CDM

Algorithms for Propositional Logic

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/∼sutner

Battleplan

- Normal Forms
- SAT Solvers
- Davis/Putnam Algorithm
- Refutation and Resolution

Normal Forms

Decision Problems

Recall the decision problems associated with propositional formulae.

Problem: **Tautology**
Instance: A propositional formula φ.
Question: Is φ a tautology?

Problem: **Satisfiability**
Instance: A propositional formula φ.
Question: Is φ a contingency?

How can we go about solving these problems, using smarter methods than just brute force (compute the truth table)?

Preprocessing

It seems reasonable to preprocess the input a bit: we can try to bring the input into some particularly useful syntactic form, a form that can then be exploited by the algorithm.

There are two issues:

- The transformation must produce an equivalent formula (but see below about possible additions to the set of variables).
- The transformation should be fast.

Here are some standard methods to transform propositional formulae.

We will focus on formulae using $\neg, \wedge, \vee$, a harmless assumptions since we can easily eliminate all implications and biconditionals.

Negation Normal Form (NNF)

Definition 1. *A formula is in* Negation Normal Form (NNF) *if it only contains negations applied directly to variables.*

A literal *is a variable or a negated variable.*

Theorem 1. *For every formula, there is an equivalent formula in NNF.*

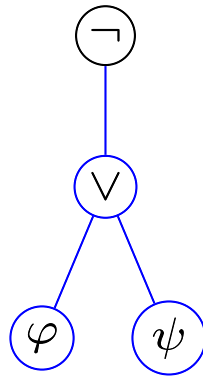
The algorithm pushes all negation signs inward, until they occur only next to a variable.

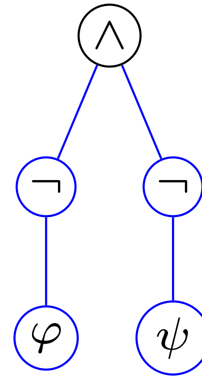
Rules for this transformation:

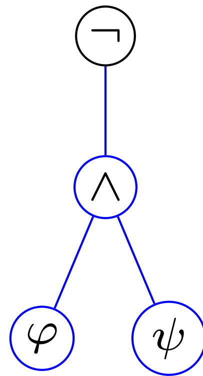
$$\begin{aligned} \neg(\varphi \wedge \psi) &\Rightarrow \neg\varphi \vee \neg\psi \\ \neg(\varphi \vee \psi) &\Rightarrow \neg\varphi \wedge \neg\psi \\ \neg\neg\varphi &\Rightarrow \varphi \end{aligned}$$

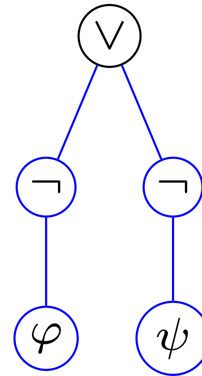
As Parse Trees

Push negations down to the leaves.









Rewrite Systems

This is an example of a *rewrite system*: replace the LHS by the RHS of a substitution rule, over and over, until a fixed point is reached.

Note that this requires pattern matching: we have to find the handles in the given expression.

$$\neg(p \wedge (q \vee \neg r) \wedge s \wedge \neg t)$$

$$\neg p \vee \neg(q \vee \neg r) \vee \neg s \vee t$$

$$\neg p \vee (\neg q \wedge r) \vee \neg s \vee t$$

Also note that the answer is not unique, here are some other NNFs for the same formula:

$$\neg p \vee \neg s \vee t \vee (\neg q \wedge r)$$

$$(\neg p \vee \neg s \vee t \vee \neg q) \wedge (\neg p \vee \neg s \vee t \vee r)$$

Conversion to NNF is a search problem, not a function problem.

Min- and Max-Terms

So we are left with a formula built from literals using connectives $\wedge$ and $\vee$. The most elementary such formulae have special names.

Definition 2.
A minterm *is a conjunction of literals.*
A maxterm *is a disjunction of literals.*

Note that a truth table is essentially a listing of all possible 2^n full minterms over some fixed variables $x_1, x_2, \ldots, x_n$ combined with the corresponding truth values of the formula $\varphi(x_1, \ldots, x_n)$.

By forming a disjunction of the minterms for which the formula is true we get a (rather clumsy) normal form representation of the formula.

The reason we are referring to the normal form as clumsy is that it contains many redundancies in general. Still, the idea is very important and warrants a definition.

Disjunctive Normal Form

Definition 3. *A formula is in* Disjunctive Normal Form (DNF) *if it is a disjunction of minterms (conjunctions of literals).*

In other words, a DNF formula is a "sum of products" and looks like so:

$$(x_{11} \land x_{12} \land \ldots \land x_{1n_1}) \lor (x_{21} \land \ldots \land x_{2n_2}) \lor \ldots \qquad \ldots \lor (x_{m1} \land \ldots \land x_{mn_m})$$

where each x_{ij} is a literal.

