Boolean Satisfiability
Lecture 3: Faster-than-2^n algorithms (Part II)

Edward A. Hirsch*

April 14, 2026

Lecture 3

In this lecture we finish studying algorithms that are able to solve ***k*-SAT** (and, to a certain extent, **SAT**) faster than in 2^n steps.

Contents

1 Derandomization of PPZ **2**
1.1 General plan 2
1.2 Permutations: Small sample space 3
1.3 Derandomizing the guesses 4

2 Randomized Local Search: another popular approach **5**
2.1 Experimental methods 5
2.2 Schöning's Random Walk algorithm 7
2.3 Derandomization: Deterministic local search 9

3 What if it is not *k*-SAT? **11**

4 A note on quantum algorithms **13**

5 Takeaway (the summary of two lectures) **13**

Historical notes and further reading **14**

*Ariel University, http://edwardahirsch.github.io/edwardahirsch

1 Derandomization of PPZ

1.1 General plan

Recall the PPZ algorithm from the previous lecture:

Algorithm 1 (Paturi, Pudlák, Zane).

```
Initialize A[1..n] by *
Pick a random permutation π ∈ S_n                                  // randomness #1
For i = 1,...,n
    If A[x_π(i)] = *
    Then
        A[x_π(i)] := random({0,1})                                 // randomness #2
        F := F[x_π(i) ← A[x_π(i)]]
        Perform unit clause elimination (updating A)
If F = True then answer ‘‘yes’’ else answer ‘‘no’’
```

This algorithm has two sources of randomness: the permutation and the choice of the next value. Recall that this algorithm can be viewed also as a decoding procedure for the "compressed" assignment consisting of the random values that we pick. Our deterministic version will

- Enumerate all the permutations — but $n!$ is too large, so we will need a smaller set $\sigma \in S_n$ of permutations with similar properties.
- Enumerate all the compressed assignments — but this is not that simple as we do not know how isolated is the assignment we are decoding, so we do not know its length; so we need a tweak.

We will deal with these two items in the two following subsections.

Ideally, we would like to see it like this (but, again, this plan will be tweaked):

Algorithm 2 (Informal derandomization plan for PPZ).

```
Initialize A[1..n] by *
For all permutations π ∈ σ ⊆ S_n
  For all B ∈ {0,1}^???
    Let j := 1
    For i = 1,...,n
      If A[x_π(i)] = * then
        A[x_π(i)] := B[j]
        j := j + 1
        F := F[x_π(i) ← A[x_π(i)]]
        Perform unit clause elimination (updating A)
```

1.2 Permutations: Small sample space

We need a polynomial-size set of permutations of n elements, $\boldsymbol{\sigma} \subset S_n$, such that

$$\forall s = (s_1, \ldots, s_k) \in [1\,..\,n]^k \;\; \forall i \in [1\,..\,k] \;\; \Pr_{\pi \in random(\boldsymbol{\sigma})} \{\pi(s_i) = \max_{1 \leqslant j \leqslant k} \pi(s_j)\} = 1/k$$

(s is a vector of numbers of variables that can potentially occur together in a clause). Such set of permutations with a probability measure on it is called a **max-k-wise independent** sample space (the literature talks usually about the symmetric notion of **min-k-wise independent** space). Small (polynomial-size) sample spaces like that are known to be constructible in polynomial time (the probability distribution does not need to be uniform, and approximate independence, that is, $1/k + \varepsilon(n)$ is also frequently sufficient). The general construction is beyond the scope of this course, however, we give a nice construction (exact and uniform) for $k = 3$ below. (This material is optional — not required for passing the course.)

MIN-3-WISE PERMUTATIONS

We will give the construction for the case where $n = p+1$ for a prime number p (it can be generalized to arbitrary n by noticing that there is always at least one prime between $n-1$ and $2n-5$ (inclusive) because of Bertrand's postulate).

Consider the p-element field $\mathbb{Z}_p$. Consider the the projective line $\boldsymbol{P}(\mathbb{Z}_p)$ over it, that is, merge the nonzero pairs from $(\mathbb{Z}_p \times \mathbb{Z}_p) \setminus \{(0,0)\}$ using the equivalence relation $(x,y) \sim (x',y')$ iff $xy' = x'y$. There are exactly $p+1 = n$ equivalence classes.

