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

Edward A. Hirsch[*]

January 8, 2024

Lecture 2

In this lecture we start studying algorithms that are able to solve ***k*-SAT** (and, to a certain extent, **SAT**) faster than in 2^n steps. Some of them also form the base for practical algorithms (SAT solvers).

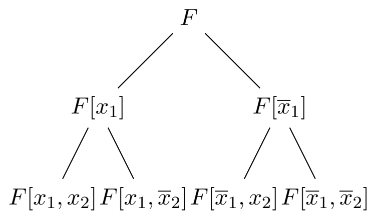
Contents

1 DPLL: Divide-and-conquer

It is obvious that we can solve **SAT** in 2^n steps, just consider the two possible assignments for every variable. In the context of **SAT**, reducing the problem for a formula F to several simpler

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

formulas $F_1, \ldots, F_k$ is called **branching** or **splitting**. (In particular, splitting over a variable x means reduction to the two formulas $F_1 = F[x \leftarrow 1]$ and $F_2 = F[x \leftarrow 0]$.)



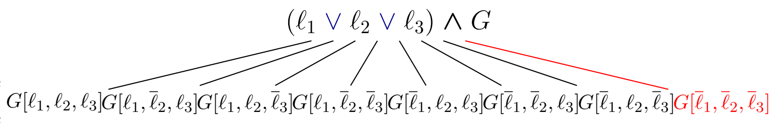
However, **can we do better than 2^n?**
Important note: In what follows we will ignore polynomial factors and pay attention to the exponential factor only. The following notation helps us stating it explicitly.

Definition 1. We write $f(n) = \widetilde{\boldsymbol{O}}(g(n))$ if $f(n) = O(g(n) \cdot \text{size}(F)^k)$ for some $k \in \mathbb{N}$. Here, "size" refers to the input length (the bit-size of the representation of F in the machine), while n refers to any complexity parameter (going to $+\infty$), such as the number of variables.

For example, we write $\widetilde{O}(2^n)$ when our algorithm performs 2^n steps, each step taking quadratic time $O(|F|^2)$.

1.1 The first attempt. Splitting over variables from the same clause

For a simple better-than-2^n algorithm, we start with solving **3-SAT** by "almost" exhaustive search. Namely, when we choose variables for splitting from the same clause, thus getting only 7 partial assignments out of the $2^3 = 8$ possible. The reason is that one assignment is definitely not satisfiable: the one that falsifies all literals of the chosen clause (see the picture). For **3-SAT**, we take a 3-clause. Then continue splitting the obtained formulas until we arrive at a formula that does not contain a 3-clause.


$(\ell_1 \vee \ell_2 \vee \ell_3) \wedge G$
$G[\ell_1, \ell_2, \ell_3]$
$G[\ell_1, \overline{\ell}_2, \ell_3]$
$G[\ell_1, \ell_2, \overline{\ell}_3]$
$G[\ell_1, \overline{\ell}_2, \overline{\ell}_3]$
$G[\overline{\ell}_1, \ell_2, \ell_3]$
$G[\overline{\ell}_1, \overline{\ell}_2, \ell_3]$
$G[\overline{\ell}_1, \ell_2, \overline{\ell}_3]$
$G[\overline{\ell}_1, \overline{\ell}_2, \overline{\ell}_3]$

We can easily analyze this algorithm. Let us estimate the number of leaves in the so-constructed branching tree.

Denote $\boldsymbol{T(F)}$:= the number of leaves in the tree for the formula F.
Also denote $T(n) := \max_{\text{vars}(F)=n} T(F)$, the number of leaves in the worst possible branching tree for a 3-CNF formula containing n variables.

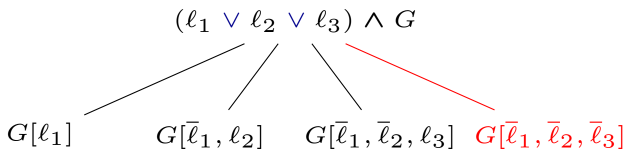
Since we remove 3 variables and get only 7 cases, we get the recurrency

$$T(n) \leqslant 7 \cdot T(n-3) \leqslant \ldots \leqslant 7^{\lceil n/3 \rceil} = O\left(\left(\sqrt[3]{7}\right)^n\right) = O(1.92^n).$$

If F does not contain a 3-clause, then its satisfiability can be checked in polynomial time. Thus the overall running time is $\widetilde{O}(1.92^n)$.

