Chapter preprint of the second edition of the Handbook of Satisfiability (2021)
Armin Biere, Marijn Heule, Hans van Maaren and Toby Walsh (Eds.)
Published by IOS Press. The print edition can be ordered at
https://www.iospress.nl/book/handbook-of-satisfiability-2/
The published electronic chapters (Version of Record) are available at
https://ebooks.iospress.nl/ISBN/978-1-64368-161-0/

Chapter 16

Worst-Case Upper Bounds

Evgeny Dantsin and Edward A. Hirsch

There are many algorithms for testing satisfiability — how to evaluate and compare them? It is common (but still disputable) to identify the efficiency of an algorithm with its worst-case complexity. From this point of view, asymptotic upper bounds on the worst-case running time and space is a criterion for evaluation and comparison of algorithms. In this chapter we survey ideas and techniques behind satisfiability algorithms with the currently best upper bounds. We also discuss some related questions: "easy" and "hard" cases of SAT, reducibility between various restricted cases of SAT, the possibility of solving SAT in subexponential time, etc.

In Section 16.1 we define terminology and notation used throughout the chapter. Section 16.2 addresses the question of which special cases of SAT are polynomial-time tractable and which ones remain **NP**-complete. The first nontrivial upper bounds for testing satisfiability were obtained for algorithms that solve k-SAT; such algorithms also form the core of general SAT algorithms. Section 16.3 surveys the currently fastest algorithms for k-SAT. Section 16.4 shows how to use bounds for k-SAT to obtain the currently best bounds for SAT. Section 16.5 addresses structural questions like "what else happens if k-SAT is solvable in time $\langle\ldots\rangle$?". Finally, Section 16.6 summarizes the currently best bounds for the main cases of the satisfiability problem.

16.1. Preliminaries

16.1.1. Definitions and notation

A *literal* is a Boolean variable or its negation. A *clause* is a finite set of literals that does not contain a variable together with its negation. By a *formula* we mean a Boolean formula in conjunctive normal form (*CNF formula*) defined as a finite set of clauses. The number of literals in a clause is called the *length* of the clause. A formula F is called a *k-CNF* formula if every clause in F has length at most k.

Throughout the chapter we write n, m, and l to denote the following natural parameters of a formula F:

- n is the number of variables occurring in F;
- m is the number of clauses in F;
- l is the total number of occurrences of all variables in F.

We also write $|F|$ to denote the length of a reasonable binary representation of F, i.e., the size of the input in the usual complexity-theoretic sense. The ratio m/n is called the *clause density* of F.

An *assignment* is a mapping from a set of variables to $\{\mathsf{true}, \mathsf{false}\}$. We identify an assignment with a set A of literals: if a variable x is mapped to true then $x \in A$; if x is mapped to false then $\neg x \in A$. Given a formula F and an assignment A, we write $F[A]$ to denote the result of substitution of the truth values assigned by A. Namely, $F[A]$ is a formula obtained from F as follows:

- all clauses that contain literals belonging to A are removed from F;
- all literals that are complementary to the literals in A are deleted from the remaining clauses.

For example, for $F = \{\{x, y\}, \{\neg y, z\}\}$ and $A = \{y\}$, the formula $F[A]$ is $\{z\}$. If C is a clause, we write $F[\neg C]$ to denote $F[A]$ where A consists of the literals complementary to the literals in C.

An assignment A *satisfies* a formula F if $F[A] = \emptyset$. If a formula has a satisfying assignment, it is called *satisfiable*.

SAT denotes the language of all satisfiable CNF formulas (we sometimes refer to this language as *General SAT* to distinguish it from its restricted versions), *k-SAT* is its subset consisting of k-CNF formulas, *SAT-f* is the subset of SAT containing formulas with each variable occurring at most f times, *k-SAT-f* is the intersection of the last two languages. *Unique k-SAT* denotes a promise problem: an algorithm solves the problem if it gives correct answers for k-CNF formulas that have at most one satisfying assignment (and there are no requirements on the behavior of the algorithm on other formulas).