Thus a point of the projective line has the form $(i \cdot t,\ j \cdot t)$ where $t,\ ij \neq 0$.
For example, for $p = 3$ we have four points:

$$\begin{array}{ccll} (0,1) & \sim & (0,2) & // \ (0,1)\cdot t \\ (1,0) & \sim & (2,0) & // \ (1,0)\cdot t \\ (1,1) & \sim & (2,2) & // \ (1,1)\cdot t \\ (1,2) & \sim & (2,1) & // \ (1,2)\cdot t \end{array}$$

We will permute these classes using invertible linear transformations of $\mathbb{Z}_p \times \mathbb{Z}_p$, that is, functions of the form $f((x,y)) = (ax+by, cx+dy)$ with $ad - bc \neq 0$. There is a polynomial number of them (namely, $(p+1)p(p-1)$).

It is easy to check that for any two triples of the points of the projective line, there exists exactly one transformation that maps the first triple to the second triple,

$$\forall\, (i,j,k), (i',j',k') \in \boldsymbol{P}(\mathbb{Z}_p) \times \boldsymbol{P}(\mathbb{Z}_p) \times \boldsymbol{P}(\mathbb{Z}_p) \;\; \exists! f \text{ s.t. } (i,j,k) = (f(i'), f(j'), f(k')).$$

So such transformations, being applied to triples coordinate-wise, permute the triples.

The situation is absolutely symmetric, in particular, for random f each of the three points $f(i), f(j), f(k)$ has the same chance to be the last one among them, so it is $1/3$.

1.3 Derandomizing the guesses

Now that we can find an optimal permutation, it remains to try all "compressed" assignments. Given that we do not know how much our satisfying assignments are isolated, we choose $\varepsilon \in (0; \frac{1}{2})$ (to be determined later) and try two possibilities:

1. There is a "low-weight" satisfying assignment containing fewer than εn values "1".
2. Otherwise, think about a satisfying assignment with the fewest values "1" possible. It is isolated in all the directions where it has the value "1", and we can assume that there are more that εn such values. Thus, there must be an εn-isolated satisfying assignment (of course, if the formula is satisfiable at all).

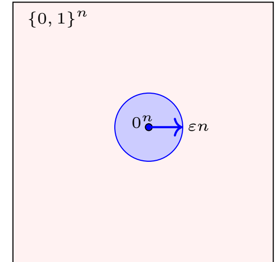
This is formalized in the following algorithm:

Algorithm 3 (PPZ, derandomized).

```
// Search in the ball of radius εn centered at the all-0 assignment
For each assignment A with ⩽ εn ''1''s
  If F[A] = True then return ''yes''.

// Now if it does not help...
For all permutations in σ,
  For all strings s ∈ {0,1}^{n(1−ε/k)+1},
    // Try to decode s...
    Initialize A[1..n] by *
    Let j := 1
    For i = 1,...,n
      If A[x_π(i)] = * then
        A[x_π(i)] := s[j]
        j := j + 1
        F := F[x_π(i) ← A[x_π(i)]]
        Perform unit clause elimination on F (updating A)
    If F = True then return ''yes''

Return ''no''
```



By making use of the estimation (say, by Stirling's formula) for the binomial coefficients, one can check that the volume (the number of assignments) of the Hamming ball of radius ϵn is $\widetilde{\Theta}(2^{H(\varepsilon)\cdot n})$, where $\widetilde{\Theta}(\ldots)$ is $\Theta(\ldots)$ ignoring the polynomial factors.

Let us estimate the running time of the algorithm:

- The first phase of the algorithm takes time $\widetilde{O}(\boldsymbol{2^{H(\varepsilon)\cdot n}})$, where $H(\varepsilon) = -\varepsilon \log_2 \varepsilon - (1-\varepsilon)\log_2(1-\varepsilon)$ is the binary entropy function.
- The second phase takes time $\widetilde{O}(\boldsymbol{2^{n(1-\varepsilon/k)}})$.

To optimize their sum, let us take $\boldsymbol{\varepsilon}$ such that $H(\varepsilon) = 1 - \varepsilon/k$. As $k \to \infty$, it leads to a $\widetilde{O}(2^{n(1-c_k)})$-time algorithm for $c_k = \frac{1}{2k} - o(\frac{1}{k})$ (computing this asymptotics is left as an exercise).

