Boolean Satisfiability

CNF: <u>Conjunctive Normal Form</u>

A CNF is a Product of Sums of literals
eg: $(x+y'+z)(x'+y+z'+u)(u'+z)$

such a representation is general in that any logic function can be represented (note: can find a CNF from DNF by applying negation & De Morgan laws:

$$f = xy'z + xuz' + yz \Rightarrow \bar{f} = (x'+y+z')(x'+u'+z)(y'+z')$$

although it is trivial to find a truth assignment for DNF i.e $x=1, y'=1, z=1$ above – it is very hard in general to find a truth assignment for CNF 1 (or equivalently an assignment to f<u>alse</u> for DNF…)

SAT: Given a CNF on a set of literals, is there an assignment of values that <u>satisfy</u> CNF
i.e <u>can</u> we find $x, y, z \ldots \ni$ CNF=1?

(note: if <u>not</u> ⇒CNF=0 for all assignments: not satifiable)
⇒ if we can build a BDD for CNF, <u>any</u> path is a SAT & no-solns⇒BDD=0
SAT is still interesting since can be solved (sometimes for tens of thousands of literals & hundreds of thousands of terms)

NP&SAT: How hard is SAT?

SAT represents evaluation of an <u>arbitrary</u> Boolean function. Thus many other problems can be cast in SAT form. This property was exploited by S.Cook to create a partition of “Hard Problems” that, in a specified sense, are equally hard.

Boolean Satisfiability

Notion: <u>Polynomial Equivalence</u>

Given a problem written in K characters, can one find another problem in no more than K^p characters for some fixed $P \ni$ the two problems are <u>equivalent</u>. i.e the first problem is valid if the second one is valid.

Given a problem in a set of polynomial equivalent problems,
if one can solve some problem in <u>Polynomial Time</u>⇒
all problems in class can be solved in P-Time.

Notion: <u>NP-Time</u> there are many problems for which, given a solution, one can check (or validate) the solution in P-Time on a <u>Turing machine</u>. Equivalently, one could solve the problems in P-Time on an exponential set of turing machines, with each machine choosing an alternate character (out of a finite alphabet) on each program cycle.

There are several NP - problems (an infinite set.)

eg. <u>Traveling Salesperson</u> i.e given a graph with weighted edges, is there a tour of each vertex with cost = $\sum$ (of weights of arcs taken) < c, a predefined constant.

eg. <u>Clique Cover</u> i.e given a graph, can one cover each edge with no more than c cliques (complete sub-graphs)?

eg. <u>Knapsack</u> given a collection of objects of different weights, and a sack with capacity M, can one fill the sack so that at most m<<M capacity is unused?

Boolean Satisfiability

SAT is clearly a member of NP: i.e given a CNF and a solution, we can clearly verify the solution in polynomial steps of a turing machine. In fact, SAT is the prototypical NP problem…

Cook showed that he could simulate the behavior of an arbitrary turing machine as a polynomial equivalent instance of SAT. Effectively, this shows that every problem in the class NP can be solved by an instance of SAT with only polynomial growth, in problem size.

So: Many problems in NP can also be used to solve arbitrary instances of SAT$\Rightarrow$Class of problems (NP-complete) which are all equally hard. What is known is that if any such problem has a polynomial time algorithm $\Rightarrow$ all of them will have such solutions.

However, no such problem has been found in 40 years or so of trying. i.e we don't know of P=NP or not, but it looks doubtful.

Back to SAT

Every decision procedure for SAT involves search, usually constrained search.