In short: $\bigvee_i \bigwedge_j x_{ij}$.

If you think of the formula as a circuit DNF means that there are two layers: an OR gate on top, AND gates below Note that this only works if we assume unbounded fan-in and disregard negation.

Conversion to DNF

Theorem 2. *For every formula, there is an equivalent formula in DNF.*

Step 1: First bring the formula into NNF.

Step 2: Then use the rewrite rules

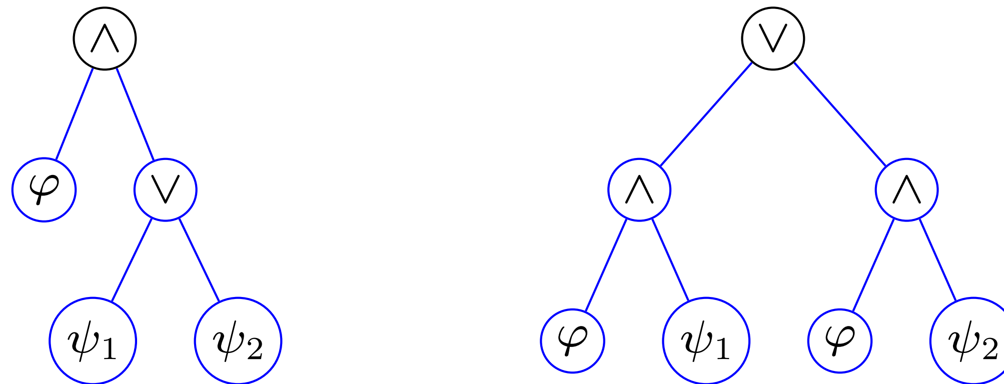
$$\varphi \wedge (\psi_1 \vee \psi_2) \;\Rightarrow\; (\varphi \wedge \psi_1) \vee (\varphi \wedge \psi_2)$$
$$(\psi_1 \vee \psi_2) \wedge \varphi \;\Rightarrow\; (\psi_1 \wedge \varphi) \vee (\psi_2 \wedge \varphi)$$

□

Exercise 1. *Prove that these rules really produce a DNF. What is the complexity of this algorithm?*

Expression Trees

Here is the corresponding operation in terms of the expression tree.



Note that we have created a second copy of φ (though in an actual algorithm we can avoid duplication by sharing subexpressions; nonetheless the ultimate output will be large).

Example

First conversion to NNF.

$$\neg(p \vee (q \wedge \neg r) \vee s \vee \neg t$$

$$\neg p \wedge \neg(q \wedge \neg r) \wedge \neg s \wedge t$$

$$\neg p \wedge (\neg q \vee r) \wedge \neg s \wedge t$$

Reorder, and then distribute $\wedge$ over $\vee$.

$$\neg p \wedge \neg s \wedge t \wedge (\neg q \vee r)$$

$$(\neg p \wedge \neg s \wedge t \wedge \neg q) \vee (\neg p \wedge \neg s \wedge t \wedge r)$$

Complexity

Computationally, there is one crucial difference between conversion to NNF and conversion to DNF:

- The size of the formula in NNF is linear in the size of the input.
- The size of the formula in DNF is not polynomially bounded by the size of the input.

Thus, even if we have a perfect linear time implementation of the rewrite process, we still wind up with an exponential algorithm.

Exercise 2. *Construct an example where conversion to DNF causes exponential blow-up.*

DNF and Truth Tables

One reason DNF is natural is that one can easily read off a canonical DNF for a formula if we have a truth table for it.

For n variables, the first n columns determine 2^n full minterms (containing each variable either straight or negated).

$$10011 \qquad \Rightarrow \qquad x_1\overline{x}_2\overline{x}_3x_4x_5$$

Select those rows where the formula is true, and collect all the corresponding minterms into a big disjunction.

Done!

Note the resulting formula has $O(2^n)$ conjunctions of n literals each.

Example

Unfortunately, this approach may not produce optimal results: for $p \vee q$ we get

p	q	$(p \vee q)$
0	0	0
0	1	1
1	0	1
1	1	1

So brute force application of this method yields 3 full minterms:

$$(p \wedge \neg q) \vee (\neg p \wedge q) \vee (p \wedge q)$$

Clearly, we need some simplification process. More about this later in the discussion of resolution.

Remember?

This is essentially the same method we used to get the expressions for sum and carry in the 2-bit adder.

In a Boolean algebra, one talks about sums of products of literals instead of DNF.

Hence, any Boolean expression can be written in sum-of-products form.

Is there also a product-of-sums representation?

Sure . . .