Complexity bounds are given using the standard asymptotic notation (O, Ω, o, etc), see for example [CLRS01]. We use log to denote the binary logarithm and ln to denote the natural logarithm. The binary entropy function is given by

$$H(x) = -x \log x - (1 - x) \log(1 - x).$$

16.1.2. Transformation rules

We describe several simple operations that transform formulas without changing their satisfiability status (we delay more specific operations until they are needed).

Unit clauses. A clause is called *unit* if its length is 1. Clearly, a unit clause determines the truth value of its variable in a satisfying assignment (if any). Therefore, all unit clauses can be eliminated by the corresponding substitutions. This procedure (iterated until no unit clauses are left) is known as *unit clause elimination*.

Subsumption. Another simple procedure is *subsumption*: if a formula contains two clauses and one is a subset of the other, then the larger one can be dropped.

Resolution. If two clauses A and B have exactly one pair of complementary literals $a \in A$ and $\neg a \in B$, then the clause $A \cup B \setminus \{a, \neg a\}$ is called the *resolvent* of A and B (by a) and denoted by $R(A, B)$. The resolvent is a logical consequence of these clauses and can be added to the formula without changing its satisfiability. The important cases of adding resolvents are:

- *Resolution with subsumption.* If F contains two clauses C and D such that their resolvent $R(C, D)$ is a subset of D, replace F by $(F \setminus \{D\}) \cup \{R(C, D)\}$.
- *Elimination of a variable by resolution.* Given a formula F and a literal a, the formula denoted $\mathrm{DP}_a(F)$ is constructed from F by adding all resolvents by a and then removing all clauses that contain a or $\neg a$ (this transformation was used in [DP60]). A typical use of this rule is as follows: if the number of clauses (resp. the number of literal occurrences) in $\mathrm{DP}_a(F)$ is smaller than in F, replace F by $\mathrm{DP}_a(F)$.
- *Bounded resolution.* Add all resolvents of size bounded by some function of the formula parameters.

16.2. Tractable and intractable classes

Consider the satisfiability problem for a certain class of formulas. Depending on the class, either this restriction is in **P** (2-SAT for example) or no polynomial-time algorithm for the restricted problem is known (3-SAT). Are there any criteria that would allow us to distinguish between tractable and intractable classes? A remarkable result in this direction was obtained by Schaefer in 1978 [Sch78]. He considered a generalized version of the satisfiability problem, namely he considered Boolean constraint satisfaction problems $\mathrm{SAT}(\mathcal{C})$ where $\mathcal{C}$ is a set of constraints, see precise definitions below. Each set $\mathcal{C}$ determines a class of formulas such as 2-CNF, 3-CNF, Horn formulas, etc. Loosely speaking, Schaefer's dichotomy theorem states that there are only two possible cases: $\mathrm{SAT}(\mathcal{C})$ is either in **P** or **NP**-complete. The problem is in **P** if and only if the class determined by $\mathcal{C}$ has at least one of the following properties:

1. Every formula in the class is satisfied by assigning true to all variables;
2. Every formula in the class is satisfied by assigning false to all variables;
3. Every formula in the class is a Horn formula;
4. Every formula in the class is a dual-Horn formula;
5. Every formula in the class is in 2-CNF;
6. Every formula in the class is affine.

We describe classes with these properties in the next section. Schaefer's theorem and some relevant results are given in Section 16.2.2.

16.2.1. "Easy" classes

Trivially satisfiable formulas. A CNF formula F is satisfied by assigning true (resp. false) to all variables if and only if every clause in F has at least one positive (resp. negative) literal. The satisfiability problem for such formulas is trivial: all formulas are satisfiable.

