c
  - set-watcher(**Clause**:c, **Literal**:l, $\{0,1\}$:r) — set l as $(r+1)$-th watcher of c
  - assign(**Literal**:l) — push l onto unit-literals and set model [var(l)]

Algorithm

Unit propagation algorithm (watched literals)

Algorithm: unit-propagate()

```
while to-propagate < |unit-literals| do
    l ← not(unit-literals [to-propagate ])
    if not unit-propagate(l) then
        return false
    to-propagate ← to-propagate + 1
return true
```

Algorithm: unit-propagate(l)

Input: A non-unit propagated false literal l
Output: false in case of a contradiction, true otherwise

```
foreach c ∈ clauses[l] do
    r ← get-rank(c, l); start ← i ← get-index(c, r)
    p ← c[get-index(c, 1-r)]
    if not satisfied(p) then
        while true do
            i ← i + 1
            if i = |c| then i ← 0
            if i = start then break
            if c[i] ≠ p then
                if not falsified(c[i]) then
                    set-watcher(c, c[i], r)
                    break
        if i = start then
            if falsified(p) then return false
            assign(p)
return true
```

- Resolution rule: [DP60,R65]

$$\frac{(\alpha \lor x) \qquad\qquad (\beta \lor \bar{x})}{(\alpha \lor \beta)}$$

  - Complete proof system for propositional logic: If the formula φ is not satisfiable, then there is sequence of resolution steps that produce the *empty clause* $\bot$

$$\begin{tikzcd}[column sep=small]
(x \lor a) \arrow[d, dash] & (\bar{x} \lor a) \arrow[dl, dash] & (\bar{y} \lor \bar{a}) \arrow[dr, dash] & (y \lor \bar{a}) \arrow[d, dash] \\
(a) \arrow[dr, dash] & & & (\bar{a}) \arrow[dll, dash] \\
& \bot & &
\end{tikzcd}$$

- Self-subsuming resolution (with $\alpha' \subseteq \alpha$): [e.g. SP04,EB05]

$$\frac{(\alpha \lor x) \qquad\qquad (\alpha' \lor \bar{x})}{(\alpha)}$$

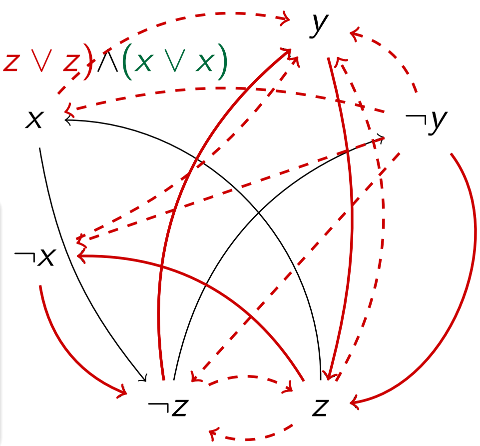
  - (α) subsumes $(\alpha \lor x)$

Theorem

Resolution solves 2-SAT in polynomial time

- Resolvant is a complete refutation system for SAT (and hence for 2-SAT)
- Resolvant clauses have at most 2 literals
  - There are at most n^2 binary clauses

$(\neg y \vee z) \wedge (\neg z \vee \neg x) \wedge (x \vee \neg z) \wedge (\neg z \vee \neg z) \wedge (y \vee y) \wedge (y \vee z) \wedge (z \vee z) \wedge (x \vee x)$



Algorithm

- $x \vee y$ is equivalent to $\neg x \implies y$ and $\neg y \implies x$
- Add transitive edges
  - If there is an inconsistency, then the formula is not satisfiable
  - If not, it is satisfiable, because the choice $x \implies \neg x$ closes a cycle only if there is a path $\neg x \implies x$

- SAT is in **NP**: if an instance is satisfiable, it is possible to prove it efficiently
  - Just show a model and check clause by clause that is it correct (it is a certificate)
- What about the question "is φ *unsatisfiable*?", or "is φ a *tautology*?"
  - There might not exist short certificates for problems in **coNP**, but we can provide a *long* one
- Proof system: maps to every *unsatisfiable* formula φ a refutation R
  - There is a polynomial algorithm (in $|R|$) to check the refutation proof
    - ★ *Pebbling* formulas