Unlike with ordinary arithmetic, in Boolean algebra there is complete symmetry between meet and join.

Conjunctive Normal Form (CNF)

Definition 4. *A formula is in* Conjunctive Normal Form (CNF) *if it is a conjunction of maxterms (disjunctions of literals).*

The maxterms are often referred to as *clauses* in this context. So, a formula in CNF looks like

$$\bigwedge_i \bigvee_j x_{ij}.$$

Theorem 3. *For every formula, there is an equivalent formula in CNF.*

Again start with NNF, but now use the rules

$$\varphi \vee (\psi_1 \wedge \psi_2) \;\Rightarrow\; (\varphi \vee \psi_1) \wedge (\varphi \vee \psi_2)$$
$$(\psi_1 \wedge \psi_2) \vee \varphi \;\Rightarrow\; (\psi_1 \vee \varphi) \wedge (\psi_2 \vee \varphi)$$

Blow-Up

The formula

$$\varphi = (p_{10} \land p_{11}) \lor (p_{20} \land p_{21}) \lor \dots (p_{n0} \land p_{n1})$$

is in DNF, but conversion to CNF using our rewrite rule produces the exponentially larger formula

$$\varphi \equiv \bigwedge_{f:[n]\to\mathbf{2}} \bigvee_{i\in[n]} p_{if(i)}$$

Exercise 3. *Show that there is no small CNF for φ: the 2^n disjunctions of length n must all appear.*

Are CNF-based (or DNF-based) algorithms then useless in practice?

Tseitin's Trick

No, but we have to be careful with the conversion process to control blow-up. Instead of preserving the underlying set of propositional variables, we extend it by a new variable q_ψ for each subformula ψ of ϕ.

For a propositional variable p we let $q_p = p$ and introduce a clause $\{p\}$. Otherwise we introduce clauses as follows:

$$
\begin{aligned}
q_{\neg\psi} &: \{q_\psi, q_{\neg\psi}\}, \{\neg q_\psi, \neg q_{\neg\psi}\} \\
q_{\psi\vee\varphi} &: \{\neg q_\psi, q_{\psi\vee\varphi}\}, \{\neg q_\varphi, q_{\psi\vee\varphi}\}, \{\neg q_{\psi\vee\varphi}, q_\varphi, q_\psi\} \\
q_{\psi\wedge\varphi} &: \{q_\psi, \neg q_{\psi\wedge\varphi}\}, \{q_\varphi, \neg q_{\psi\wedge\varphi}\}, \{\neg q_\psi, \neg q_\varphi, q_{\psi\wedge\varphi}\}
\end{aligned}
$$

The intended meaning of q_ψ is pinned down by these clauses, e.g.

$$q_{\psi\vee\varphi} \equiv q_\psi \vee q_\varphi$$

Example

Consider again the formula

$$\varphi = (p_{10} \wedge p_{11}) \vee (p_{20} \wedge p_{21}) \vee \ldots (p_{n0} \wedge p_{n1})$$

Set $B_k = (p_{k0} \wedge p_{k1})$ and $A_k = B_k \vee B_{k+1} \vee \ldots B_n$ for $k = 1, \ldots, n$. Thus, $\varphi = A_1$ and all the subformulae other than variables are of the form A_k or B_k.

The the clauses in the Tseitin form of φ are as follows (we ignore the variables):

- q_{A_k}: $\left\{q_{B_k}, \neg q_{B_k \wedge A_{k-1}}\right\}, \left\{q_{A_{k-1}}, \neg q_{B_k \wedge A_{k-1}}\right\}, \left\{\neg q_{B_k}, \neg q_{A_{k-1}}, q_{B_k \wedge A_{k-1}}\right\}$
- q_{B_k}: $\{\neg p_{k0}, q_{B_k}\}, \{\neg p_{k1}, q_{B_k}\}, \{\neg q_{B_k}, p_{k1}, p_{k0}\}$

Exercise 4. *Make sure you understand in the example how any satisfying assignment to φ extends to a satisfying assignment of the Tseitin CNF, and conversely.*

Preserving Satisfiability

Theorem 4. *The set C of clauses in Tseitin CNF is satisfiable if, and only if, ϕ is so satisfiable. Moreover, C can be constructed in time linear in the size of ϕ.*

Proof.

$\Rightarrow$: Suppose that $\sigma \models C$.

An easy induction shows that for any subformula ψ we have $[\![\psi]\!]_\sigma = [\![q_\psi]\!]_\sigma$. Hence $[\![\phi]\!]_\sigma = [\![q_\phi]\!]_\sigma = 1$ since $\{q_\phi\}$ is a clause in C.

$\Leftarrow$: Assume that $\sigma \models \phi$.

Define a new valuation τ by $\tau(q_\psi) = [\![\psi]\!]_\sigma$ for all subformulae ψ. It is easy to check that $\tau \models C$.