1.2 The second attempt. Grouping the cases in splitting over a clause

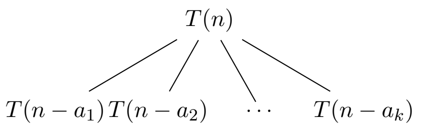
Let us do the same thing more efficiently. Indeed, let us be lazy and let us delay splitting of $G[\ell_1]$ for the future. Same with $G[\overline{\ell}_1, \ell_2]$. Look at the picture what we get.


$(\ell_1 \vee \ell_2 \vee \ell_3) \wedge G$
$G[\ell_1]$
$G[\overline{\ell}_1, \ell_2]$
$G[\overline{\ell}_1, \overline{\ell}_2, \ell_3]$
$G[\overline{\ell}_1, \overline{\ell}_2, \overline{\ell}_3]$

Thus the recurrency is now $T(n) \leqslant T(n-1) + T(n-2) + T(n-3)$. In fact, we can prove that $T(n) = O(1.84^n)$; to resolve such recurrencies fast, we prove a general lemma.

1.3 Branching tuples: Estimating the recurrency

Assume $k \geqslant 2$, $T(n) \leqslant \sum_{i=1}^{k} T(n - a_i)$.
Call $\vec{a} = (a_1, \ldots, a_k) \in \mathbb{N}_{\geqslant 1}^{k}$ a **branching tuple**.
Call $\chi_{\vec{a}}(x) = 1 - \sum_{i=1}^{k} x^{-a_i}$ its **characteristic polynomial**.



Note that for any $\vec{a}$, $\chi_{\vec{a}}(x) \underset{x \to 0+}{\to} -\infty$ and $\chi_{\vec{a}}(x) \underset{x \to +\infty}{\to} 1$, and $\chi_{\vec{a}}$ is monotone in between. Thus $\chi_{\vec{a}}$ has exactly one positive root, let us call this root the **branching number** and denote it by $\tau_{\vec{a}}$.

Lemma 1 (branching numbers)**.** *Let μ: {formulas} $\to \mathbb{R}_{\geqslant 0}$ be some complexity measure of formulas (for example, the number of variables).*

Consider a tree that splits formulas as $G \mapsto G_1, \ldots, G_k$, where $\forall i$ $\mu(G_i) \leqslant \mu(G) - a_i$. The number k and the branching tuple $\vec{a}$ can be different for different formulas present in the tree.

Let $\tau = \max_{\vec{a}} \tau(\vec{a})$ for all branching tuples $\vec{a}$ present in the tree, and let F be the formula in the root of the tree. Then $T(F) \leqslant \tau^{\mu(F)}$.

Proof. We proceed by induction. The base (a leaf L) is trivial: $1 \leqslant \tau^{\mu(L)}$ since μ is nonnegtive.

Now assume that the statement is true for all formulas smaller than F (that is, with fewer variables). Then

$$T(F) \leqslant \sum_{j=1}^{k} T(F_j) \overset{(ind)}{\leqslant} \sum_{j=1}^{k} \tau^{\mu(F_j)} \leqslant \sum_{j=1}^{k} \tau^{\mu(F)-a_j} = \tau^{\mu(F)} \cdot \sum_{j=1}^{k} \tau^{-a_j} = \tau^{\mu(F)}(1 - \chi_{\vec{t}}(\tau)) \leqslant \tau^{\mu(F)}.$$

The last inequality holds, because τ is the root of *some* characteristic polynomial present in the tree. If it corresponds to $\vec{t}$, then $\chi_{\vec{t}}(\tau) = 0$, otherwise the branching number for $\vec{t}$ is even smaller, and thus $\chi_{\vec{t}}(\tau) \geqslant 0$ by the monotonicity of $\chi_{\vec{t}}$. □

Remark 1. While we will be using natural μ such as the number of variables or the number of clauses, more complicated measures of complexity are also used in the literature, for example, (not necessarily nonnegative) linear combinations involving the number of 2-clauses and other good or bad things appearing in the formula. This is called amortized analysis, and it is out of scope of this course.