Horn and dual-Horn formulas. A CNF formula is called *Horn* if every clause in this formula has at most one positive literal. Satisfiability of a Horn formula F can be tested as follows. If F has unit clauses then we apply unit clause elimination until all unit clauses are eliminated. If the resulting formula F' contains the empty clause, we conclude that F is unsatisfiable. Otherwise, every clause in F' contains at least two literals and at least one of them is negative. Therefore, F' is trivially satisfiable by assigning false to all variables, which means that F is satisfiable too. It is obvious that this method takes polynomial time. Using data flow techniques, satisfiability of Horn formulas can be tested in linear time [DG84].

A CNF formula is called *dual-Horn* if every clause in it has at most one negative literal. The satisfiability problem for dual-Horn formulas can be solved similarly to the case of Horn formulas.

2-CNF formulas. A linear-time algorithm for 2-SAT was given in [APT79]. This algorithm represents an input formula F by a labeled directed graph G as follows. The set of vertices of G consists of $2n$ vertices corresponding to all variables occurring in F and their negations. Each clause $a \vee b$ in F can be viewed as two implications $\neg a \to b$ and $\neg b \to a$. These two implications produce two edges in the graph: $(\neg a, b)$ and $(\neg b, a)$. It is easy to prove that F is unsatisfiable if and only if there is a variable x such that G has a cycle containing both x and $\neg x$. Since strongly connected components can be computed in linear time, 2-SAT can be solved in linear time too.

There are also other methods of solving 2-SAT in polynomial time, for example, random walks [Pap94] or resolution (note that the resolution rule applied to 2-clauses produces a 2-clause, so the number of all possible resolvents does not exceed $O(n^2)$).

Affine formulas. A *linear equation over the two-element field* is an expression of the form $x_1 \oplus \ldots \oplus x_k = \delta$ where $\oplus$ denotes the sum modulo 2 and δ stands for 0 or 1. Such an equation can be expressed as a CNF formula consisting of 2^{k-1} clauses of length k. An *affine formula* is a conjunction of linear equations over the two-element field. Using Gaussian elimination, we can test satisfiability of affine formulas in polynomial time.

16.2.2. Schaefer's dichotomy theorem

Definition 16.2.1. A function $\phi\colon \{\mathsf{true}, \mathsf{false}\}^k \to \{\mathsf{true}, \mathsf{false}\}$ is called a *Boolean constraint* of arity k.

Definition 16.2.2. Let ϕ be a constraint of arity k and $x_1, \ldots, x_k$ be a sequence of Boolean variables (possibly with repetitions). The pair $\langle \phi, (x_1, \ldots, x_k) \rangle$ is called a *constraint application*. Such a pair is typically written as $\phi(x_1, \ldots, x_k)$.

Definition 16.2.3. Let $\phi(x_1, \ldots, x_k)$ be a constraint application and A be a truth assignment to $x_1, \ldots, x_k$. We say that A *satisfies* $\phi(x_1, \ldots, x_k)$ if ϕ evaluates to true on the truth values assigned by A.

Definition 16.2.4. Let $\Phi = \{C_1, \ldots, C_m\}$ be a set of constraint applications and A be a truth assignment to all variables occurring in Φ. We say that A is a *satisfying assignment* for Φ if A satisfies every constraint application in Φ.

Definition 16.2.5. Let $\mathcal{C}$ be a set of Boolean constraints. We define SAT($\mathcal{C}$) to be the following decision problem: given a finite set Φ of applications of constraints from $\mathcal{C}$, is there a satisfying assignment for Φ?

Example. Let $\mathcal{C}$ consist of four Boolean constraints $\phi_1, \phi_2, \phi_3, \phi_4$ defined as follows:

$$\begin{aligned}\phi_1(x, y) &= x \vee y;\\ \phi_2(x, y) &= x \vee \neg y;\\ \phi_3(x, y) &= \neg x \vee \neg y;\\ \phi_4(x, y) &= \mathsf{false}.\end{aligned}$$

Then SAT($\mathcal{C}$) is 2-SAT.