2 Randomized Local Search: another popular approach

2.1 Experimental methods

GSAT algorithm. Since the beginning of 1990s, there was a steady interest in designing heuristic algorithms that use local search, that is, perform a walk in the Boolean cube changing one variable at a time, in order to improve the current assignment and eventually reach the satisfying assignment. The following simple greedy algorithm starting from a random assignment was suggested by Bart Selman, Hector Levesque, and David Mitchell; they investigated it experimentally with some positive results.

Algorithm 4 (GSAT).
Input:
A formula F in CNF, integers `MAXFLIPS` and `MAXTRIES`.
Output:
A truth assignment for the variables of F, or “no”.

```
Repeat MAXTRIES times:

  -- Pick an assignment A at random.

  -- Repeat MAXFLIPS times:

     -- If A satisfies F, then output A.

     -- Choose a variable x in F such that A^x satisfies the maximal possible
        number of clauses of F.

     -- A := A^x.

Output ``no''.
```

A lower bound for GSAT. As it happens with many other local search algorithms that try to optimize some goal function greedily (the number of satisfied clauses in our case), they may end up in a local maximum that is not at all the global maximum. (In practice, they even more frequently get stuck at a large “plateau” where the goal function does not change.)

It is not difficult to show an exponential lower bound for GSAT and some other similar algorithms. Namely, we will give a series of uniquely satisfiable formulas F_n for which the only chance for GSAT to reach the satisfying assignment is to choose it or its direct neighbour initially (thus with probability $(n+1)/2^n$). Our formula F_n contains n variables and three groups of clauses (a clause is described as a set of literals); the first and the third group are obtained using shifting the variable numbers by 1 modulo n:

$$
\begin{array}{lllll}
 & \{x_1, & \overline{x_2}, \ldots, \overline{x_{n-3}}, & \overline{x_{n-2}}, \overline{x_{n-1}}\} \\
 & \{x_1, & \overline{x_2}, \ldots, \overline{x_{n-3}}, & \overline{x_{n-1}}, \overline{x_n}\} \\
 & \{x_1, & \overline{x_2}, \ldots, \overline{x_{n-3}}, & \overline{x_n}, \overline{x_{n-2}}\} \\
\{x_1, \overline{x_2}\} & & & \\
\{x_2, \overline{x_3}\} & & & \\
\ldots & \{x_2, & \overline{x_3}, \ldots, \overline{x_{n-2}}, & \overline{x_{n-1}}, \overline{x_n}\} \\
\{x_n, \overline{x_1}\} & \{x_2, & \overline{x_3}, \ldots, \overline{x_{n-2}}, & \overline{x_n}, \overline{x_1}\} \\
 & \{x_2, & \overline{x_3}, \ldots, \overline{x_{n-2}}, & \overline{x_1}, \overline{x_{n-1}}\} \\
 & \ldots & & \\
\{x_1, x_2\} & \{x_n, & \overline{x_1}, \ldots, \overline{x_{n-4}}, & \overline{x_{n-3}}, \overline{x_{n-2}}\} \\
 & \{x_n, & \overline{x_1}, \ldots, \overline{x_{n-4}}, & \overline{x_{n-2}}, \overline{x_{n-1}}\} \\
 & \{x_n, & \overline{x_1}, \ldots, \overline{x_{n-4}}, & \overline{x_{n-1}}, \overline{x_{n-3}}\}
\end{array}
$$

It follows from the first two groups (2-clauses) that the satisfying assignment is $x_1 = x_2 = \ldots = x_n = 1$ (and it also obviously satisfies the third group).

Now look at the neighbourhood of the satisfying assignment.

- Every assignment at Hamming distance 1 from it falsifies 4 clauses.
- Every assignment at Hamming distance 2 from it falsifies 2 or 3 clauses.
- For every assignment B at distance 2, there is a direction (variable) x such that B^x falsifies the same number of clauses as B (one can take x adjacent to exactly one of the wrong values and not $\in \{x_1, x_2\}$). Obviously, B^x is at distance 3 from the satisfying assignment.

Thus assignments at Hamming distance 2 form an “insurmountable ring” around the satisfying assignment: once the algorithm reaches one of them, it is forced to make the next step in the wrong direction.

