

Lecture 1: 1.18.04

Lecturer: Christos

Scribe: Michael Demmer / Prabal Dutta

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

These notes are based on the work of Sonesh Surana and Norm Zhao from the 2003 offering of this class.

1.1 Satisfiability

Satisfiability or SAT is a basic problem in computer science. The problem is to determine whether there exists a truth assignment to variable appearing in a boolean expression ϕ in Conjunctive Normal Form (CNF) such that ϕ is satisfied (true). A boolean expression in CNF is

- a set of clauses which are logically AND-ed together (i.e. a conjunction of clauses)
- in which in each clause is a disjunction of literals (i.e. logical OR-ing)
- where a literal is simple a positive or negated variable (i.e. X or \bar{X})

CNF goes by other names in other contexts. For example, in digital circuit design, CNF is called the Product of Sums (POS) although most digital designers prefer to represent boolean expressions in the dual representation of Sum of Products (SOP).

Let m , n , and l represent

- m - the number of clauses
- n - the number of variable
- l - the number of literals

appearing in a boolean expression ϕ in CNF. For example, for the following expression in CNF:

$$(X \vee Y \vee Z) \wedge (\bar{X} \vee Y) \wedge (\bar{Y} \vee Z) \wedge (\bar{Z} \vee X) \wedge (\bar{X} \vee \bar{Y} \vee \bar{Z}) \quad (1.1)$$

In this expression, $m = 5$, $n = 3$, and $l = 12$. Is this expression satisfiable? No. The reason is that for the first clause to be true, either X , Y , or Z must be true. For the last clause to be true, either X , Y , or Z must be false. The three middle clauses imply that X , Y , and Z must all be the assigned the *same* truth assignment. A clause of the form $(\bar{X} \vee Y)$ means that X implies Y , written $X \rightarrow Y$, since if X is *true*, then Y must be *true* for the clause to be satisfied. So, the three middle clauses provide the following implications: $X \rightarrow Y$, $Y \rightarrow Z$, $Z \rightarrow X$. The only assignment that simultaneously satisfies these implications are assignments in which $X = Y = Z$.

A naïve approach to solving SAT would be brute force. There are 2^n such truth assignments and l literals to set for each assignment, giving a total of $O(l \cdot 2^n)$ operations. In practice, expressions involving dozens or hundreds of variables and literals are not uncommon, making brute force unusable for all but the simplest boolean expressions. SAT is, in general, an NP-complete problem but restrictions on the type of the boolean expressions can give rise to specific formulations of SAT which can be solved in polynomial time.

1.2 Davis-Putnam Algorithm

The original algorithm for solving SAT was proposed by Davis and Putnam, and is now called the Davis-Putnam procedure. To demonstrate how this procedure works, suppose we have the expression

$$(A \vee C)(\bar{A} \vee C)(B \vee \bar{C})(A \vee \bar{B}) \quad (1.2)$$

We pick any one literal, say \bar{A} , and set it to *true*. Then, we can simplify the formula by removing all *clauses* in which \bar{A} appears and also by removing all *occurrences* of the literal's negation, which, in this case is the literal A . This procedure is called “chasing a literal” and is the fundamental step in the Davis-Putnam procedure.

Chasing A in the above expression results in the following, simpler, expression

$$(C)(B \vee \bar{C})(\bar{B}) \quad (1.3)$$

The key idea is to chase a literal to get a simpler formula, then chase another literal to get a still simpler formula, and so on until one of two end game scenarios emerge:

- If an empty clause results, then we abort since the expression cannot be satisfied.
- If all clauses are deleted, then we know that the expression has been satisfied with the truth assignments we selected when chasing the literals.

If we encounter a unit clause (a clause containing only one literal), we set that literal to true and chase it. Otherwise, we pick a variable and chase both its positive and negative literal. Chasing literals in this manner captures all of the ramifications of each variable being *true* or *false* by allowing us to branch out into the space of truth assignments. The key insight that the Davis-Putnam procedure codifies is that it is possible to determine truth or falsehood even with a partially completed truth assignment due to the construction of the boolean expression – conjunctive normal form – since a clause is *true* if *any* of its literals are *true* even if other literals have not been assigned truth values. Similarly, if *any* clause is *false*, which occurs when *all* of its literals are *false*, then the entire expression is also *false*. This insight provides an efficient manner in which to examine the entire subtrees in the search space.