Theorem 16.2.1 (Schaefer's dichotomy theorem [Sch78])**.** *Let $\mathcal{C}$ be a set of Boolean constraints. If $\mathcal{C}$ satisfies at least one of the conditions (1)–(6) below then SAT($\mathcal{C}$) is in* **P**. *Otherwise SAT($\mathcal{C}$) is* **NP***-complete.*

1. *Every constraint in $\mathcal{C}$ evaluates to* true *if all arguments are* true.
2. *Every constraint in $\mathcal{C}$ evaluates to* true *if all arguments are* false.
3. *Every constraint in $\mathcal{C}$ can be expressed as a Horn formula.*
4. *Every constraint in $\mathcal{C}$ can be expressed as a dual-Horn formula.*
5. *Every constraint in $\mathcal{C}$ can be expressed as a 2-CNF formula.*
6. *Every constraint in $\mathcal{C}$ can be expressed as an affine formula.*

The main part of this theorem is to prove that SAT($\mathcal{C}$) is **NP**-hard if none of the conditions (1)–(6) holds for $\mathcal{C}$. The proof is based on a classification of the sets of constraints that are closed under certain operations (conjunction, substitution of constants, existential quantification). Any such set either satisfies one of the conditions (3)–(6) or coincides with the set of all constraints.

Schaefer's theorem was extended and refined in many directions, for example a complexity classification of the polynomial-time solvable case (with respect to $\mathbf{AC}^0$ reducibility) is given in [ABI+05]. Likewise, classification theorems similar to Schaefer's one were proved for many variants of SAT, including the counting satisfiability problem #SAT, the quantified satisfiability problem QSAT, the maximum satisfiability problem MAX SAT (see Part 4 of this book for the definitions of these extensions of SAT), and others, see a survey in [CKS01].

A natural question arises: given a set $\mathcal{C}$ of constraints, how difficult is it to recognize whether SAT($\mathcal{C}$) is "easy" or "hard"? That is, we are interested in the complexity of the following "meta-problem": Given $\mathcal{C}$, is SAT($\mathcal{C}$) in **P**? As shown in [CKS01], the complexity of the meta-problem depends on how the constraints are specified:

- If each constraint in $\mathcal{C}$ is specified by its set of satisfying assignments then the meta-problem is in **P**.
- If each constraint in $\mathcal{C}$ is specified by a CNF formula then the meta-problem is **co-NP**-hard.

- If each constraint in $\mathcal{C}$ is specified by a DNF formula then the meta-problem is **co-NP**-hard.

16.3. Upper bounds for k-SAT

SAT can be trivially solved in 2^n polynomial-time steps by trying all possible assignments. The design of better exponential-time algorithms started with papers [MS79, Dan81, Luc84, MS85] exploiting a simple divide-and-conquer approach[1] for k-SAT. Indeed, if a formula contains a 3-clause $\{x_1, x_2, x_3\}$, then deciding its satisfiability is equivalent to deciding the satisfiability of three simpler formulas $F[x_1]$, $F[\neg x_1, x_2]$, and $F[\neg x_1, \neg x_2, x_3]$, and the recurrent inequality on the number of leaves in such a computation tree already gives a $|F|^{O(1)} \cdot 1.84^n$-time algorithm for 3-SAT, see Part 1, Chapter 8 for a survey of techniques for estimating the running time of such algorithms.

Later development brought more complicated algorithms based on this approach both for k-SAT [Kul99] and SAT [KL97, Hir00], see also Section 16.4.2. However, modern randomized algorithms and their derandomized versions appeared to give better bounds for k-SAT. In what follows we survey these new approaches: *critical clauses* (Section 16.3.1), *multistart random walks* (Section 16.3.2), and *cube covering* proposed in an attempt to derandomize the random-walk technique (Section 16.3.3). Finally, we go briefly through recent improvements for particular cases (Section 16.3.4).