$\square$

Knights in Shiny Armor

Exercise 5. *Construct a formula* Φ_n *that is satisfiable if, and only if, the* $n \times n$ *chessboard has the property that a knight can reach all squares by a sequence of admissible moves.*

What would your formula look like in CNF and DNF?

Exercise 6. *Construct a formula* Φ_n *that is satisfiable if, and only if, the* $n \times n$ *chessboard admits a knight's tour: a sequence of admissible moves that touches each square exactly once.*

Again, what would your formula look like in CNF and DNF?

Some Questions

Exercise 7. *How hard is it to convert a formula to CNF?*

Exercise 8. *Show how to convert directly between DNF and CNF.*

Exercise 9. *Show: In CNF, if a clause contains x and $\overline{x}$, then we can remove the whole clause and obtain an equivalent formula.*

Exercise 10. *Suppose a formula is in CNF.*
How hard is it to check if the formula is a tautology?

Exercise 11. *Suppose a formula is in DNF.*
How hard is it to check if the formula is a tautology?

Exercise 12. *How about checking whether a formula in DNF (or CNF) is a contradiction?*

SAT Solvers

Satisfiability Testing

Truth tables allow one to check any property (tautology, contradiction, satisfiability). But: the table has exponential size and so is useless in practice where one often deals with formulae with thousands of variables.

Recall that Satisfiability testing is enough in the sense that

- φ is a tautology iff $\neg\varphi$ is not satisfiable.
- φ is a contradiction iff $\neg\varphi$ is a tautology iff φ is not satisfiable.

Note that these equivalences coexist uneasily with normal forms. For example, if φ is in CNF then $\neg\varphi$ can be easily converted into DNF, but CNF is far off.

So for algorithms that depend on a specific form of the input there may be a problem if conversion is slow.

Normal Forms and Meaning

It should be noted that CNF and DNF are not particularly useful for a human being when it comes to understanding the meaning of a formula (NNF is not quite as bad). But that's not their purpose: they provide a handle for specialized algorithms to test validity and satisfiability. We'll focus on the latter.

First note that one can perform various cleanup operations without affecting satisfiability in CNF.

- We can delete any clause that contains a literal and its negation.
- We can delete any clause that contains another clause (as a subset).

The last step is justified by the equivalence

$$\varphi \wedge (\varphi \vee \psi) \equiv \varphi$$

Example: CNF Tautology Testing

Here is a very small example. We verify that Peirce's Law

$$((A \to B) \to A) \to A$$

is a tautology. Rewriting the implications we get

$$\neg(\neg(\neg A \lor B) \lor A) \lor A$$

which turns into

$$(\neg A \lor B \lor A) \land (\neg A \lor A)$$

By the first simplification rule we are done.

SAT Algorithms

There is an old, but surprisingly powerful satisfiability testing algorithm due to *Davis and Putnam*, originally published in 1960. Modern versions of the algorithm (some of them commercial and proprietary) are still widely used today.

The Davis/Putnam Paper

It is worth noting that the original paper goes by the title

A Computing Procedure for Quantification Theory

Thus, the real target is predicte logic (first order logic) rather than propositional logic. They usa a refutation method based on Herbrand universes. The method produces a sequence of larger and larger propositional formulae obtained from the negation of the given formula, that each must be tested for satisfiability. If a non-satisfiable formula appears the algorithm terminates (in which case the original formula is proven valid), otherwise it continues indefinitely. Each round employs what is now the classical Davis/Putnam method.

As the authors point out, their method yielded a result in a 30 minute hand-computation where another algorithm running on an IBM 704 failed after 21 minutes. The variant presented below was first implemented by Davis, Longman and Loveland in 1962 on an IBM 704.

The Main Idea

The basic idea of the propositional part of the algorithm is beautifully simple:

DPLL assumes that the input formula is in CNF and performs certain simple cleanup operations – until they apply no longer.

Then it bites the bullet: it picks a variable and explicitly tries to set it to "true" and "false", respectively.

Recurse.

The wary algorithm designer will immediately suspect exponential behavior, but as it turns out in many practical cases the algorithm performs very well.

Davis/Putnam Algorithm

Suppose the formula φ is given in CNF. We are trying to solve the decision problem Satisfiability.

In this context, the disjunctions in CNF are often called *clauses*. Since the order of terms in a clause does not matter, one usually writes them as sets of literals.

So a whole formula in CNF might be written as

$$\{x, \overline{y}, u\}\ ,\ \{\overline{x}, y, u\}\ ,\ \{x, \overline{u}\}$$

A clause is a *unit clause* iff it contains just one literal.

Note that an *empty clause* corresponds to $\bot$: there are no literals that one could try to set to a truth value that would render the whole clause true.