WalkSAT: a heuristic fix to the bug. The authors of GSAT suggested the following tweak that allows the algorithm to escape local maxima — and it performed much better experimentally. Among other heuristics, it uses random moves.

Algorithm 5 (WalkSAT, mixes greedy and random moves).

Input: F in CNF, integers `MAXFLIPS` and `MAXTRIES`, probability p.
Output: A truth assignment for the variables of F, or “no”.

`Define` $\beta(A, x)$`: the number of clauses satisfied by` A `but not by` A^x`.`

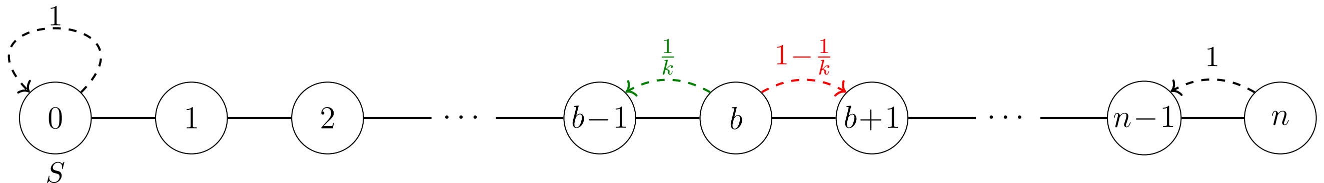
`Repeat MAXTRIES times:`

```
-- Pick an assignment A at random.
-- Repeat MAXFLIPS times:
    -- If A satisfies F, then output A.
    -- Choose an unsatisfied C ∈ F at random.
    -- If there is x in C such that β(A,x) = 0,
       Then
       -- choose this x
       Else
       -- with probability  p choose x from C to minimize β(A,x)    //greedy move
       -- with probability  1-p choose x from C at random            //random move
    -- A := A^x.
Output "no".
```

2.2 Schöning's Random Walk algorithm

Let us consider a much simplified version of WalkSAT: the algorithm that takes an unsatisfied clause C in the current (unsatisfying) assignment A, chooses a variable from C at random, and flips it.

Think about a specific satisfying assignment S. Certainly, at least one of the variables in C has different values in A and S, so with probability at least $1/|C|$ the Hamming distance between A and C decreases by 1 (and otherwise it increases by 1). This can be viewed as the following random walk on a line with integer states (let $k = |C|$), the state number being the distance from S:



Warm-up: the 2-SAT case. Even though we know a polynomial-time deterministic algorithm for **2-SAT**, let us analyze the behaviour of this very simple random walk algorithm on 2-CNF as a warm-up (it was suggested by Christos Papadimitrou). Let x_k be the expectation of the number of steps needed to reach $b-1$ from b. By the linearity of expectation, $x_b = \frac{1}{2}\cdot 1 + \frac{1}{2}\cdot(1+x_{b+1}+x_b)$, that is, $x_{b+1} = x_b + 2$. Even in the very unfortunate case where the initial randomly chosen assignment is in the state n, the expectation of the number of steps to reach 0 is $x_n + x_{n-1} + \ldots + x_1 = n^2$.

The 3-SAT case. In this case we do care about the state where the initial assignment lands. Also we limit the length of the random walk.

Algorithm 6 (Schöning, 3-SAT version).
Input: F in 3-CNF with n variables,
Output: A truth assignment for the variables of F, or "not found".

```
-- Pick an assignment A uniformly at random.

-- Repeat at most n times:

    -- If A satisfies F, then output A.

    -- Choose an unsatisfied C ∈ F.

    -- Choose a variable x from C at random.

    -- A := A^x.

-- Output ‘‘not found’’.
```

Assume w.l.o.g. that 3 divides n. Let us hope that we land in the state $n/3$ and compute two probabilities: to land in this state, and to reach 0 provided we land in $n/3$.

The first probability is $\binom{n}{n/3}/2^n$.

The second probability can be computed as the number of paths (patterns of going in the right and in the wrong direction at each step) on the 2-dimensional integer grid that correspond to walks from $n/3$ to 0, where

- the horizonal axis: time, that is, the number of steps, going from 0 to n,
- the vertical axis: distance from S, going from 0 to n,
- the path starts at $(0, n/3)$ and ends at $(n, 0)$,
- the path does not intersect the horizontal axis,
- every edge goes either northeast $(i,j) \to (i+1, j+1)$ (wrong direction) or southeast $(i,j) \to (i+1, j-1)$ (right direction),
- the last edge is going, of course, southeast $(n-1, 1) \to (n, 0)$.