We can now apply the lemma to the algorithm in Sect. 1.2; solving the equation $1 - x^{-1} - x^{-2} - x^{-3} = 0$ numerically, we get $x \approx 1.839$, thus getting $O(1.84^n)$ leaves. Note that we could avoid referring to **2-SAT** $\in$ **P** and use 2-clauses for splitting as well, because for the respective recurrency $T(n) \leqslant T(n-1)+T(n-2)$ the branching number is even smaller, namely, it is the root of $1-x^{-1}-x^{-2}=0$, that is, $x^2-x-1=0$, and this is the golden ratio $\phi < 1.62 < 1.84$.

It would be great to split always over a 2-clause!

1.4 The third attempt: We can always find a 2-clause

Let us formulate the notion of a **DPLL** algorithm in general terms.

Algorithm 1 (DPLL = Davis, Putnam, Logemann, Loveland). The algorithm uses an external polynomial-time procedure Π for checking the satisfiability for formulas for a class $\mathcal{C}$ (for example, 2-CNF), and also uses two polynomial-time procedures *Split* and *Simplify*.
Method.

```
If $F \in \mathcal{C}$ then apply $\Pi$ to $F$
else
   1. Split:  Transform $F$ into formulas $F_1, F_2, \ldots, F_k$.
   2. Simplify:  For each $i = 1..k$,
          simplify $F_i$ (e.g., by unit propagation)
          and apply the algorithm recursively to the result.
          If any of the recursive calls returns ''yes''
             then return ''yes''
             else return ''no''.
```

The step "Split" reduces the problem to a constant number of subproblems such that $F \in \mathbf{SAT} \iff F_1 \in \mathbf{SAT} \vee \ldots \vee F_k \in \mathbf{SAT}$.

The step "Simplify" simplifies the problem without splitting, so for a formula F_i it results in a formula $\mathbf{Reduce}(F_i)$ such that $F_i \in \mathbf{SAT} \iff \mathbf{Reduce}(F_i) \in \mathbf{SAT}$.

Both steps are done in polynomial time.

Let us assume that $\mathbf{Reduce}(F_i) = F[A]$ for a certain partial assignment A, as it is the case if we only use substitutions (as we do) and unit clause elimination (which is also a substitution), which we will also include in the simplification procedure.

If $F[A]$ does not contain 2-clauses, then $F[A] \subseteq F$, because substitutions to a 3-CNF cannot result in new 3-clauses.

Thus in such a case we can replace F by $F[A]$ instead of splitting it, just add this rule to the simplification procedure as well.

Therefore, we can always

- either find a $(1,2)$-splitting over a 2-clause (it corresponds to a branching number $\phi = 1.61\ldots$),
- or we get a formula after the $F \mapsto F[A]$ reduction (without splitting), which removes at least one variable, so even after splitting the reduced formula over a 3-clause we get a $(2,3,4)$-splitting (related to the unreduced formula), with a much better branching number $1.46\ldots$.

The only exception is the very first formula, where we cannot guarantee anything better than a $(1,2,3)$-splitting over a 3-clause, but it only influences the constant in $O(\ldots)$. Overall, we achieve the running time $\widetilde{O}(\phi^n)$.

Here is how the final algorithm looks like:

Algorithm 2 (ϕ^n-time algorithm for 3-SAT).

```
Procedure $\textbf{Reduce}(F)$:
    Perform Unit-clause-elimination($F$)
    Do the step
       For every substitution $A$ to at most three variables
          If $F[A] \subseteq F$, then replace $F$ by $F[A]$
    Until it does not change the formula
    Return the changed (simplified) copy of $F$

Main Algorithm($F$):
    If $F$ is in 2-CNF then
    then
       apply a polynomial-time 2-SAT algorithm to $F$
    else
       Choose a 2-clause $\ell_1 \vee \ell_2$ in $F$ or (if none) a 3-clause $\ell_1 \vee \ell_2 \vee \ell_3$ in $F$.
       For every $i = 1, \ldots, k$
          Consider an assignment $A_i = \{\ell_i\} \cup \{\overline{\ell}_j \text{ for all } j < i\}$
          Construct $F_i = \textbf{Reduce}(F[A_i])$
          Apply the algorithm to $F_i$ recursively
       If any of the recursive calls returns "yes"
          then return "yes"
          else return "no"
```

1.5 Other tricks to extend DPLL

There is a bunch of other satisfiability-equivalent transformations. Here are some of them:

Pure literal. If $\ell \in F$, $\overline{\ell} \notin F$, then $F := F[\ell]$.

Subsumption. If $C \subseteq D$, then remove D.