Unit Clause Elimination (UCE)

Unit clauses are easy to deal with:

The only way to satisfy a single literal x is by setting $\sigma(x) = 1$.

Note that once we decide $\sigma(x) = 1$, we can perform

- Unit Subsumption: delete all clauses containing x, and
- Unit Resolution: remove $\overline{x}$ from all remaining clauses.

This process is called *unit clause elimination*.

The crucial point is: a CNF formula φ containing unit clause $\{x\}$ is satisfiable iff there is an assignment σ setting x to true, and satisfying φ' obtained from φ by UCE.

Pure Literal Elimination (PLE)

A *pure literal* in a CNF formula is a literal that occurs only directly, but not negated. So the formula may contain a variable x but not $\overline{x}$ or, conversely, only $\overline{x}$ but not x.

Clearly, if the formula contains x but not $\overline{x}$ we can simply set $\sigma(x) = 1$ and remove all the clauses containing the variable.

Likewise, if the formula contains $\overline{x}$ but not x we can set $\sigma(x) = 0$ and remove all clauses containing the negated variable.

This may sound utterly trivial, but note that in order to do PLE efficiently we should probably keep a counter for the total number of occurrences of both x and $\overline{x}$ for each variable.

More on PLE

Here is a closer look at PLE. Let Φ be in CNF, x a variable. Define

- Φ_x^+: the clauses of Φ that contain x positively,
- Φ_x^-: the clauses of Φ that contain x negatively, and
- Φ_x^*: the clauses of Φ that are free of x.

So we have the partition

$$\Phi = \Phi_x^+ \cup \Phi_x^- \cup \Phi_x^*$$

This partition gives rise to a trie data structure.

PLE Lemma

Proposition 1. *If* Φ_x^+ *or* Φ_x^- *is empty then* Φ *is equivalent to* Φ_x^*.

In other words, one can replace Φ by Φ_x^*: Φ is satisfiable iff Φ_x^* is satisfiable.

Since Φ_x^* is smaller than Φ (unless x does not appear at all) this step simplifies the problem of deciding satisfiability.

Of course, we get stuck when all variables have positive and negative occurrences.

The DPLL Algorithm

- Perform unit clause elimination until no unit clauses are left.
- Perform pure literal elimination, call the result ψ.
- If an empty clause has appeared, return false.
- If all clauses have been eliminated, return true.
- Splitting: otherwise, cleverly pick one of the remaining literals, x. Recursively test *both*

  $$\psi, \{x\} \qquad \text{and} \qquad \psi, \{\overline{x}\}$$

  for satisfiability.
  Return true if at least one of the branches returns true, false otherwise.

So this is dangerously close to brute-force search. The algorithm still succeeds beautifully in the real world since it systematically exploits all possibilities to prune irrelevant parts of the search tree.

Example

After three unit clause elimination steps (no pure literal elimination) and one split on d we get the answer "satisfiable":

```
1    {a,b,c} {a,!b} {a,!c} {c,b} {!a,d,e} {!b}
2    {a,c}          {a,!c} {c}   {!a,d,e}
3    {a}            {a}          {!a,d,e}
4                                {d,e}
5    -
```

We could also have used pure literal elimination (on d):

```
1    {a,b,c} {a,!b} {a,!c} {c,b} {!a,d,e} {!b}
2    {a,b,c} {a,!b} {a,!c} {c,b}          {!b}
3    {a,c}          {a,!c} {c}
4                   {a}
5    -
```

Finding an Assignment

Note that this algorithm implicitly also solves the *search problem*: we only need to keep track of the assignments made to literals. In the example, the corresponding assignment is

$$\sigma(b) = 0, \sigma(c) = \sigma(a) = \sigma(d) = 1$$

The choice for e does not matter.

Note that we also could have chosen $\sigma(e) = 1$ and ignored d.

Exercise 13. *Implement a version of the algorithm that returns a satisfying truth assignment if it exists.*

How about all satisfying truth assignments?

Correctness

Claim. *The Davis/Putnam algorithm is correct: it returns true if, and only if, the input formula is satisfiable.*

Proof.

Suppose φ is in CNF and has a unit clause $\{x\}$. Then φ is satisfiable iff there is a satisfying truth assignment σ such that $\sigma(x) = 1$.

But then $\sigma(C) = 1$ for any clause containing x, so Unit Subsumption does not affect satisfiability. Also, $\sigma(C) = \sigma(C')$ for any clause containing $\overline{x}$, where C' denotes the removal of literal $\overline{x}$. Hence Unit Resolution does not affect satisfiability either.

Suppose z is a pure literal. If σ satisfies φ then σ' also satisfies φ where

$$\sigma(u) = \begin{cases} \sigma(u) & \text{if } u = z, \\ 1 & \text{otherwise.} \end{cases}$$

Correctness, contd.