Algorithm 1: DAVIS-PUTNAM PROCEDURE

- 1: S is a state of partial truth assignments, initially $\{?? \dots\}$ (undecided)
 - 2: **while** $S \neq \emptyset$ **do**
 - 3: pick $t \in S$ and an undecided in t
 - 4: pick variable x
 - 5: $S = S - t + \mathbf{chase}(t, x) + \mathbf{chase}(t, \bar{x})$
 - 6: **end while**
 - 7: output UNSAT
-

The Davis-Putnam procedure is considered a heuristic; that is, an “algorithm” that works reasonably well but we don’t know why. Davis-Putnam’s efficacy depends on picking the “right” variables to chase. Various heuristics have been developed to improve the chances of picking the optimal variables but the procedure’s running time is still exponential. However, in general, the Davis-Putnam procedure runs much more efficiently than brute force because each time a literal is chased, an entire subtree of possibilities are eliminated.

 Algorithm 2: **chase**(ϕ, x)

```

1: set literal  $x$  to true
2: delete all clauses containing  $x$  from  $\phi$ 
3: delete all occurrences of literal  $\bar{x}$  from all clauses in  $\phi$ 
4: if empty clause results then
5:   return nothing
6: end if
7: if unit clause containing only one literal  $l$  results then
8:   chase( $\phi, l$ )
9: end if
10: if all clauses deleted then
11:   output SAT
12: end if

```

1.3 Horn-SAT

It turns out that by imposing some restrictions on the form of the boolean expression, we can solve the satisfiability problem in polynomial time. One example is Horn-SAT, in which case the expression is a conjunction of horn clauses, in which at most one literal is positive. For example, $(W \vee \bar{X} \vee \bar{Y} \vee \bar{Z})$ is a horn clause, but $(X \vee Y)$ is not.

For example, the following expression is an example of a horn formula:

$$(\bar{X} \vee Y \vee \bar{Z})(\bar{X} \vee Y)(X)(\bar{X} \vee \bar{Y} \vee Z)(\bar{Z} \vee \bar{W} \vee \bar{X})(W \vee \bar{Z} \vee \bar{Y})(\bar{W} \vee \bar{U})$$

The restriction to horn clauses means that we can rewrite all horn formulas as a sequence of implications, in that the truth of all the negative literals implies the truth of the lone positive literal. If there are no negative literals in the clause, then the lone positive literal is asserted to be true. If there is no positive literal, then the conclusion is simple *False*. Thus we can rewrite the above horn formula as:

$$(XZ \implies Y)(X \implies Y)(\implies X)(XY \implies Z)(ZWX \implies)(ZY \implies W)(WU \implies)$$

Note first of all that if there are no clauses of the form $(\implies X)$, then the formula can be trivially satisfied by setting all variables to false. This will be the intuition behind our algorithm, which conservatively sets variables to true, only when necessary, hence why it is called the stingy algorithm.

 Algorithm 3: **Stingy**

```

1:  $t \leftarrow False^n$ 
2: while there exists an implication  $l \implies X$  where all  $l$  literals are true do
3:   if  $X$  is nil then
4:     return "UNSAT"
5:   end if
6:   set  $X \leftarrow True$  in  $t$ 
7: end while
8: return "SAT"

```

Examining our previous example, we would first set X to true. Then, we see that Y must also be true

because of the $(X \implies Y)$ clause. In turn, Z must also be true due to the $(XY \implies Z)$ clause. Now we know that W must also be true due to the $(ZY \implies W)$ clause. Finally, we note that the $(ZWX \implies)$ clause has no right hand side, and therefore the formula is unsatisfiable.

Theorem 1.1 *If t is returned by Stingy and t' satisfies the formula Φ , then t' has a True literal wherever t has a true literal.*

Proof: By contradiction. Suppose $X = False$ in t and $X = True$ in t' . Take the earliest such X . In other words, if we arrange the variables in the order that they were flipped by the Stingy algorithm, X is the first variable in the ordering that is *False* in t and *True* in t' . Therefore, we know that t and t' agree up to this point.