Equivalent literals. If F contains $\overline{\ell} \vee \ell'$ and $\overline{\ell}' \vee \ell$, then eliminate ℓ' and $\overline{\ell}'$ by replacing them with ℓ and $\overline{\ell}$, resp.

Blocked clause. A literal ℓ in a clause C blocks it if $\forall D \in F$ $(\overline{\ell} \in D \Rightarrow |C \cap \overline{D}| \geqslant 2)$, that is, if $\overline{\ell} \in D$, then C and D must also have at least one other pair of contrary literals.

If a clause is blocked (by any literal), then we can remove C. (It's an easy exercise to show that we do not add add new satisfying assignments this way.)

The following simple notion plays a crucial role in **SAT** algorithms, solvers, proof systems. It's first-order counterpart plays a crucial role in (first-order) automated theorem proving.

Definition 2 (Resolvent). Given $x \vee C$ and $\overline{x} \vee D$ such that $C \cap \overline{D} = \emptyset$, their **resolvent** (by x) is $C \vee D$.

It is easy to see that the resolvent of two clauses is semantically replied by their conjunction, so it is a valid step of logical reasoning, and resolvents can be added to the formula without changing its satisfiability.

The Davis–Putnam procedure eliminates variables one by one using the following function

```
Function DP_x(F)                                    // simplifies F by getting rid of x
  • Add all resolvents[1] by x to F.
  • Remove all clauses containing x or x̄.
  • Return F.
```

The simplified formula $\mathrm{DP}_x(F)$ is satisfiable iff F is satisfiable. A satisfying assignment for F can also be reconstructed from a satisfying assignment for $\mathrm{DP}_x(F)$, if any.

Therefore, if we apply DP_x to each variable: $\mathrm{DP}_{x_1}(\mathrm{DP}_{x_2}(\ldots(\mathrm{F})))$, we get rid of all the variables, so we get a trivial formula `False` or `True`, and this is a correct answer about the satisfiability of F; a satisfying assignment can be reconstructed backwards step by step.

Remark 2. Note that DP may increase the number of clauses and that it can introduce 4-clauses when applied to a formulas in 3-CNF. However, there are situations where it does not bring too much harm: for example, if one of the clauses is a 2-clause, the resolvent is no longer than the second clause; if DP is applied to a variable that occurs at most four times, then the number of clauses does not grow, etc. However, even in such cases it can change other syntactic properties of the formula (for example, increase the number of occurrences of variables), so it must be carefully watched regarding its effect on the complexity. In particular, the Davis–Putnam procedure solves **SAT** in potentially exponential time because of the blow-up of the number of clauses.

1.6 Making DPLL practical

So far we talked about **3-SAT** only, but the bounds we proved can be easily generalized to ***k*-SAT**. Also the same tricks, of course, work for general **SAT**, even if they do not lead to theoretical worst-case upper bounds.

DPLL-type algorithms are not currently the best in theory, but they can be made very practical. In particular, contemporary SAT solvers that use clause learning (which is a technique for adding useful resolvents), called CDCL[2] solvers (coming in the next lectures), have DPLL as their base.

[1]In the definition of a resolvent we require that C and D do not have a contrary pair; indeed, if they do, the resolvent is useless, as it is trivially true.

[2]Conflict-Driven Clause Learning.

To make these algorithms even more practical, one needs to optimize the data structures. Classically, one represents a CNF using adjacency lists:

- A clause is a list of literals.
- For every variable, there is a list of clauses where it occurs.

These are easy to program and easy to verify structures, because at every moment the formula is described by its data structure very naturally. They are enough for theoretical results and for the use in "proof-of-concept" solvers. However, in practice DPLL and CDCL solvers need to update a lot of data after assigning a single variable, and it is not always worth doing it immediately, as maybe we will backtrack soon and never look at some parts of the formula. In "lazy" data structures certain updates are delayed until the data is used. We will discuss them when we discuss the solvers.

2 PPZ: Random permutations

We consider variables for splitting in a certain order, for example, we tend to take variables from the same clause. Is there a better order for a formula in k-CNF? Let us try a random order and see what happens!

2.1 The case of a single assignment

Observation 1. Consider a **uniquely satisfiable** formula, that is, it has a single satisfying assignment A. For each variable x, there **must** be a clause of pattern $\boxed{- \cdots - +}$ for x, that is, $C = \ell_1 \vee \ldots \vee \ell_{k-1} \vee \ell_k$, for $\ell_k \in \{x, \overline{x}\}$ and $A[\ell_1] = \ldots = A[\ell_{k-1}] =$ `False`, $A[\ell_k] =$ `True`). Indeed, otherwise if we change the value of x in A, we find one more satisfying assignment.