16.3.1. Critical clauses

To give the idea of the next algorithm, let us switch for a moment to a particular case of k-SAT where a formula has at most one satisfying assignment (Unique k-SAT). The restriction of this case ensures that every variable v must occur in a *v-critical* clause that is satisfied only by the literal corresponding to v (otherwise one could change its value to obtain a different satisfying assignment); we call v the *principal* variable of this clause. This observation remains also valid for general k-SAT if we consider an *isolated* satisfying assignment that becomes unsatisfying if we flip the value of exactly one variable. More generally, we call a satisfying assignment *j-isolated* if this happens for exactly j variables (while for any other of the remaining $n - j$ variables, a flip of its value yields another satisfying assignment). Clearly, each of these j variables has its critical clause, i.e., a clause where it is the principal variable.

Now start picking variables one by one at random and assigning random values to them eliminating unit clauses after each step. Clearly, each critical clause of length k has a chance of $1/k$ of being "properly" ordered: its principal variable is chosen after all other variables in the clause and thus will be assigned "for free" during the unit clause elimination provided the values of other variables are chosen correctly. Thus the expected number of values we get "for free" is j/k; the probability to guess correctly the values of other variables is $2^{-(n-\lfloor j/k \rfloor)}$.

[1] Such SAT algorithms are usually called *DPLL algorithms* since the approach originates from [DP60, DLL62].

While this probability of success is the largest for 1-isolated satisfying assignments, j-isolated assignments for larger j come in larger "bunches". The overall probability of success is

$$\sum_j 2^{-(n-\lfloor j/k \rfloor)} \cdot (\text{number of } j\text{-isolated assignments}),$$

which can be shown to be $\Omega(2^{-n(1-1/k)})$ [PPZ97]. The probability that the number of "complimentary" values is close to its expectation adds an inverse polynomial factor. Repeating the procedure in order to get a constant probability of error yields an $|F|^{O(1)} \cdot 2^{n(1-1/k)}$-time algorithm, which can be derandomized for some penalty in time.

The above approach based on critical clauses was proposed in [PPZ97]. A follow-up [PPSZ98] (journal version: [PPSZ05]) gives an improvement of the $2^{n(1-1/k)}$ bound by using a preprocessing step that augments the formula by all resolvents of size bounded by a slowly growing function. Adding the resolvents formalizes the intuition that a critical clause may become "properly ordered" not only in the original formula, but also during the process of substituting the values.

Algorithm 16.3.1 (The PPSZ algorithm, [PPSZ05])**.**

`Input:` k-CNF formula F.
`Output:` `yes`, if F is satisfiable, `no` otherwise.[2]

1. Augment F using bounded resolution up to the length bounded by any slowly growing function such that $s(n) = o(\log n)$ and s tends to infinity as n tends to infinity.
2. Repeat $2^{n\left(1-\frac{\mu_k}{k-1}+o(1)\right)}$ times, where $\mu_k = \sum_{j=1}^{\infty} \frac{1}{j\left(j+\frac{1}{k-1}\right)}$:
   (a) Apply unit clause elimination to F.
   (b) Pick randomly a literal a still occurring in F and replace F by $F[a]$.[3]

Theorem 16.3.1 ([PPSZ05])**.** *The PPSZ algorithm solves k-SAT in time*

$$|F|^{O(1)} \cdot 2^{n\left(1-\frac{\mu_k}{k-1}+o(1)\right)}$$

with error probability $o(1)$, where μ_k is an increasing sequence with $\mu_3 \approx 1.227$ and $\lim_{k\to\infty} \mu_k = \pi^2/6$.

16.3.2. Multistart random walks

A typical local-search algorithm for SAT starts with an initial assignment and modifies it step by step, trying to come closer to a satisfying assignment (if any). Methods of modifying may vary. For example, greedy algorithms choose a variable and flip its value so as to increase some function of the assignment, say,

[2] This algorithm and the other randomized algorithms described below may incorrectly say "no" on satisfiable formulas.