Let P be the number of northeast edges and Q is the number of southeast edges; $P + Q = n$ and $P - Q = n/3$. It is not difficult to see (some pictures are in the slides) that the number of such paths is

$$\binom{P+Q-1}{P-1} - \binom{P+Q-1}{P} = \frac{P-Q}{P+Q}\binom{P+Q}{P} = \frac{1}{3}\binom{n}{2n/3}.$$

The probability for such a path to happen can be estimated as $(1/3)^P(2/3)^Q = (1/3)^{2n/3}(2/3)^{n/3}$.

Overall, the chances to come from $n/3$ to 0 are $\frac{1}{3}\binom{n}{2n/3} \cdot (1/3)^{2n/3}(2/3)^{n/3}$, and thus the overall success probability of the algorithm is at least

$$\frac{\binom{n}{n/3}}{2^n} \cdot \frac{1}{3}\binom{n}{2n/3} \cdot \frac{1}{3^{2n/3}}\frac{2^{n/3}}{3^{n/3}} = \frac{1}{3} \cdot \binom{n}{n/3}^2 \cdot \frac{1}{3^n \cdot 2^{2n/3}} = \widetilde{\Omega}\left(\left(\frac{2^{2H(1/3)}}{3 \cdot 2^{2/3}}\right)^n\right).$$

The base of this exponent is

$$\frac{3^{2/3} \cdot (3/2)^{4/3}}{3 \cdot 2^{2/3}} = \frac{3}{4}.$$

Repeating the procedure $\widetilde{O}((4/3)^n)$ times guarantees that we find a satisfying assignment with probability at least 1/2.

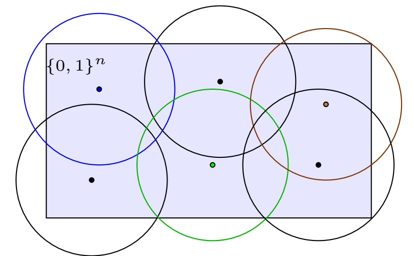
This algorithm can be easily generalized to ***k*-SAT** achieving the running time $\widetilde{O}((2-\frac{2}{k})^n)$ and for **CSP** achieving the running time $\widetilde{O}\left(\left(|D|(1-\frac{1}{k})\right)^n\right)$ for domain D and relation arity k.

2.3 Derandomization: Deterministic local search

Schöning's algorithm hopes to choose the initial assignment in the Hamming ball of radius $n/3$ centered in a satisfying assignment and then performs a random walk. Let us do both things deterministically.

The derandomized algorithm.

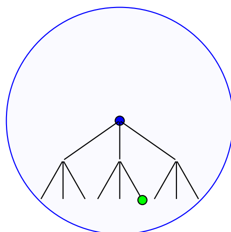
Consider a set of Hamming balls of a certain radius R covering our search space (this is called a **covering code** or radius R and length n). To start in a good position, **try centers** of these balls.

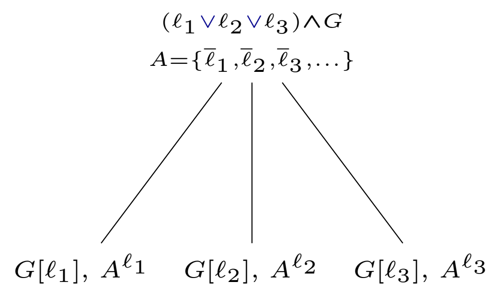


Let $0 < \rho < 1/2$. For every n, we can efficiently construct a covering code of length n, radius at most ρn, and cardinality $\widetilde{O}(2^{(1-H(\rho))n})$. (We discuss the constructions below.)

Start the search in a center of the ball. Let us talk about ***k*-SAT**. When performing our **local search within a ball**, we

- take an unsatisfied clause $\ell_1 \vee \ell_2 \vee \ldots \vee \ell_t$ and try all t possibilities to flip a variable in it thus developing a recursion tree with $\leqslant k$ children of every node,
- make substitutions to the formula (as we never come back: contrary to the randomized version, we explore all possibilities and thus do not need to fix errors), so our search is somewhat similar to the DPLL algorithm,
- stop at depth R.