Since all variables start out *False*, at some point X was turned true in t' . This implies that X appears in the right hand side of some implication in Φ and all the literals on the left hand side are *True*. Therefore X must be true, so there is a clause in t' that is not satisfied with $X = False$. This is a contradiction and X must be *True* in t' . Therefore all variables that are *True* under t must also be *True* in t' . ■

In the context of the above example, say we line up the variables in order as described in the proof:

Variables:	X	Z	Y	W	\dots
Stingy (t):	T	T	T	T	\dots
Satisfies Φ (t'):	T	T	T	T	\dots

Thus if we set any variable to *False* then we can see a contradiction since it either cannot have been returned from Stingy, or it must not satisfy the formula.

What is the running time of the Stingy algorithm? $O(n^2)$. Can we make this linear time? Yes, using some data structures to help us out:

CLAUSELIST(x):	List of clauses containing X in the LHS
RHS(c):	Variable in the RHS of clause c
COUNTNEG(c):	Count of negative clauses in c
Q(c):	Queue of violated clauses

The Q is initialized with all clauses of the form $(\implies X)$. While there is a clause c in Q, for each clause c' in CLAUSELIST(RHS(c)), decrement COUNTNEG(c'). If the count is 0, add it to Q. This algorithm is linear because each clause is only examined as many times as there are literals.

1.4 2-SAT

Another restriction on the forms of boolean expressions is to allow the disjunction of at most two literals in every clause. This is the 2-SAT problem and can be solved easily in polynomial time.

Why is 2-SAT easy while 3-SAT (or 4-SAT...) is NP-complete? The intuition is that once you fix a truth assignment for a variable X , that creates a deterministic “ripple” effect, forcing the truth assignments of all other variables that share a clause with X . Essentially, running **chase** once, from any starting point, wins.

For example, take the formula:

$$(X \vee Y)(Z \vee \bar{Y})(\bar{X} \vee \bar{Y})(\bar{X} \vee Y)(W \vee Z)(\bar{Z} \vee U)(\bar{U} \vee \bar{W})(X \vee W)(W \vee \bar{Z})$$

To test for satisfiability, suppose we set X to be true and run **chase** on it. This has the immediate effect of removing the clauses $(X \vee Y)$ and $(X \vee W)$, and simplifying out all the \bar{X} literals, leaving us with:

$$(Z \vee \bar{Y})(\bar{Y})(Y)(W \vee Z)(\bar{Z} \vee U)(\bar{U} \vee \bar{W})(W \vee \bar{Z})$$

From which we it is apparent that this truth assignment is impossible, since we're left with $(\bar{Y})(Y)$, which when simplified, will always result in an empty clause.

Trying again by setting X to false, we're left with:

$$(Y)(Z \vee \bar{Y})(W \vee Z)(\bar{Z} \vee U)(\bar{U} \vee \bar{W})(W)(W \vee \bar{Z})$$

Now, continuing by setting Y to true and then setting W to true, we get:

$$(Z)(Z)(\bar{Z} \vee U)(\bar{U})(\bar{Z})$$

And again, there is a contradiction, this time $(Z)(\bar{Z})$, so this formula is unsatisfiable.

1.5 Random-Walk 2-SAT

Another algorithm to solve 2-SAT illustrates a randomized approach:

Algorithm 4: Random-Walk 2-SAT

- 1: $t \leftarrow$ any
 - 2: **while** there are unsatisfied clauses (and year \neq 2010) **do**
 - 3: pick any clause
 - 4: pick a random literal in the clause
 - 5: flip the literal
 - 6: **end while**
-

Why does this work? Well, suppose there is a satisfying truth assignment t_0 . Then t starts randomly trying various truth assignments, moving one step closer to t_0 with probability half or one step away from t_0 with probability one half.

By definition, t cannot ever be more than n steps away from t_0 since there are only n variables and the worst case is that they're all wrong in t . Therefore, it should take $O(n^2)$ steps to reach t_0 . To be safe, one might want to add some assurance, say a constant factor $k = 10$ to the number of times the algorithm is run.

This is a basic nature of unbiased random walks, that it takes $O(n^2)$ steps to reach either $+n$ or $-n$ from the starting point 0 with high probability.