CS270 Combinatorial Algorithms & Data Structures **Spring 2006**

Lecture 1: 1.17.06

Lecturer: Satish *Scribe: Jason Wolfe*

1.1 Propositional Satisfiability (SAT)

A *propositional formula* ϕ is *satisfiable* iff there exists a substitution of truth values for its variables that makes it true. The *SAT* problem is to, given a formula ϕ, either find such a *satisfying assignment* or prove that none exists and thus that ϕ is *unsatisfiable*.

3-SAT, a prototypical NP-complete problem, is a special case of SAT where ϕ is restricted to *3-CNF* (3-conjunctive normal form). A 3-CNF formula is expressed as a conjunction of *clauses*, each of which is represented by a disjunction of at most three *literals* (possibly negated variables). For example, the following is an unsatisfiable 3-SAT formula with 12 literals, 5 clauses, and 3 variables:

$$(X \vee Y \vee Z)(X \vee \overline{Y})(Y \vee \overline{Z})(Z \vee \overline{X})(\overline{X} \vee \overline{Y} \vee \overline{Z}) \tag{1.1}$$

This formula can be seen to be unsatisfiable by inspection: the first and last clause require (respectively) that at least one variable is true and at least one variable is false, whereas the middle clauses (together) require that all variables have the same truth value (recall that $(X \vee \overline{Y}) \Leftrightarrow (Y \Rightarrow X) \Leftrightarrow (\overline{X} \Rightarrow \overline{Y})$).

Since 3-SAT is NP-complete, checking if a given assignment satisfies ϕ can be (trivially) accomplished in polynomial time, but finding such an assignment or proving that ϕ is unsatisfiable requires *exponential time* in general. Letting m, n, and l be (respectively) the number of clauses, variables, and literals in ϕ, the naive "try everything" or "guess and check" algorithm runs in $O(2^n * l)$ time, since each of the 2^n possible truth assignments to n variables requires at most l time to check.

1.2 The Davis-Putnam Procedure

Davis and Putnam's original SAT-solving algorithm, now called the Davis-Putnam Procedure, can be much faster than simply "trying everything". The procedure relies on the fact that we can often determine the truth value of a clause or formula without knowing all of its variables' values, since a single *true* literal makes a clause *true* and a single $false$ clause makes a formula $false$. Consider the following logical formula ϕ:

$$(A \vee C)(\overline{A} \vee C)(B \vee \overline{C})(A \vee \overline{B}) \tag{1.2}$$

The procedure works by repeatedly *chasing* variables out of the formula, by first setting a literal to *true* and then simplifying the resulting formula. For example, chasing $\overline{A}$ in ϕ gives the following residual formula:

$$(C)(B \vee \overline{C})(\overline{B}) \tag{1.3}$$

In chasing $\overline{A}$, we first set it to *true*, then delete all clauses containing $\overline{A}$ from ϕ, and finally delete all instances of the literal A from any remaining clauses. If ϕ contains a *unit clause* consisting of a single literal, this

literal is always chased first[1]; otherwise, an arbitrary literal is selected and the results of chasing it and its negation are considered separately. The crucial invariant in this procedure is that whenever a residual formula is satisfiable, the entire original formula ϕ is satisfiable as well (but not vice-versa).

This process is repeated on each possible line of reasoning until the empty formula is reached, meaning that the current assignment satisfies ϕ, or an empty clause is produced, indicating that ϕ is unsatisfiable under the current assignment. This early detection of unsatisfiability, along with the chasing of unit literals, can often cut entire branches from the search space.

Algorithm 1 Davis-Putnam Procedure

```
1: procedure DAVIS-PUTNAM(φ)
2:     S ← {(φ, [])}            ▷ Initially, S contains only a pairing of φ with the empty truth assignment.
3:     while S is not empty do
4:         (φ', t) ← an arbitrary element of S
5:         x ← an arbitrary literal in φ'
6:         S ← (S − {(φ', t)}) ∪ CHASE(φ, x, t) ∪ CHASE(φ, x̄, t)
7:     end while
8: end procedure

9: function CHASE(φ, x, t)
10:    Set x to true in t
11:    Delete all clauses from φ that contain the literal x
12:    Delete the literal x̄ from all clauses in φ
13:    if φ is empty then abort from DAVIS-PUTNAM with t        ▷ t is a satisfying assignment
14:    elseif φ contains an empty clause then return {}         ▷ t contains bad decisions
15:    elseif φ contains a unit clause (y) then return CHASE(φ, y, t)   ▷ Continue the chase
16:    else return {(φ, t)}
17: end function
```

Pseudocode for the Davis-Putnam Procedure is provided in Algorithm 1[2]. This algorithm template does not specify what choices to make at the two arbitrary decision points; numerous specialized heuristics have been developed for this purpose, which typically improve the procedure's average-case behavior but do not eliminate its worst-case exponential time complexity.