Let x be any literal in φ. Then by Shannon expansion

$$\varphi \equiv (x \wedge \varphi[1/x]) \vee (\neg x \wedge \varphi[0/x])$$

But splitting checks exactly the two formulae on the right for satisfiability; hence φ is satisfiable if, and only if, at least one of the two branches returns true.

Termination is obvious.

$\Box$

Note that in Splitting there usually are many choices for x. This provides an opportunity to use clever heuristics to speed things up. One plausible strategy is to pick the most frequent literal. Why?

Davis/Putnam In Practice

Bad News: DPLL may take exponential time!

In practice, though, Davis/Putnam is usually quite fast.

It is not entirely understood why formulae that appear in real-world problems tend to produce only polynomial running time when tackled by Davis/Putnam.

Take the notion of "real world" here with a grain of salt. For example, in algebra DPLL has been used to solve problems in the theory of so-called quasi groups (cancellative groupoids). In a typical case there are n^3 Boolean variables and about n^4 to n^6 clauses; n might be 10 or 20.

The point is that instances seem to have to be maliciously constructed to make DPLL perform poorly.

Example: Exactly One

Neither UCE nor PLE applies here, so the first step is a split.

```
{{!a,!b},{!a,!c},{!a,!d},{!a,!e},{!b,!c},{!b,!d},{!b,!e},{!c,!d},
  {!c,!e},{!d,!e},{a,b,c,d,e}}

{{!a},{!a,!b},{!a,!c},{!a,!d},{!a,!e},{!b,!c},{!b,!d},{!b,!e},
  {!c,!d},{!c,!e},{!d,!e},{a,b,c,d,e}}

{{!b},{!b,!c},{!b,!d},{!b,!e},{!c,!d},{!c,!e},{!d,!e},{b,c,d,e}}

{{!c},{!c,!d},{!c,!e},{!d,!e},{c,d,e}}

{{d},{d,e},{!d,!e}}

True
```

Of course, this formula is trivially satisfiable, but note how the algorithm quickly homes in on one possible assignment.

The Real World

If you want to see some cutting edge problems that can be solved by SAT algorithms (or can't quite be solved at present) take a look at

`http://www.satcompetition.org`

`http://www.satlive.org`

Try to implement DPLL yourself, you will see that it's quite hard to get up to the level of the programs that win these competitions.

Refutation and Resolution

Refutation

In summary, DPLL is a practical and powerful method to tackle fairly large instances of Satisfiability (many thousands of variables).

The main idea in DPLL is to organize the search for a satisfying truth-assignment in a way that often circumvents the potentially exponential blow-up.

Here is a crazy idea: How about the opposite approach?

How about systematically trying to show there cannot be satisfying truth-assignment?

As with DPLL, the procedure should be usually fast, but on occasion may blow-up exponentially.

Resolvents

Suppose x is a variable that appears in clause C and appears negated in clause C':

$$C = \{x, y_1, \ldots, y_k\} \qquad C' = \{\overline{x}, z_1, \ldots, z_l\}$$

Then we can introduce an new clause, a *resolvent* of C and C'

$$D = \{y_1, \ldots, y_k, z_1, \ldots, z_l\}$$

Proposition 2. *$C \wedge C'$ is equivalent to $C \wedge C' \wedge D$.*

We write $\text{Res}(C, C') = D$, but note that there may be several resolvents.

Example

The CNF formula

$$\varphi = \{\{x, \overline{y}, \overline{z}\}, \{\overline{x}, \overline{y}\}, \{\overline{y}, z\}, \{y\}\}$$

admits the following ways to compute resolvents:

$$\begin{aligned} \mathsf{Res}(\{x, \overline{y}, \overline{z}\}, \{\overline{x}, \overline{y}\}) &= \{\overline{y}, \overline{z}\} \\ \mathsf{Res}(\{\overline{y}, z\}, \{y\}) &= \{z\} \\ \mathsf{Res}(\{\overline{y}, \overline{z}\}, \{z\}) &= \{\overline{y}\} \\ \mathsf{Res}(\{y\}, \{\overline{y}\}) &= \emptyset \end{aligned}$$

This iterative computation of resolvents is called *resolution*.

The last resolvent corresponds to the empty clause, indicating that the original formula is not satisfiable.

(Bad) Notation

It is a sacred principle that in the context of resolution methods one writes the empty clause thus:

Yep, that's a little box, like the end-of-proof symbol or the necessity operator in modal logic. Grin and bear it.

As we will see shortly, $\Box$ is a resolvent of Φ if, and only if, the formula is a contradiction.

Resolution

More precisely, given a collection of clauses Φ, let $\mathsf{Res}(\Phi)$ be the collection of all resolvents of clauses in Φ plus Φ itself.