This clause C is called **critical** for x (w.r.t. A), and x is called a critical variable for C.

Then if we know the values of ℓ_1, ℓ_2, we can figure out the value of x, if it is to be determined later than those of ℓ_1, ℓ_2.

Note that a critical clause can be critical for a specific variable only: critical clauses for two different variables are always different.

The idea of the next algorithm is to choose the order of variables at random and hope that we have enough variables to be determined later than the other variables in their critical clauses.

Algorithm 3 (Paturi, Pudlák, Zane).

```
Initialize $A[1..n]$ by $*$.
Pick a random permutation $\pi \in S_n$.
For $i = 1, \ldots, n$
    If $A[x_{\pi(i)}] = *$ then
        $A[x_{\pi(i)}] := \mathrm{random}(\{0,1\})$                    (G)
        $F := F[x_{\pi(i)} \leftarrow A[x_{\pi(i)}]]$
        Perform unit propagation (updating $A$)
If $F =$ True then return $A$ else return "no"
```

The probability of a correct guess at step `(G)` is $1/2$.

Consider a formula in k-CNF. For a variable x, the chances that it is the last one (w.r.t. π) among the variables present in a clause C, is at least $1/k$. Let f be the number of values given for free, not guessed. Let $E := \boldsymbol{E} f \geqslant n \cdot 1/k = n/k$ (by the linearity of expectation). Thus, the number of such values $\geqslant n/k$ (so the number of random guesses $\leqslant n - n/k$) with probability $\geqslant 1/n$. Indeed, assuming w.l.o.g.[3] that $n \equiv 0 \pmod{k}$, if the desired probability is $< 1/n$, then

$$E \leqslant n \cdot \Pr\{f \geqslant n/k\} + (n/k - 1) \cdot \Pr\{f < n/k\} < n \cdot 1/n + (n/k - 1) \cdot 1 \leqslant n/k.$$

The overall success probability is thus $\geqslant 2^{-(n-n/k)} \cdot 1/n$, because random guesses are, of course, independent of the choice of permutation.

Recall that if a one-sided error algorithm has success probability $\geqslant p$, repeating it $1/p$ times fails with probability $\leqslant (1-p)^{1/p} < 1/\boldsymbol{e}$.

Thus repeating our algorithm $O(n \cdot 2^{n-n/k})$ times yields a constant probability of error, and repeating it $\widetilde{O}(2^{n-n/k})$ times can give us any exponentially small probability of error.

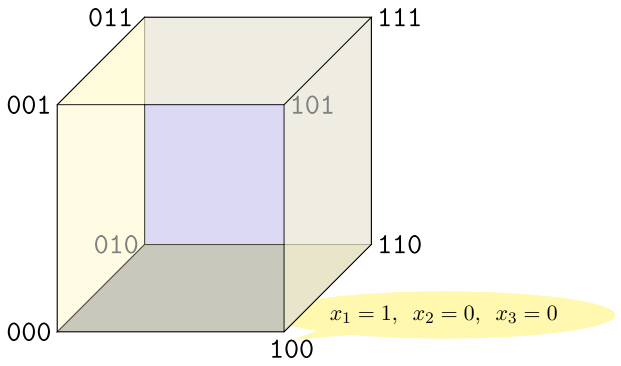
2.2 The general case

2.2.1 Boolean cube and Satisfiability Coding Lemma

It is handy to think about the 2^n Boolean assignments arranged in a graph that is the 1-dimensional "skeleton" of the n-dimensional cube $\{0,1\}^n$, where

- the assignments are the nodes,
- the edges connect the assignments that are at Hamming distance one from each other (that is, differ in the value of exactly one variable).

Note that the distance in this graph is exactly the Hamming distance (the number of values of variables that are different in two assignments).



So far we managed to find a satisfying assignment for a uniquely satisfiable formula. Let us consider the general case now. Intuitively, the more satisfying assignments we have, the easier the problem.

[3]We estimate the running time up to $\widetilde{O}$.

Definition 3 (neighbouring assignments and isolation). Let A^x be A with x's value flipped $(0 \leftrightarrow 1)$. An assignment A is **d-isolated** if there are d variables x s.t. $F[A^x] = 0$.