1.3 Horn-SAT

For some restricted classes of logical formulae, the satisfiability problem can be solved efficiently in polynomial time. One such class is that of *Horn* formulae, which consist only of *Horn clauses* with at most one positive literal. For example, $(A \vee \overline{B} \vee \overline{C})$ is a Horn clause but $(A \vee B \vee \overline{C})$ is not. Here is a sample Horn formula ϕ:

$$(\overline{X} \vee \overline{Z} \vee Y)(\overline{X} \vee Y)(X)(\overline{X} \vee \overline{Y} \vee \overline{Z})(\overline{Z} \vee \overline{W} \vee \overline{X})(W \vee \overline{Z} \vee \overline{Y})(\overline{W} \vee \overline{U}) \tag{1.4}$$

Each clause in a Horn formula corresponds to an implication with a conjunction of zero or more positive literals on the left (tail), and zero or one positive literals on the right (head)[3]. Thus, ϕ can be rewritten as:

$$(XZ \Rightarrow Y)(X \Rightarrow Y)(\Rightarrow X)(XYZ \Rightarrow)(ZWX \Rightarrow)(ZY \Rightarrow W)(WU \Rightarrow) \tag{1.5}$$

[1] Since the literal must be true for the formula to be satisfiable

[2] At the cost of a bit of complexity, DAVIS-PUTNAM could be made more efficient by eliminating the second call to CHASE when $\overline{x}$ does not appear anywhere in ϕ'

[3] An empty tail corresponds to *true*, and an empty head corresponds to *false*.

Notice that if the (X) (i.e., $(\Rightarrow X)$) were not present, we could satisfy ϕ by simply setting all variables to $false$. This observation suggests a simple "stingy" algorithm for finding a satisfying assignment, which begins by setting all variables to $false$ and proceeds by setting the head of any *activated* implication (with a *true* tail) to *true*. This process continues until all clauses are satisfied, or until a headless clause is activated, indicating that ϕ is unsatisfiable.

Algorithm 2 Stingy Horn-SAT

```
1: function STINGY(φ)
2:     t ← [false, ..., false]                ▷ Begin with all variables set to false.
3:     while (∃c ∈ φ) c is false under t do
4:         if c is headless then return "UNSAT"
5:         Set the head of c to true in t
6:     end while
7:     return t                               ▷ φ is satisfiable with t
8: end function
```

Pseudocode for the STINGY procedure is provided in Algorithm 2. Intuitively, this algorithm works since every variable it sets to *true* is *true* in *every* satisfying assignment; consequently, when it activates a headless clause, the formula must be unsatisfiable. More formally, we have:

Lemma 1.1 *For any truth assignment t' satisfying Horn formula ϕ, every variable that is true in* STINGY*(ϕ) is also true in t'*

Proof: By contradiction. Consider the first (w.r.t. STINGY) true variable x in t that is false in t'. At the point when STINGY set x to true, it was the head of some activated clause c. Since x was the *first* disagreement between t and t', c must also be activated in t'; however, since its head x is false in t', t' does not satisfy c and thus is not a satisfying assignment at all (a contradiction). ■

One can easily see that STINGY runs in $O(n * l)$ time. In fact, the algorithm can be improved to run in linear time by adding some clever data structures. For each variable x we maintain tail-list[x], which records the list of clauses in which x appears in the tail. In addition, for each clause c we keep track of head[c], its head, as well as margin[c], the number of variables in the tail that currently take on value $false$.

Algorithm 3 Improved Stingy Horn-SAT

```
1: function LINEAR-STINGY(φ)
2:     t ← [false, ..., false]                ▷ Begin with all variables set to false.
3:     q ← the set of clauses c ∈ φ with empty tails   ▷ q is a queue of activated clauses.
4:     while q is not empty do
5:         c ←POP(q)
6:         if c is headless then return "UNSAT"
7:         if head[c] is true in t then continue   ▷ head[c] was already set to true earlier.
8:         Set head[c] to true in t
9:         for c' ∈ tail-list[head[c]] do
10:            margin[c']←margin[c']−1
11:            if margin[c']= 0 then PUSH(c', q)
12:        end for
13:    end while
14:    return t                               ▷ φ is satisfiable with t
15: end function
```

The pseudocode for LINEAR-STINGY, shown in Algorithm 3, runs in $O(l)$ time. While Horn form might

seem too restrictive to be useful, it actually has many interesting practical applications (e.g., the logic programming language Prolog is actually based on Horn clauses).