[3] Contrary to [PPSZ05], we choose a literal not out of all $2n$ possibilities (for n variables still remaining in the formula) but from a possibly smaller set. However, this only excludes a chance to assign the wrong value to a pure literal.

the number of satisfied clauses. Another method was used in Papadimitriou's algorithm for 2-SAT [Pap91]: flip the value of a variable chosen at random from an unsatisfied clause. Algorithms based on this method are referred to as *random-walk* algorithms.

Algorithm 16.3.2 (The multistart random-walk algorithm)**.**

`Input:` k-CNF formula F.
`Output:` satisfying assignment A if F is satisfiable, `no` otherwise.
`Parameters:` integers t and w.

1. Repeat t times:
   (a) Choose an assignment A uniformly at random.
   (b) If A satisfies F, return A and halt. Otherwise repeat the following instructions w times:
   - Pick any unsatisfied clause C in F.
   - Choose a variable x uniformly at random from the variables occurring in C.
   - Modify A by flipping the value of x in A.
   - If the updated assignment A satisfies F, return A and halt.
2. Return `no` and halt.

This multistart random-walk algorithm with $t = 1$ and $w = 2n^2$ is Papadimitriou's polynomial-time algorithm for 2-SAT [Pap91]. The algorithm with $t = (2 - 2/k)^n$ and $w = 3n$ is known as Schöning's algorithm for k-SAT where $k \geq 3$ [Sch99].

Random-walk algorithms can be analyzed using one-dimensional random walks. Consider one random walk. Suppose that the input formula F has a satisfying assignment S that differs from the current assignment A in the values of exactly j variables. At each step of the walk, A becomes closer to S with probability at least $1/k$ because any unsatisfied clause contains at least one variable whose values in A and in S are different. Thus, the performance of the algorithm can be described in terms of a particle walking on the integer interval $[0..n]$. The position 0 corresponds to the satisfying assignment S. At each step, if the particle's position is j, where $0 < j < n$, it moves to $j - 1$ with probability at least $1/k$ and moves to $j + 1$ with probability at most $1 - 1/k$.

Let p be the probability that a random walk that starts from a random assignment finds a fixed satisfying assignment S in at most $3n$ steps. Using the "Ballot theorem", Schöning shows in [Sch99] that $p \geq (2 - 2/k)^{-n}$ up to a polynomial factor. An alternative analysis by Welzl reduces the polynomial factor to a constant: $p \geq (2/3) \cdot (2 - 2/k)^{-n}$, see Appendix to [Sch02]. Therefore, Schöning's algorithm solves k-SAT with the following upper bound:

Theorem 16.3.2 ([Sch02])**.** *Schöning's algorithm solves k-SAT in time*

$$|F|^{O(1)} \cdot (2 - 2/k)^n$$

with error probability $o(1)$.

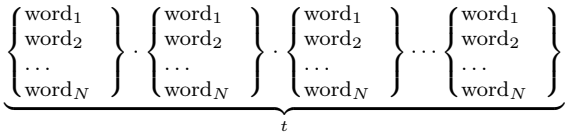

Figure 16.1. A block code $\mathcal{C}^t$ for code $\mathcal{C} = \{\text{word}_1, \ldots, \text{word}_N\}$.

16.3.3. Cube covering

Schöning's algorithm solves k-SAT using an exponential number of relatively "short" random walks that start from random assignments. Is it possible to derandomize this approach? A derandomized version of multistart random walks is given in [DGH+02].

The idea behind the derandomization can be described in terms of a covering of the Boolean cube with Hamming balls. We can view truth assignments to n variables as points in the Boolean cube $\{0,1\}^n$. For any two points $u, v \in \{0,1\}^n$, the *Hamming distance* between u and v is denoted by $\delta(u,v)$ and defined to be the number of coordinates in which these points are different. The *Hamming ball* of radius R with center w is defined to be

$$\{u \in \{0,1\}^n \mid \delta(u,w) \leq R\}.$$

Let $\mathcal{C}$ be a collection of Hamming balls of radius R with centers $w_1, \ldots, w_t$. This collection is called a *covering* of the cube if each point in the cube belongs to at least one ball from $\mathcal{C}$. When thinking of the centers $w_1, \ldots, w_t$ as *codewords*, we identify the covering $\mathcal{C}$ with a *covering code* [CHLL97].