Note that A^x is a neighbour of A in the Boolean cube.

One can consider the PPZ procedure as decoding a "compressed" version of a satisfying assignment (just replace the guesses by the correct values: the number of these values is potentially much smaller than n). By performing the procedure we recover the values of the remaining values using the clause elimination procedure.

The more isolated is a satisfying assignment, the better compression we have. We formalize it in the following lemma.

Lemma 2 (Satisfiability Coding Lemma for ***k*-SAT**)**.** *If a satisfying assignment A is d-isolated, then the mathematical expectation of the number of correct values needed to recover it (the number of random guesses in the PPZ algorithm) needed to recover it is $\leqslant n - d/k$ (for a random permutation).*

Proof. If A is isolated in the direction of a variable x, then x has a critical clause C.

In total, we have d such variables, their critical clauses are different (because a critical clause is critical for a single variable only), so x is the last in C with prob. $1/|C| \geqslant 1/k$, and $\boldsymbol{E}\#$(such variables) $\geqslant d \cdot 1/k$, and thus $\boldsymbol{E}\#$(guesses) $\leqslant n - d \cdot 1/k$. □

We do not assume the divisibility by k any more, as d van be arbitrary, so let us do rounding. We claim that the number of guesses is at most $n - \lfloor d/k \rfloor$ with probability $\geqslant 1/n$. Indeed, let us again denote by f the number of values given for free. If the probability is not as claimed, then $\boldsymbol{E}f < n \cdot \Pr\{f \geqslant \lfloor d/k \rfloor\} + (\lfloor d/k \rfloor - 1) \cdot \Pr\{f < \lfloor d/k \rfloor\} < n \cdot 1/n + (\lfloor d/k \rfloor - 1) \cdot 1 = \lfloor d/k \rfloor$ contradicting the assumption.

We will use this lemma to show that the error probability of the PPZ algorithm stays the same in the general case as in the uniquely satisfiable case. We need, however, another technical lemma that allows us to show that, in a sense, there is a compromise between the isolation of the assignments and their number.

Define the isolation degree of an assignment A (w.r.t. formula F), $\delta(A) := \delta_F(A) := d$ if A is d-isolated in F.

Lemma 3 (Isolated satisfying assignments)**.** *Then* $\sum\limits_{A:F[A]=1} 2^{\delta(A)-n} \geqslant 1$, *where F contains n variables.*

Proof. Induction on the dimension of the Boolean cube (the number of variables). Let us prove that if the statement is true for the dimension $n-1$ (in particular, for $F[x \leftarrow 0]$ and $F[x \leftarrow 1]$), then it holds for F. Consider two cases:

1. If both formulas are **SAT**, then
$$\sum_{A:F[A]=1} 2^{\delta(A)-n} = \sum_{A \ni x:F[A]=1} 2^{\delta(A)-n} + \sum_{A \ni \overline{x}:F[A]=1} 2^{\delta(A)-n}$$
$$\geqslant \frac{1}{2} \cdot \sum_{B:F[x\leftarrow 1][B]=1} 2^{\delta_{\dots}(B)-n+1} + \frac{1}{2} \cdot \sum_{C:F[x\leftarrow 0][C]=1} 2^{\delta_{\dots}(C)-n+1} \overset{\boldsymbol{ind}}{\geqslant} \frac{1}{2} + \frac{1}{2} = 1.$$

2. If one of the formulas is **UNSAT**, then for the other one (call it G), $\delta_G(A\backslash\{x,\overline{x}\}) = \delta_F(A)-1$, so $\sum_{A:F[A]=1} 2^{\delta(A)-n} = \sum_{B:G[B]=1} 2^{\delta_G(B)-n+1} \overset{ind}{\geqslant} 1$. □

2.2.2 Putting it together

Recall the algorithm we are talking about:

Algorithm 4 (Paturi, Pudlák, Zane).

```
Initialize $A[1..n]$ by $*$.
Pick a random permutation $\pi \in S_n$.
For $i = 1,\ldots,n$
    If $A[x_{\pi(i)}] = *$ then
        $A[x_{\pi(i)}] := \mathrm{random}(\{0,1\})$                    (G)
        $F := F[x_{\pi(i)} \leftarrow A[x_{\pi(i)}]]$
        Perform unit propagation (updating $A$)
```

