Today:  In the previous two lectures, we discussed ETH and SETH. Both of these are conjectures about the time complexity of SAT. But to really understand why people believe in these conjectures, it’s important to know what techniques are known for solving SAT faster. So for the next two lectures, we will discuss the best known theoretical algorithms and techniques for solving SAT in the worst case. (In practice, SAT is often solved very efficiently. But there are instances in practice that make SAT solvers trip up, and run in exponential time.)

Think about how to do the exercises below! They’ll help you understand the material. (But you don’t have to turn in solutions to them.)

1 Randomized reduction: a first start

We’ll start with a simple algorithm that beats $2^n$ time in a special case of 3SAT. It will illustrate some useful principles in some of the algorithms to come. In the following, a 3-clause is a clause with three literals.

**Theorem 1.1** 3-SAT can be solved in $O^*((3/2)^t)$ randomized time on formulas with at most $t$ 3-clauses. In particular, our randomized algorithm always outputs “UNSAT” on unsatisfiable formulas, and outputs “SAT” on satisfiable formulas with probability greater than $1 - 10^{-9}$.

Note, some 3-SAT instances could have many 2-clauses and a small number of (say, $O(\log n)$) 3-clauses. The above theorem shows that these instances could be solved quickly with randomness. The algorithm is basically a randomized reduction from 3SAT to 2SAT.

**Proof.** First we’ll give an algorithm RANDO which takes a formula $F$ with at most $t$ 3-clauses. Then we’ll explain why RANDO works.

RANDO($F$):
Repeat for $20 \cdot (3/2)^t$ trials:
   Let $F'$ be the set of 2-CNF and 1-CNF clauses in $F$.
   For every 3-clause $C = (\ell_1 \lor \ell_2 \lor \ell_3)$ in $F$,
      Randomly choose an $\ell_i$ and remove $\ell_i$ from $C$, forming a new 2-clause $C'$. Add $C'$ to $F'$.
   End for
Solve the resulting 2-SAT instance $F'$ in polynomial time. If $F'$ is satisfiable then return “SAT”.
End repeat
Return "UNSAT".

**Exercise 1.1** Hmm... how do you solve 2-SAT in polynomial time? (There are multiple ways you could prove this.)

Now for the analysis of RANDO. The key observation is this: if any $F'$ is satisfiable by some assignment $A$, then $F$ is also satisfiable by $A$, because $F'$ is a restriction to $F$ (its clauses are only shorter than those of $F$).
First, if $F$ is unsatisfiable, then every $F'$ computed by RANDO is also unsatisfiable, and the algorithm will never return “SAT”.

Second, suppose $F$ is satisfiable, and let $A$ be a satisfying assignment to it.

**Claim 1.1** For all clauses $C$, $\Pr[A \text{satisfies } C'] \geq 2/3$.

**Proof of Claim:** In the worst case, exactly one literal $\ell$ in $C$ is satisfied by $A$. (In fact, if more than one literal in $C$ is satisfied by $A$, then $A$ satisfies $C'$ with probability $1$!) This literal $\ell$ is removed with probability $1/3$. And if you remove any other literal instead, the remaining clause is still satisfied.

QED

Since each literal removed is an independent choice, we have:

$$\Pr[F' \text{ is SAT by } A] \geq (2/3)^t.$$  

In RANDO, we repeat the inner loop for $r = 20 \cdot (3/2)^t$ trials. Therefore

$$\Pr[\text{No } F' \text{ is SAT by } A \text{ over all trials}] \leq \left(1 - \frac{(2/3)^t}{r}\right) = \left(1 - \frac{20 \cdot (3/2)^t}{(3/2)^t}\right) \leq \exp(-20 \cdot (2/3)^t) \cdot (3/2)^t = \exp(-20).$$

(Here, we applied the useful inequality $(1 - x) \leq \exp(-x)$.) Therefore, when the algorithm reports "UNSAT", the probability it is wrong is less than $e^{-20} < 10^{-9}$.

Note: as far as I know, the best known algorithm of this kind (in terms of the number of 3-clauses $t$) is deterministic and runs in $O^*(1.3645^t)$ time [BE05].

2 Algorithms which beat $2^n$ time in general

Now we turn to algorithms which really do beat $2^n$ time for $k$-SAT. First, we give some simple improvements over exhaustive search. We’ll start with branching (a.k.a. backtracking) algorithms, as they are very natural. In a branching algorithm, the idea is that you try to cleverly pick variables to assign to true or false. As you assign them, all of their occurrences in the formula get removed, and their removal simplifies the formula. (If you are clever about how you pick them, you could simplify the formula considerably.) Then you recurse on the simplified formula.