Thus our tree contains at most k^R leaves and the running time of the search in one ball is $\widetilde{O}(k^R)$. Let $R = n/(k+1)$. Then the overall running time of the algorithm is at most $\widetilde{O}$ of

$$\begin{aligned}
&2^{(1-H(1/(k+1)))n} \cdot k^{n/(k+1)} \\
&= 2^{(1-\log(k+1)/(k+1)+(1-1/(k+1))\log(1-1/(k+1)))n} \cdot 2^{n\log k/(k+1)} \\
&= 2^{(1-\log(k+1)/(k+1)+(k/(k+1))\log(k/(k+1)))n} \cdot 2^{n\log k/(k+1)} \\
&= 2^{n(1-\log(k+1)/(k+1)+\underline{(k/(k+1))\log k}-(k/(k+1))\log(k+1))+\underline{\log k/(k+1)})} \\
&= 2^{n(1+\underline{\log k}-\log(k+1))} = (2-\tfrac{2}{k+1})^n .
\end{aligned}$$

How to construct a covering code of size $2^{(1-H(r/n))n}$? First of all, let us check that the required code *exists*. (This fact, and more combinatorial facts about covering codes can be found in the book G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein. *Covering Codes*, volume 54 of *Mathematical Library*. Elsevier, 1997.)

Let $\delta(A,B)$ denote the Hamming distance between A and B, and let $V(d,r) = |\{A : \delta(A,\vec{0}) \leqslant r\}| = (2^{H(r/d)d})$ be the volume of a ball of radius r.

Lemma 1. *There exists a code of length d, radius at most r, and size $\leqslant \lceil d \cdot 2^d/V(d,r)\rceil$.*

Proof. We use the probabilistic method. Choose $d \cdot 2^d/V(d,r)$ random strings $\in \{0,1\}^d$. Let $a \in \{0,1\}^d$. Then $\Pr_{b\in U(\{0,1\}^d)}\{a \text{ is in the ball centered at } b\} \geqslant V(d,r)/2^d$.

The probability that a misses all our balls is $\left(1 - V(d,r)/2^d\right)^{d\cdot 2^d/V(n,r)} \leqslant \boldsymbol{e}^{-d}$. □

The problem is that a code is a huge subset of $\{0,1\}^n$, and it is too difficult even to verify properties of such a set that involve distances to all possible assignments.

The solution is to construct a covering code S for small length d and take a direct product of such codes, that is,

$$\underbrace{S \times S \times \ldots \times S}_{n/d} = \{w_1 w_2 \ldots w_{n/d} : w_i \in S\}.$$

(We identify a vector $(a_1, a_2, \ldots, a_t) \in \{0,1\}^t$ with a bit string $a_1 a_2 \cdots a_t$.)

For example, by taking a direct product of many copies of the code $\{00000, 01010, 10101\}$

$$\begin{pmatrix}00000\\01010\\10101\end{pmatrix} \times \begin{pmatrix}00000\\01010\\10101\end{pmatrix} \times \cdots \times \begin{pmatrix}00000\\01010\\10101\end{pmatrix}$$

we will get a code like

$$\begin{pmatrix}0000000000\ldots00000\\0000001010\ldots01010\\1010100000\ldots01010\\\text{et cetera}\end{pmatrix}.$$

It is easy to see that relative to n, the parameters of the direct product code are the same (*if all the numbers below are magically integers!!!*) as for the small code.

Let d be the length of the small code, ρd be its radius, and the cardinality is $2^{\alpha d}$. Then the length of the product is $n/d \times d = n$, the radius is $n/d \cdot \rho d = \rho n$, and the cardinality is $\left(2^{\alpha d}\right)^{n/d} = 2^{\alpha n}$.

There are two approaches to find the small code:

- $d = O(1)$, then rounding errors add some ε (depending on the constant in $O(\ldots)$) to the exponent.
- $d = n/10$, then to construct the code we can use a greedy approximation algorithm for Set Cover. The universum here is $U = \{0,1\}^d$, and the sets $S \subset U$ that we can choose are all the balls of radius ρd. At each step the algorithm adds a ball that covers as many yet-uncovered elements of U as possible. We can recalculate these numbers for all the 2^d balls at every step at the cost of $\widetilde{O}(2^d)$ per ball; given that the number of iterations is at most 2^d, this greedy algorithm constructs a code in time $\widetilde{O}(2^{3d})$, which for $d = n/10$ is even better than $(2 - 2/(k+1))^n$ taken by the main algorithm. The cardinality of the code is within a d factor of the optimal value.