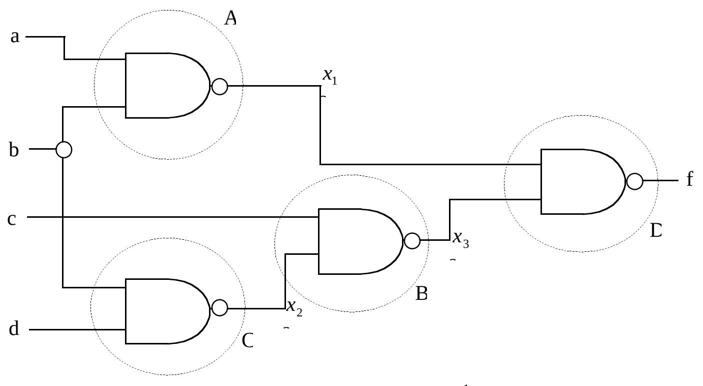
*There are some interesting results for restricted SAT problems:

3-SAT i.e SAT restricted to only 3 literals per product.
$\Rightarrow$This problem is equivalent to SAT!

pff: any SAT problem can be solved if we can evaluate the implied Boolean function. So represent this function as a circuit of 2-input NAND gates. (it is not so easy to show this takes polynomial spree/time). Once in circuit form: write an instance of 3-SAT equivalent to the circuit.

Boolean Satisfiability

Eg: $(a+b'+c'+d')(a+c)(b+c)$



note: for a 2-input NAND } $\leftrightarrow$GATE A$\Rightarrow$ $x_1 \equiv (ab)^1 \equiv a^1 + b^1$

so $x_1 \oplus (ab)' \equiv 0 \equiv x_1 ab + x_1' a' + x_1' b'$

$A \Rightarrow 1 \equiv \left(x_1' + a' + b'\right)(x_1 + a)(x_1 + b)$

To satisfy entire circuit, we must have each gate satisfied.

Boolean Satisfiability

eg:

$$1 \equiv A \cdot B \cdot C \cdot D$$

$$\Rightarrow 1 \equiv \left(x_1^1 + a' + b'\right)\left(x_1 + a\right)\left(x_1 + b\right)\left(x_2' + b' + d'\right)\left(x_2 + b\right)\left(x_2 + d\right) \cdot$$
$$\left(x_3^1 + x_2' + c\right)\left(x_3 + x_2\right)\left(x_3 + c\right)\left(f' + x_1' + x_3'\right)\left(f + x_1\right)\left(f + x_3\right)$$

So each gate can be represented by 3 terms, with at most 3 literals per term: 3-SAT instance.

$\Rightarrow$ Any instance of SAT can be polynomically reduced to an instance of 3-SAT.

(surprisingly, <u>2-SAT</u> can be solved in <u>linear</u> time!)

Boolean Satisfiability

<u>Backtrack Search:</u> (DP)

```
Vector-Value ← unknown;
DPL (Value-Vector)      {
        set some open literal → T
        if (no failing unit clause) {
                DPL (value-vector)
                if success return (value-vector) }
                else {
                set literal → F
                if (no failing unit clause) {
                        DPL (value-vector)
                        if success return (value-vector) }
                return (fail)
```

---

* Clause with no open assignment is true or false
* Clause with one open assignment is unit clause.

Boolean Satisfiability

*Trick 1 $\Rightarrow$ look ahead $\Rightarrow$check for unit values or binary clauses$\Rightarrow$forces sine literals$\Rightarrow$ propagate values to reduce remaining clauses or fail.
(details)

*Trick 2 $\Rightarrow$ choose new literal based on a search heuristic i.e: keep track of # of open clauses in which literal occurs and try to make new literal reduce as many as possible.

*Trick 3$\Rightarrow$ Both 2-SAT & 1-SAT can we solved in linear time$\Rightarrow$characterizing solution as new clauses & continue.

Naïve backtrack spends most of its time in regions of Boolean space with no solutions. Also, typically discover same contradictions many times.

$\rightarrow$Futile backtrack contradiction back tracks a decision that is not related to failure.

$\rightarrow$Poor ordering$\rightarrow$split case for ℓ even when both values of ℓ fail.

Boolean Satisfiability

An Alternative Approach to SAT:

Stålmarck's Method

*Not all problems are equally hard $\rightarrow$ eq. a 3-SAT problem built for a network of 2-input gates has n 3-literal terms and $(2n)$ 2-literal terms. The problem contains sub-problems which can be solved in linear time.

*Some CAD & Formal problems are very large, but not necessarily hard… the following describes a method suited to such problems.