Let $S :=$ {set of all satisfying assignments for F}.

$$\Pr_{A\in S}\{\text{success}\} \overset{\text{non-intersecting}}{=} \sum_{A\in S} \Pr\{A \text{ is output}\} \geqslant \sum_{A\in S} \frac{1}{n}\cdot 2^{\lfloor \delta(A)/k\rfloor - n} \geqslant$$
$$\geqslant \sum_{A\in S} \frac{1}{n}\cdot 2^{\delta(A)/k-1-n} = \sum_{A\in S} \frac{1}{2n}\cdot 2^{\delta(A)/k-n} =$$
$$= \frac{1}{2n}\cdot 2^{n/k-n}\cdot \sum_{A\in S} 2^{(\delta(A)-n)/k} \geqslant (\text{since } \sqrt[k]{\varepsilon} \geqslant \varepsilon \text{ for } 1 \geqslant \varepsilon \geqslant 0)$$
$$\geqslant \frac{1}{2n}\cdot 2^{n/k-n}\cdot \sum_{A\in S} 2^{\delta(A)-n} \geqslant (\text{by Lemma 3})$$
$$\geqslant \frac{1}{2n}\cdot 2^{n/k-n}. \quad \text{Time } \widetilde{O}(2^{n-n/k}) \text{ for } \boldsymbol{k}\textbf{-SAT},\ \widetilde{O}(2^{2n/3}) \text{ for } \textbf{3-SAT}.$$

Remark 3. The running time of this algorithm can be improved using the following extension:

Algorithm 5 (Paturi, Pudlák, Saks, Zane).

```
- Add all resolvents of size at most $o(\log n)$.
- Execute PPZ.
```

We will not prove the improved upper bound.

Historical notes and further reading

These historical remarks are related both to this lecture and to the next one.

The first less-than-2^n bounds for ***k*-SAT** were published in 1979–1985 independently by Evgeny Dantsin in USSR and by Burkhard Monien and Ewald Speckenmeyer in Germany. The branching numbers technique was suggested in mid-1990s by Oliver Kullmann and Horst Luckhardt. Among other things, they present simple rules to compare branching numbers without computing them. Also in late-1980s and 1990s there was a series of DPLL-type algorithms with more and more intricate case analysis for **3-SAT** culminated in particular in Kulmann's $O(1.497^n)$-time upper bound. Ramamohan Paturi, Pavel Pudlák, and Francis Zane suggested their random permutations approach in late 1990s, and their randomization approach turned out to be much easier for the analysis than DPLL; the same authors plus Michael Saks improved it further. Local search algorithms that employ either random or deterministic walks in the Boolean cube have been exploited in an experimental manner by many authors in the 1990s: Bram Cohen, Henry Kautz, Hector Levesque, David Mitchell, Bart Selman, to mention a few. The simple lower bound for them mentioned in the lecture is due to myself. The random walk approach for **2-SAT** was suggested by Christos Papadimitriou. The random walk algorithm for **3-SAT** and (k, d)-**CSP** in general (described in the next lecture) was suggested by Uwe Schöning at the very end of the century. Then it was derandomized by three independent groups (Evgeny Dantsin + Edward A. Hirsch, Andreas Goerdt + Uwe Schöning, Ravi Kannan + Jon Kleinberg + Christos Papadimitriou + Prabhakar Raghavan) and the summary of the somewhat similar approaches was published together. The clause shortening approach to general **SAT** was suggested by Rainer Schuler in 2000s and then derandomized by several authors; the last reduction from **SAT** to ***k*-SAT** was published in a paper by Chris Calabro, Russell Impagliazzo and Ramamohan Paturi in 2006.

The respective references can be found in SAT Handbook (*Handbook of Satisfiability*, edited by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh; published by IOS Press), see the chapter "Worst-Case Upper Bounds" and the chapter "Incomplete Algorithms".

However, the subject is covered sufficiently in these lecture notes and there is no need to look into the Handbook or check the references therein.

Later the approaches of random permutations and random walks were generalized in a series of results combining the two approaches and leading to even better time bounds. Also better derandomization results appeared. The field is still developing, so it is difficult to point to the latest results in every direction.

The results concerning weakly exponential running time upper bounds have stimulated a new field of *Parameterized Complexity (Fixed-Parameter Tractability)*, which became very popular in the new century. The Handbook also contains a chapter on the basics of this field, but it is out of the scope for this lecture course.