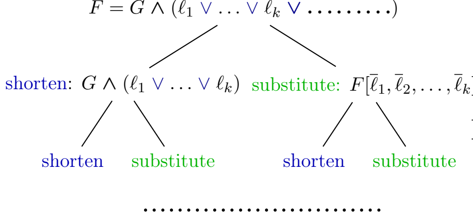
3 What if it is not k-SAT?

Can we do better than 2^n if the clauses can have arbitrary length? The following algorithm shows that yes we can; however, the base of the exponent will depend on the formula "density", that is, on the ratio of the number of clauses to the number of variables.

The algorithm actually reduces the problem to **k-SAT**, where k is to be determined later based on the running time bound of the **k-SAT** algorithm that will be used as a subroutine.

Algorithm 7 (Calabro, Impagliazzo, Paturi).

```
Clause-shortening (F,k,A):                              // A is a k-SAT algorithm
  if F is in k-CNF then return A(F)
  else (so there is C ∈ F s.t. |C| > k)
    take distinct ℓ_1,...,ℓ_k ∈ C
    if Clause-shortening(F ∪ {ℓ_1 ∨ ... ∨ ℓ_k}) \ {C}, k, A) = "yes"
      then return "yes"
    elif Clause-shortening(F[¬ℓ_1,...,¬ℓ_k], k, A) = "yes"
      then return "yes"
    else return "no"
```



The algorithm chooses any long clause C in F and examines recursively two cases:

1. One of the first k literals in C is satisfied by some satisfying assignment.
2. In all satisfying assignments all the first k literals in C are false.

For the analysis, it gives us two branches that we call "left" and "right":

1. In the left branch we eliminate one long clause.
2. In the right branch we eliminate k variables.

This algorithm's run corresponds to a recursion tree where the leaves are the applications of the $\boldsymbol{k}$**-SAT** algorithm.

Consider the input formula F that has n variables and m clauses. If a path goes through L left branches and R right branches, then $L \leqslant m$ and $R \leqslant n/k$. Thus the total number of paths (root→leaf) with R right branches does not exceed $\binom{L+R}{R} \leqslant \binom{m+n/k}{R}$.

Assume that $\boldsymbol{k}$**-SAT** is solvable in time $\widetilde{O}(2^{a_k n})$ by a randomized algorithm. By repeating it a polynomial number of times we can assume that its probability of error is smaller than $2^{-100n^2m^2}$, thus with overwhelming probability it will not err in any leaf of our branching tree.

Assume $m \geqslant n/k$. Since $\mathcal{A}$ is applied to a formula in k-CNF where at most $n - kR$ variables are still remaining, the total runnning time of our algorithm including the time taken by $\mathcal{A}$ is at most

$$\sum_{R=0}^{n/k} \binom{m+n/k}{R} \cdot \mathrm{time}_{\mathcal{A}}(m, n-kR) \leqslant \sum_{R=0}^{n/k} \binom{m+n/k}{R} \cdot 2^{a_k(n-kR)}$$

$$= 2^{a_k n} \cdot \sum_{R=0}^{n/k} \binom{m+n/k}{R} \cdot 2^{-a_k kR} \leqslant \text{ (now use } \textstyle\sum_{i=0}^{N} \binom{N}{i} a^i = (1+a)^N)$$

$$\leqslant 2^{a_k n}(1+2^{-a_k k})^{m+n/k} = 2^{a_k n} \underbrace{\left((1+2^{-a_k k})^{2^{a_k k}}\right)}_{\leqslant \boldsymbol{e}}{}^{2^{-a_k k}(m+n/k)}$$

$$\leqslant 2^{a_k n} \boldsymbol{e}^{2^{-a_k k}(m+n/k)} \leqslant 2^{a_k \boldsymbol{n} + (2\boldsymbol{m} \log_2 \boldsymbol{e})/2^{a_k k}} \text{ (as } m \geqslant n/k).$$