Among the algorithms we’ll see, the branching paradigm is most like actual real-life SAT solvers. (In the 1990s, local search was the fastest, but in the early 2000s people began engineering fast branching algorithms coupled with additional heuristics, and they’ve been the fastest ever since, with many technical refinements and improvements added over the last 20 years.)

**Theorem 2.1** $k$-SAT on $n$ variables is in $2^{n-k2^k \cdot \text{poly}(n)}$ time.

**Proof.** As promised, this will be a backtracking/branching algorithm. The key idea is this: whenever we have a clause of $\ell$ literals, there are only $2^\ell - 1$ satisfying assignments to the clause, even though there are $2^\ell$ total assignments to the clause. This difference of one assignment is enough to get some running time improvement!

**Algorithm A** ($F$): // $F$ is a $k$-CNF formula
If $F$ has no clauses, return SAT.
If $F$ contains an empty clause (a clause with no literals), return UNSAT.
Take the shortest clause in $F$, call it $C = (x_1 \lor \ldots \lor x_\ell)$.
For all $2^\ell - 1$ satisfying assignments $a \in \{0, 1\}^\ell$ to $C$,

---

1Note you could have interpreted $!$ as factorial, and the statement is still true. You could have even interpreted the footnote symbol as an exponent of 1, and the statement is still true.

2That is, there are fixed constants $c, d > 0$ such that for all $k$, $k$-SAT on $n$ variables is solvable in $2^{n-k2^k \cdot n^d}$ time.
Call \( \text{A}(F|_{x_1=a_1,...,x_\ell=a_\ell}) \).

// this notation just means \( x_1, \ldots, x_\ell \) are replaced by the bits in \( a \), so \( F|_{x_1=a_1,...,x_\ell=a_\ell} \) has \( \ell \) fewer variables;
// also, any already satisfied clauses are removed.

If one of these calls returns SAT, then return SAT. Otherwise return UNSAT.

**Analysis:** In the worst case, the shortest clause \( C \) is always of length \( k \). (It couldn’t be any longer.) In that case, our running time recurrence is \( T(n) \leq (2^k - 1)T(n - k) + O(\text{poly}(n)) \). This easily solves to \( T(n) \leq (2^k - 1)^{n/k} \text{poly}(n) \).

Using the inequality \( 1 - x \leq e^{-x} \), we find that
\[
(2^k - 1)^{n/k} = 2^n (1 - 1/2^k)^{n/k} \leq 2^n e^{-n/(k2^k)},
\]
and we are done. \( \Box \)

**Exercise 2.1** Convince yourself that we can already conclude the following:
For all \( k \), there is a \( \delta < 1 \) such that \( k \)-SAT can be solved in \( O(2^{\delta n}) \) time.
(Wait, what happened to the \( \text{poly}(n) \) factor in the running time? Why can we omit it?)
Note the difference between the above statement, and SETH.

We can improve the dependence on \( k \) slightly with a cleverer branching:

**Theorem 2.2** \( k \)-SAT on \( n \) variables is in \( 2^{n-n/O(2^k)} \) time.

**Proof.** This is based on the branching/backtracking algorithm of Monien and Speckenmeyer [MS85].

**Algorithm A(F):** // \( F \) is a \( k \)-CNF formula
If \( F \) has no clauses, return SAT. If \( F \) has an "empty" clause (clause set false), return UNSAT.
If \( F \) is 2-CNF, solve SAT in polytime and return the answer.
If there is a 1-CNF clause \( (x) \), call \( \text{A} \) on \( F|_{x=1} \).
Take shortest clause \( (x_1 \lor \ldots \lor x_L) \).
Call \( \text{A} \) on \( F|_{x_1=0}, F|_{x_1=1}, \ldots, F|_{x_1=0,...,x_{i-1}=0,x_i=1}, \ldots, F|_{x_1=0,x_2=0,...,x_L=1} \).
If any of the \( L \) calls says SAT, then return SAT; else return UNSAT.