Set

$$\mathsf{Res}^{\star}(\Phi) = \bigcup_{n} \mathsf{Res}^{n}(\Phi)$$

so $\mathsf{Res}^{\star}(\Phi)$ is the least fixed point of the resolvent operator applied to Φ.

We will show that

Lemma 1. Φ *is a contradiction if, and only if,* $\Box \in \mathsf{Res}^{\star}(\Phi)$.

DAG Perspective

One often speaks of a *resolution proof* for the un-satisfiability of Φ as a directed, acyclic graph G whose nodes are clauses. The following degree conditions hold:

- The clauses of Φ have indegree 0.
- Each other node has indegree 2 and corresponds to a resolvent of the two predecessors.
- There is one node with outdegree 0 corresponding to the empty clause.

The graph is in general not a tree, just a DAG, since nodes may have outdegrees larger than 1 (a single clause can be used in together with several others to produce resolvents).

Other than that, you can think of G as a tree with the clauses of Φ at the leaves, and $\Box$ at the root.

Exercise

Just to familiarize yourself with resolutions, show the following.

Suppose we have a resolution proof for some contradiction Φ.

For any truth-assignment σ there is a uniquely determined path from a clause of Φ to the root $\Box$ such that for any clause C along that path we have: $\sigma(C) = 0$.

Proof?

Where to start?

More Resolution

There are two issues we have to address:

- Correctness: if a formula has $\Box$ as a resolvent then it is indeed a contradiction.
- Completeness: if a formula is a contradiction then it has $\Box$ as a resolvent.

Note that for a practical algorithm the last condition is actually a bit too weak: there are many possible ways to construct a resolution proof, since we do not know ahead of time which method will succeed we need some kind of robustness: it should not matter too much which clauses we resolve first.

On the upside, note that we do not necessarily have to compute all of $\text{Res}^\star(\Phi)$: if $\Box$ pops up we can immediately terminate.

Correctness

Lemma 2. *For any truth-assignment σ we have*

$$\sigma(C) = \sigma(C') = 1 \qquad \textit{implies} \qquad \sigma(\mathrm{Res}(C, C')) = 1$$

Proof.

If $\sigma(y_i) = 1$ for some i we are done, so suppose $\sigma(y_i) = 0$ for all i.

Since σ satisfies C we must have $\sigma(x) = 1$. But then $\sigma(\overline{x}) = 0$ and thus $\sigma(z_i) = 1$ for some i.

Hence σ satisfies $\mathrm{Res}(C, C')$. $\Box$

It follows by induction that if σ satisfies Φ it satisfies all resolvents of Φ.

Hence resolution is correct: only contradictions will produce $\Box$.

Completeness

Theorem 5. *Resolution is complete.*

Proof.

By induction on the number n of variables.

We have to show that if Φ is a contradiction then $\Box \in \mathrm{Res}^{\star}(\Phi)$.

$n = 1$: Then $\Phi = \{x\}, \{\overline{x}\}$.

In one resolution step we obtain $\Box$.

Done.

Proof contd.

Assume $n > 1$ and let x be a variable.

Let Φ_0 and Φ_1 be obtained by performing unit clause elimination for $\{x\}$ and $\{\overline{x}\}$.

Note that both Φ_0 and Φ_1 must be contradictions.

Hence by IH $\Box \in \text{Res}^\star(\Phi_i)$.

Now the crucial step: by repeating the "same" resolution proof with Φ rather than Φ_i, $i = 0, 1$, we get $\Box \in \text{Res}^\star(\Phi)$ if this proof does not use any of the mutilated clauses.

Otherwise, if mutilated clauses are used in both cases, we must have

- $\{x\} \in \text{Res}^\star(\Phi)$ from Φ_1, and
- $\{\overline{x}\} \in \text{Res}^\star(\Phi)$ from Φ_0.

Hence $\Box \in \text{Res}^\star(\Phi)$.

$\Box$

A Simple Algorithm

It is clear that we would like to keep the number of resolvents introduced in the resolution process small. Let's say that clause ψ *subsumes* clause φ if $\psi \subseteq \varphi$: ψ is at least as hard to satisfy as φ.

We keep a collection of "used" clauses U which is originally empty. The algorithm ends when C is empty.

- Pick a clause ψ in C and move it to U.
- Add all resolvents of ψ and U to C except that:
  - Tautology elimination: delete all tautologies.
  - Forward subsumption: delete all resolvents that are subsumed by a clause.
  - Backward subsumption: delete all clauses that are subsumed by a resolvent.

Exercise 14. *Show that this algorithm (for any choice of ψ in the first step) is also correct and complete.*

Efficiency

So how large is a resolution proof, even one that uses the subsumption mechanism?

Can we find a particular problem that is particularly difficult for resolution?