Recall that PPZ solves $\boldsymbol{k}$**-SAT** in time $\widetilde{O}(2^{n(1-1/k)})$, that is, $a_k \leqslant 1 - 1/k$. Choose $k = \lceil 2\log_2(m/n) \rceil + c$, where c is a positive integer constant to be determined later, and let us show the upper bound $\widetilde{O}\left(2^{n\left(1 - \frac{1}{O(\log(m/n))}\right)}\right)$ on the running time of our algorithm. Indeed,

$$a_k n + (2m \log_2 \boldsymbol{e})/2^{a_k k} \leqslant a_k n + (4m)/2^{a_k k} \leqslant n\left(1 - \tfrac{1}{k}\right) + \tfrac{4m}{2^{k-1}} \leqslant n\left(1 - \frac{1}{2\log_2(m/n)+c} + \frac{1}{(m/n)\,2^{c-3}}\right). \quad (*)$$

Let $m > n$. It is easy to see that indeed $m \geqslant n/k$ as required before, and that[1] for a large enough constant c,

$$(m/n)\,2^{c-3} - 2\log_2(m/n) - c \geqslant \alpha \log_2(m/n)$$

[1] Denote $p = m/n$. Consider $f(p) = (p\,2^{c-3} - 2\log_2(p) - c)/\log_2 p$. We need $(2^{c-3}p - c)/\log_2 p \geqslant const > 2$. For $p > 1$, the function $p/\log_2(p)$ is decreasing until $p = \boldsymbol{e}$ (where it equals $\boldsymbol{e}\ln 2$) and is increasing afterwards (you can check the derivative).

for certain constant $\alpha > 0$, which means that[2] $(*) \leqslant n\left(1 - \frac{1}{O(\log(m/n))}\right)$.

4 A note on quantum algorithms

(This material is optional — not required for passing the course.)

Can quantum algorithms solve ***k*-SAT** faster?

Grover's quantum algorithm speeds up the search for the following problem that has the complexity $\Omega(2^n)$ in both the deterministic and randomized settings:

Problem (black-box database search):
For any black-box $f\colon S \to \{0,1\}$,
finds $x \in S$ s.t. $f(x) = 1$ using $O(\sqrt{|S|})$ queries to f.

This immediately gives us an $\widetilde{O}(2^{\sqrt{n}})$-time algorithm for **SAT**.

Once we convert one of our improved algorithms into such f with smaller-than-2^n search space S, we will get an even better algorithm. For many known ***k*-SAT** algorithms this is possible.

On the negative side, this (black-box) way quantum algorithms could not help more than we already see: $O(\sqrt{|S|})$ is also the lower bound on the number of queries even in the quantum setting.

5 Takeaway (the summary of two lectures)

We studied three approaches to solving ***k*-SAT** faster than 2^n:

- the DPLL split-and-simplify approach,
- the PPZ random permutations approach,
- Schöning's random walk algorithm.

We also studied the clause-shortening approach that reduces **SAT** to ***k*-SAT** and gives a less-than-2^n bound (though not contant^n).

There is a number of results combining and improving these algorithms resulting in both randomized and deterministic algorithms with better and better running time bounds. How far can we go? Can we solve **3-SAT** in time 1.3^n? 1.2^n? $(1+\varepsilon)^n$? What about the asymptotics of

Suppose $\log_2(p) \geqslant 100$. Then $(2^{c-3}p - c)/\log_2 p \geqslant 2^{100+c-3}/100 - c/100$, which is greater than 5 for all integer $c > 0$.

Now assume $\log_2(p) < 100$. Take $c = 100$, then $(2^{c-3}p - c)/\log_2 p \geqslant (2^{c-3}\boldsymbol{e}\ln 2 \cdot \log_2 p - c)/\log_2 p \geqslant 2^{97}\boldsymbol{e}\ln 2 - 100/\log_2 p > 5$.

[2] Caution! The constant in $O(\dots)$ is relative to $n \to \infty$, it has nothing to do with $\log(m/n)$, it is valid for all values of $m > n$ (and of this positive-valued logarithm).

this constant w.r.t. k for $\boldsymbol{k}$-**SAT**? What about other versions of **SAT**? In the next lecture we will at least connect such problems to each other.

Historical notes and further reading

See Lecture 2.