**Recurrence for the running time:** \( T(n) \leq \sum_{i=1}^{k} T(n - i) + O(\text{poly}(n)) \).

**Exercise 2.2** Why is this the running time recurrence? Why is this algorithm correct?

**How to solve it?** We will guess that \( T(n) = 2^{\alpha n} \) for some parameter \( \alpha > 0 \), and we’ll try to find \( \alpha < 1 \) that satisfies the recurrence. Inductively, we want the following to be true:
\[
T(n) \leq \sum_{i=1}^{k} T(n - i) \leq \sum_{i=1}^{k} 2^{\alpha(n-i)} \leq 2^{\alpha n}.
\]

In particular, we want \( \sum_{i=1}^{k} 2^{\alpha(n-i)} \leq 2^{\alpha n} \). Dividing both sides by \( 2^{\alpha n} \), this is \( \sum_{i=1}^{k} 2^{-\alpha i} \leq 1 \). We need to find an \( \alpha \) that will satisfy this inequality. By the usual expression for the sum of a geometric series, this inequality is equivalent to
\[
(1 - 2^{-\alpha(k+1)})/(1 - 2^{-\alpha}) - 1 \leq 1.
\]
Manipulating this around, we get that the above is equivalent to
\[
(1 - 2^{-\alpha(k+1)})/(1 - 2^{-\alpha}) \leq 2 \iff 1 - 2^{-\alpha(k+1)} \leq 2 - 2^{1-\alpha} \iff 2^{1-\alpha} - 2^{-\alpha(k+1)} \leq 1.
\]
Being incredible guessers, let's try \(\alpha = 1 - \log_2(e)/(5 \cdot 2^k)\). Then
\[
2^{1-\alpha} - 2^{-\alpha(k+1)} = 2^{\log(e)/(10 \cdot 2^k)} - 2^{-\alpha(k+1)(1 - \log(e)/(5 \cdot 2^k))}
\]
which equals
\[
e^{1/(5 \cdot 2^k)} - 2^{-(k+1)(1 - \log(e)/(5 \cdot 2^k))} < 1 + 3/(10 \cdot 2^k) - 2^{-(k+1)(1 - \log(e)/(5 \cdot 2^k))} < 1,
\]
where we used the inequalities \(e^x < 1 + 3x/2\) for all \(x \in (0, 1)\) and \(2^{(k+1)\log(e)/(5 \cdot 2^k)}/2^{k+1} > 3/(10 \cdot 2^k)\) for all \(k \geq 1\).

\[\square\]

**Exercise 2.3** Is there a cleaner derivation of \(\alpha \leq 1 - O(1/2^k)\)?

In general, the following is useful for analyzing backtracking algorithms:

**Theorem 2.3** Every recurrence of the form
\[
T(n) \leq T(n - k_1) + T(n - k_2) + \ldots + T(n - k_i) + O(\text{poly}(n))
\]
has as a solution
\[
T(n) \leq O(r(k_1, \ldots, k_i)^n \cdot \text{poly}(n)),
\]
where \(r(k_1, \ldots, k_i)\) is a positive root of the expression \(P(x) = 1 - \sum_{j=1}^{i} x^{-k_j}\).

For example, consider
\[
T(n) \leq T(n - 1) + T(n - 2) + O(\text{poly}(n)).
\]
By the above theorem, we want to find positive \(x\) satisfying \(1 - 1/x - 1/x^2 = 0\). (Ideally, we want the smallest such \(x\) too, to optimize the running time bound!) This is equivalent to \(x^2 - x - 1 = 0\) which is the same as \(x(x - 1) = 1\). Solutions for \(x\) are \(x = 1.618 \ldots, -0.618 \ldots\), so \(T(n) \leq O(1.618^n)\).

**Exercise 2.4** Think about how you might prove the above theorem.

### 3 Improved Algorithms for \(k\)-SAT

In this section, we will give the shortest proof we know that \(k\)-SAT has an \(2^{n - n/O(k)}\)-time algorithm.

**Theorem 3.1** \(k\)-SAT on \(n\) variables is in \(2^{n - n/O(k)}\) time.

This is the essentially the best dependence on \(k\) in the exponent that we know (to date). Improving on the exponent of \(n(1 - 1/O(k))\) is a major open problem! For example, is there an algorithm with exponent \(n(1 - \log(k)/O(k))\)? In the literature, there is a hypothesis called Super-SETH [VW19] which posits that there is no unbounded function \(f : \mathbb{N} \rightarrow \mathbb{N}\) such that \(k\)-SAT can be solved in \(2^{n - f(k)n/k}\) time. (Of course, Super-SETH implies SETH.) Ryan probably gives \(\leq 10\%\) likelihood that Super-SETH is true. (For other estimates, see [Wil19].)
Local search and random walks. Schoening [Sch99, Sch02] obtained an improved solution of $k$-SAT using an entirely different strategy. It is based on an earlier local search / random walk algorithm for solving 2-SAT, due to Papadimitriou [Pap91].

**Theorem 3.2 (Pap91)** There is a randomized algorithm for 2-SAT running in $\text{poly}(n)$ time.

We’ll discuss the theorem next time in more detail.

**References**