Recall that there is a Boolean formula $\mathsf{EO}_k(x_1, \ldots, x_k)$ of size $\Theta(k^2)$ such that σ satisfies $\mathsf{EO}_k(x_1, \ldots, x_k)$ iff σ makes exactly one of the variables $x_1, \ldots, x_k$ true.

$$\mathsf{EO}_k(x_1, \ldots, x_k) = (x_1 \lor x_2 \ldots \lor x_k) \land \bigwedge_{1 \leq i < j \leq k} \neg(x_i \land x_j)$$

Note that formula is essentially in CNF.

Das Dirichletsche Schubfachprinzip

Aka Pigeonhole Principle.

Lemma 3. *There is no injective function from* $[n+1]$ *to* $[n]$*.*

This sounds utterly trivial, but the Pigeonhole Principle is a standard combinatorial principle that is used in countless places.

The classical proof is by induction on n.

Alternatively, we could translate th PHP into a Boolean formula (for any particular value of n, not in the general, parametrized version).

Idea: Variable x_{ij} is true iff pigeon i sits in hole j (or, in less ornithological language, $f(i) = j$).

Pigeonhole Principle

We need a formula that expresses PHP in terms of these variables.

We have variables x_{ij} where $1 \leq i \leq m$ and $1 \leq j \leq n$.

$$\Phi_{mn} = \bigwedge_{i \leq m} \mathsf{EO}_n(x_{i1}, x_{i2}, \ldots, x_{in})$$

Then Φ_{mn} is satisfiable iff $m \leq n$.

In particular we ought to be able to use resolution to show that $\Phi_{n+1,n}$ is a contradiction.

Exponential Lower Bound

By completeness there must be a resolution proof showing that $\Phi_{n+1,n}$ is a contradiction.

But:

Theorem 6. *Every resolution proof for the contradiction* $\Phi_{n+1,n}$ *has exponential length.*

Proof is quite hairy.

Easy Cases

One might wonder if there is perhaps a special class of formulae where a resolution type approach is always fast.

We can think of a clause

$$\{\overline{x}_1, \overline{x}_2, \ldots, \overline{x}_r, y_1, y_2, \ldots, y_s\}$$

as an implication:

$$x_1 \wedge x_2 \wedge \ldots \wedge x_r \longrightarrow y_1 \vee y_2 \vee \ldots \vee y_s$$

When $s = 1$ these implication are particularly simple.

Horn Formulae

Definition 5. *A formula is a* Horn formula *if it is in CNF and every clause contains at most one unnegated variable.*

Example:

$$\varphi = \{\overline{x}, \overline{y}, z\}, \{\overline{y}, \overline{z}\}, \{x\}$$

or equivalently

$$\varphi = x \wedge y \to z, y \wedge z \to \bot, x$$

Horn Clauses

In other words, a Horn formula has only Horn clauses, and a Horn clause is essentially an implication of the form

$$C = x_1 \wedge x_2 \wedge \ldots \wedge x_n \to y$$

where we allow $y = \bot$.

We also allow single unnegated variables (if you like: $\top \to x$).

Note that if we have unit clauses x_i then a resolvent of these and C will be y.

This gives rise to the following algorithm.

Marking Algorithm

Testing Horn formulae for Satisfiability

- Mark all variables x in unit clauses $\{x\}$.
- If there is a clause $x_1 \wedge x_2 \wedge \ldots \wedge x_n \rightarrow y$ such that all the x_i are marked, mark y. Repeat till a fixed point is reached.
- If $\perp$ is ever marked, return No.
- Otherwise, return Yes.

You can also think of this as a graph exploration algorithm: node y is marked only if all predecessor nodes x_i are already marked. (Careful though, y can be the RHS of several clauses.)

Note that Marking Algorithm is linear in the size of Φ (in any reasonable implementation).

Truth Assignment

We can read off a satisfying truth-assignment if the formula is satisfiable:

$$\sigma(x) = \begin{cases} 1 & x \text{ is marked,} \\ 0 & \text{otherwise.} \end{cases}$$

Then $\sigma(\Phi) = 1$.

Moreover, $\tau(\Phi) = 1$ implies that

$$\forall\, x\, (\tau(x) \leq \sigma(x))$$

so σ is the "smallest" satisfying truth-assignment.

Summary

- Satisfiability of Boolean formulae is a very expressive problem: lots of other combinatorial (decision) problems can be rephrased as a Satisfiability problem.
- No polynomial time algorithm is known to tackle Satisfiability in general. There are good reasons to believe that no such algorithm exists.
- The Davis/Putnam algorithm often handles Satisfiability in polynomial time, on very large instances.
- There are commercial versions of the algorithm using clever strategies and data structures.
- Resolution is a refutation based method for satisfiability testing.