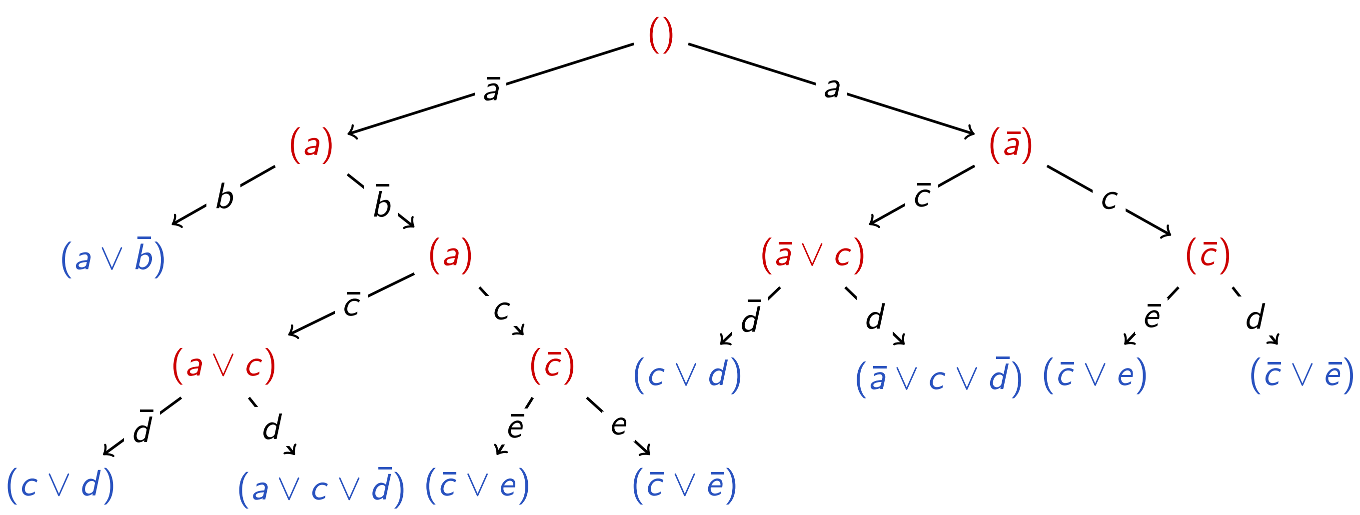
Resolution as a proof system

$$\varphi = (a \vee \neg b) \wedge (\neg a \vee c \vee \neg d) \wedge (a \vee c \vee \neg d) \wedge (\neg c \vee \neg e) \wedge (\neg c \vee e) \wedge (c \vee d)$$

$$\begin{array}{lcll}
c_1 & = & (\neg c \vee e) & \in \varphi \\
c_2 & = & (\neg c \vee \neg e) & \in \varphi \\
c_3 & = & (\neg c) & \text{resolvant of } c_1 \text{ and } c_2 \\
c_4 & = & (a \vee c \vee \neg d) & \in \varphi \\
c_5 & = & (\neg a \vee c \vee \neg d) & \in \varphi \\
c_6 & = & (c \vee \neg d) & \text{resolvant of } c_4 \text{ and } c_5 \\
c_7 & = & (c \vee d) & \in \varphi \\
c_8 & = & (c) & \text{resolvant of } c_6 \text{ and } c_7 \\
c_9 & = & () & \text{resolvant of } c_3 \text{ and } c_8
\end{array}$$

Resolution proofs from tree search

$$\varphi = (a \vee \neg b) \wedge (\neg a \vee c \vee \neg d) \wedge (a \vee c \vee \neg d) \wedge (\neg c \vee \neg e) \wedge (\neg c \vee e) \wedge (c \vee d)$$



Resolution is sound and complete

- *Soundness*: if there exists a resolution refutation then the formula is unsatisfiable
  - Resolution is a *sound* proof system simply because the resolution step is sound
- *Completeness*: if a formula is unsatisfiable then there exists a resolution refutation of that formula
  - Tree search is obviously a complete proof system
  - To every search tree we can associate a resolution proof
  - Therefore resolution is a *complete* proof system

Resolution: conciseness

- What does make a proof system good? (besides soundness and completeness)
- A good proof system is one that allows shorter proofs
  - If refutations are polynomial size in general, then $\mathbf{NP} = \mathbf{coNP}$
- For any tree search refutation, there is a resolution refutation of same size
- There exist formulas with short resolution refutation but *exponential* tree search refutations

Pigeon Hole Principle

If $m > n$ there is no injective mapping of m objects onto n

$PHP^{m \rightarrow n}$:	$(x_{1,1} \vee x_{1,2} \vee \ldots \vee x_{1,n}) \wedge$	Pigeon 1 needs a hole
	$\ldots$	
	$(x_{m,1} \vee x_{m,2} \vee \ldots \vee x_{m,n}) \wedge$	Pigeon m needs a hole
	$\bigwedge_{1 \leq i < j \leq m} (\bar{x_{i,1}} \vee \bar{x_{j,1}}) \wedge$	Hole 1 can contain at most 1 pigeon
	$\ldots$	
	$\bigwedge_{1 \leq i < j \leq m} (\bar{x_{i,n}} \vee \bar{x_{j,n}})$	Hole n can contain at most 1 pigeon

- Resolution refutations of the pigeon hole principle are exponential
- Using induction, for instance, one can make a linear size refutation