The method relies on yet another logic form:

all logic formulae are written as: $P \Leftrightarrow (q \rightarrow r)$
we write:(p,q,r) for $P \Leftrightarrow (q \rightarrow r)$
when p a literal or $\perp$ (false)

Then it is easy to see that we can write any formula:

$$\sim A \leftrightarrow A \rightarrow \perp$$
$$A \cup B \leftrightarrow \sim A \rightarrow B \leftrightarrow (A \rightarrow \perp) \rightarrow B$$
$$A \cap B \leftrightarrow \sim (A \rightarrow \sim B) \leftrightarrow \sim (A \rightarrow B \rightarrow \perp)$$
$$\leftrightarrow (A \rightarrow (B \rightarrow \perp)) \rightarrow \perp$$
$$\dots$$

Boolean Satisfiability

Suppose we have such a formula (only $\rightarrow \& \perp$ on literals $a_1 \ldots a_n$) and compound subformulas $B_1 \ldots B_k$ where B_k is A (the formula) and each B_i is $C_i \rightarrow D_i$.

We add $b_1 \ldots b_k$ new literals where $b_i \equiv B_i$

Then we can write *A* as a set of <u>triplets</u>

$$\left(b_i, c_1, d_i\right) \Leftrightarrow \left(b_i, rep(C_i), rep(D_i)\right)$$

on literals since $\exists_{l=b_i, a_i}$ such that $\ell \equiv rep\left(C_i\right) \ldots$

Now we can represent any prop. formula as a set of triplets on literals and 0,1 where 0=$\perp$ & 1 is shorthand for true.

Eg: $p \rightarrow (q \rightarrow p)$

$q \rightarrow p \equiv b_1$

$p \rightarrow (q \rightarrow p) \equiv b_2$

$\{(b_1, q, p)$

$(b_2, p, b_1)\}$ (remember: $(x, y, z) \equiv x \Leftrightarrow (y \rightarrow z)$)

Boolean Satisfiability

Basic Procedure: the goal is to prove that the formula is a <u>tantology</u>. So we assume it to be false and drive a <u>contradiction</u>.

There are two kinds of rules to apply to triplets:

1) <u>reduction rules</u>
2) <u>dilemma rule</u>

Each application of a reduction rule reduces the size of the formula (# of triplets) – but we need to also resolve dilemmas (corresponding to the splitting rule) which can make the representation larger.

Strategy: Apply reduction rules until fixed point, then choose literal and apply dilemma rule. Repeat until contradiction or validity.

There are only 7 reduction rules:

R_1:	$(0, y, z)$	$\Rightarrow$	$y \equiv 1, z \equiv 0$
R_2:	$(x, y, 1)$	$\Rightarrow$	$x \equiv 1$
R_3:	$(x, 0, z)$	$\Rightarrow$	$x \equiv 1$
R_4:	$(x, 1, z)$	$\Rightarrow$	$x \equiv z$
R_5:	$(x, y, 0)$	$\Rightarrow$	$x \equiv \sim y$
R_6:	(x, x, z)	$\Rightarrow$	$x \equiv 1, z \equiv 1$
R_7:	(x, y, y)	$\Rightarrow$	$x \equiv 1$

Boolean Satisfiability

A triplet is terminal if it is condradictory

Eg: $(1,1,0) \Rightarrow 1 \rightarrow 0$ must be false

All terminals: $\{(1,1,0),(0,x,1),(0,0,x)\}$

All reduction rules remove a triplet and replace literals in the other triplets

Eg: to prove: $p \rightarrow (q \rightarrow p)$ we assume it to be false

$\{(b1,q,p),(b2,p,b_1)\}$ ie. $b_2 \leftarrow 0$

$\{(b1,q,p),(0,p,b_1)\}$

$R_1 \rightarrow (0,p,b_1) \Rightarrow p \leftarrow 1 \;\; b1 \leftarrow 0$

$\Rightarrow \{(0,q,1)\}$ which is terminal $\Longrightarrow$ Tautology

Unfortunately this is not complete: we need one more rule:

Boolean Satisfiability

Dilemma Rule:

Given set R and choice of variable x

Form $R[x \leftarrow 1]$ and $R[x \leftarrow 0]$

for each set apply all reduction rules possible.

i.e. for $R[x \leftarrow 1] \underset{Rules}{\longrightarrow} U$ and for $R[x \leftarrow 0] \underset{Rules}{\longrightarrow} V$

now: form $U \cap V$ i.e. the intersection of U & V

note: if either U or V is terminal $\Longrightarrow$ choose other derivation.
if both U and V terminal $\Longrightarrow$ contradiction, so premise is proven.