1.4 2-SAT

The satisfiability problem for *2-CNF* formulae, in which each clause has at most two literals, is called *2-SAT* and can also be solved efficiently in polynomial time. Here is an example 2-CNF formula ϕ:

$$(X \vee Y)(Z \vee \overline{Y})(X)(\overline{X} \vee \overline{Y})(W \vee Z)(\overline{Z} \vee W)(\overline{U} \vee \overline{W})(\overline{Z} \vee U) \tag{1.6}$$

It turns out that the original Davis-Putnam Procedure (see Algorithm 1) suffices for efficiently solving 2-SAT, with one small modification. To understand why this is the case, consider what happens when we chase X in ϕ:

$$(Z \vee \overline{Y})(\overline{Y})(W \vee Z)(\overline{Z} \vee W)(\overline{U} \vee \overline{W})(\overline{Z} \vee U) \tag{1.7}$$

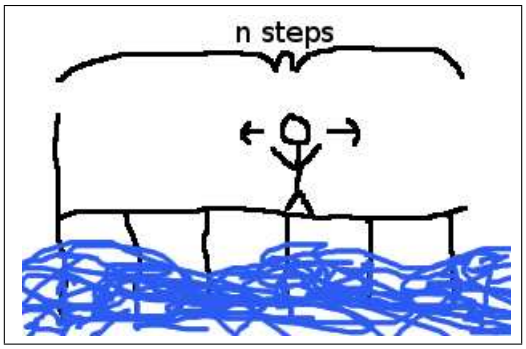
Note that every clause that remains is either a unit clause, or was present in ϕ. When unit clauses are present, they can be eliminated by continuing the chase:

$$(W \vee Z)(\overline{Z} \vee W)(\overline{U} \vee \overline{W})(\overline{Z} \vee U) \tag{1.8}$$

Again, the remaining non-unit clauses were all present in ϕ; in fact, this must be the case since DAVIS-PUTNAM only modifies clauses by deleting literals. Crucially, because of this fact, the residual formula is satisfiable *iff* ϕ was. Thus, there is no need to backtrack, and 2-SAT can be solved by a linear sequence of linear chases. In DAVIS-PUTNAM, this corresponds to skipping the second call to CHASE whenever the first call returns a non-empty result.

1.4.1 A Brief Aside

Consider the following drunken sailor:



The sailor finds himself on a 2-dimensional pier with a wall at one end and open water at the other. In his drunken state, the sailor is only able to wander at random, stumbling a step to the left or right with 50% probability at each point. If he attempts to stumble into the wall, he bounces off.

Lemma 1.2 *With probability* ≥ 0.5*, the sailor gets wet within* n^2 *steps*

Proof: Omitted. ■

To get some intuition for this result, consider the following fact: if the sailor was wandering on an infinite line instead, after k steps (starting at 0) the probability distribution over his position would be roughly normal with mean 0 and variance k.

1.4.2 Randomized 2-SAT

In addition to governing the fate of drunken sailors, Lemma 1.2 also has implications for the running time of SAT-solving algorithms that incorporate randomness.

Algorithm 4 Randomized 2-SAT

```
1: function RANDOMIZED-2SAT(φ)
2:     t ← an arbitrary random truth assignment
3:     while (∃ c ∈ φ) c is false under t do
4:         c ← a random element of {c ∈ φ | c is false under t}
5:         Set a random literal from c to true in t
6:     end while
7:     return t                                   ▷ φ is satisfiable with t
8: end function
```

Algorithm 4 shows pseudocode for a randomized 2-SAT solver, which begins with a random truth assignment, and then repeatedly sets a random literal from some unsatisfied clause to *true* until ϕ becomes satisfied.

Assume that ϕ is satisfiable with some unknown assignment t', and consider the *agreement* between t and t' at each point (the number of variables on which they agree). When this number reaches n, $t = t'$ and the problem has been solved. Furthermore, RANDOMIZED-2SAT's procedure for choosing x guarantees that with each step the agreement between t and t' increases by 1 with probability $\geq .5$ and decreases by 1 otherwise. Thus, by Lemma 1.2, after n^2 iterations a solution has been found with probability $\geq .5$. This success probability can be amplified arbitrarily: after $c * n^2$ steps, the probability that a solution has not yet been found is $\leq .5^c$.

Note that this procedure can never prove that ϕ is unsatisfiable; nonetheless, some of today's most successful SAT-solvers rely on randomization, much like RANDOMIZED-2SAT.