Boolean Satisfiability

Eg: Suppose we have to refute the following set of triplets

$\{(1,\sim p, p)$
$(1, p, \sim p)\}$
none of the rules we have apply here.

Dilemma Rule:

Set $p \leftarrow 1$: $\{(1,0,1),(1,1,0)\}$
$(1,1,0)$ is terminal (false) So this bunch is contradictory.

Set $p \leftarrow 0$: $\{(1,1,0),(1,0,1)\}$
is also terminal $\Rightarrow$
both branches of set are contradiction $\Rightarrow$ formula is refuted.

In general, we might have to apply the Dilemma rule recursively on the sub-derivations of other open dilemmas. (This is how the system obeys NP-completeness).

So although we have a proof system, we need a strategy for applying rules to make an efficient procedure.

Boolean Satisfiability

Note: By applying a saturating strategy for the rules, we define the proof depth as the maximal # of open dilemmas needed to refute to the premise.

(Saturation here means successful reduction of the set by simple rules until no further reduction is possible.)

$\Rightarrow$ 0 degree $\Rightarrow$ only simple rules suffice to refute
1 degree $\Rightarrow$ must apply dilemma, but not recursively
2 degree $\Rightarrow$ dilemma clauses are degree 1
…

In the worst case, K-literal problems might need Kth degree…
However, a study of several benchmark problems in VLSI and formal design showed that the complexity was very often {0,1,2}. In such cases, this decision procedure is polynomial on problem size. i.e. 0 problems degree are linear time.

We can generalize this proof system slightly to improve performance by adding a few more reduction rules and replacing "$\Rightarrow$" with any 2-input Boolean connective operator:

Triplett becomes: $x \equiv y \circ z$ for any $0 \in \{\&, |, \oplus ...\}$

Boolean Satisfiability

Eg: $\left(A \cap (B \cup C)\right) \to \left((A \cap B) \cup (A \cap C)\right)$

a: A	$(d, b \cup c)$
b: B	$(e, a \cap d)$
c: C	$\vdots$
d: $b \cup c$	
e: $a \cap d$	
f: $a \cap b$	
g: $a \cap c$	
h: $f \cup g$	
i: $e \to h$	

$\longrightarrow$ much larger set of rules, Eg: $d{:}x \cap x \to d \equiv x$ or $d \leftarrow x$

Boolean Satisfiability

The procedure on triplets is as follows:

```
$R \in$ set of triplets
sat $(R, \phi)$ = {
        $Q \leftarrow$ Compound $(R)$        $\leftarrow$only simplify non-literals
        while $(Q \neq \phi)$        {
                choose $q \in Q$
                if (contradiction $(R(q))$)  return $R(q)$
                else {  $Q \leftarrow Q \cup$ changed $(R(q) - R)$
                        $R \leftarrow R(q)$
                     }
        }
        return $R$
}
```

Boolean Satisfiability

```
sat $(R, K+1)$ = {
    do {
        $L \leftarrow \text{sup}(R)$
        $R' \leftarrow R$
        for each literal $(\ell \in L)${
            $R1 \leftarrow \text{sat}(R[\ell \leftarrow \text{false}], K)$
            $R2 \leftarrow \text{sat}(R[\ell \leftarrow \text{true}], K)$
            if contradiction $(R1)$ and contradiction $(R2)$
                return $(R1 \cup R2)$
            else if contradiction $(R1)$
                $R \leftarrow R2$
            else if contradiction $(R2)$
                $R \leftarrow R1$
            else
                $R \leftarrow R1 \cap R2$
        }
    } while $(R' \neq R)$
} return $(R)$
```

Overall: proof of depth K on length m formula is $\vartheta(m^{2K+1})$