The derandomization of Schöning's algorithm for k-SAT is based on the following approach. First, we cover the Boolean cube with Hamming balls of some fixed radius R (two methods of covering are described below). This covering should contain as few balls as possible. Then we consider every ball and search for a satisfying assignment inside it (see the procedure *Search* below). The time of searching inside a ball depends on the radius R: the smaller R we use, the less time is needed. However, if the radius is small, many balls are required to cover the cube. It is shown in [DGH+02] that for the particular procedure *Search* the overall runnning time is minimum when $R = n/(k+1)$.

Covering of the Boolean cube with Hamming balls. How many Hamming balls of a given radius R are required to cover the cube? Since the volume of a Hamming ball of radius R is at most $2^{nH(R/n)}$ where H is the binary entropy function, the smallest possible number of balls is at least $2^{n(1-H(R/n))}$. This trivial lower bound is known as *sphere covering bound* [CHLL97]. Our task is to construct a covering code whose size is close to the sphere covering bound. To do it, we partition n bits into d blocks with n/d bits in each block. Then we construct a covering code of radius R/d for each block and, finally, we define a covering code for the n bits to be the direct sum of the covering codes for blocks (see Fig. 16.1). More exactly, the cube-covering-based algorithm in [DGH+02] uses the following two methods of covering.

- *Method A: Constant number of blocks of linear size.* We partition n bits into d blocks where d is a constant. Then we use a greedy algorithm for the Set Cover problem to construct a covering code for a block with n/d bits. The resulting covering code for n bits is optimal: its size achieves the sphere covering bound up to a polynomial factor. However, this method has a disadvantage: construction of a covering code for a block requires exponential space.
- *Method B: Linear number of blocks of constant size.* We partition n bits into n/b blocks of size b each, where b is a constant. A covering code for each block is constructed using a brute-force method. The resulting covering code is constructed within polynomial space, however this code is only "almost" optimal: its size achieves the sphere covering bound up to a factor of $2^{\varepsilon n}$ for arbitrary $\varepsilon > 0$.

Search inside a ball. After constructing a covering of the cube with balls, we search for a satisfying assignment inside every ball. To do this, we build a tree using the local search paradigm: if the current assignment A does not satisfy the input formula, choose any unsatisfied clause $\{l_1, \ldots, l_k\}$ and consider assignments $A_1, \ldots, A_k$ where A_i is obtained from A by flipping the literal l_i (note that while a traditional randomized local search considers just *one* of these assignments, a deterministic procedure must consider *all* these assignments). More exactly, we use the following recursive procedure $Search(F, A, R)$. It takes as input a k-CNF formula F, an assignment A (center of the ball), and a number R (radius of the ball). If F has a satisfying assignment inside the ball, the procedure outputs yes (for simplicity, we describe a decision version); otherwise, the answer is no.

1. If F is true under A, return yes.
2. If $R \leq 0$, return no.
3. If F contains the empty clause, return no.
4. Choose any clause that is false under A. For each literal l in C, run $Search(F[l], A, R-1)$. Return yes if at least one of these calls returns yes. Otherwise return no.

The procedure builds a tree in which

- each degree is at most k;
- the height is at most R (the height is the distance from the center of the ball).

Therefore, $Search(F, A, R)$ runs in time k^R up to a polynomial factor.

Algorithm 16.3.3 (The cube-covering-based algorithm, [DGH+02])**.**

Input: k-CNF formula F.
Output: yes if F is satisfiable, no otherwise.
Parameter: $\varepsilon > 0$ (the parameter is needed only if Method B is used).

1. Cover the Boolean cube with Hamming balls of radius $R = n/(k+1)$ using either Method A or Method B.
2. For each ball, run $Search(F, A, R)$ where A is the center of the ball. Return yes if at least one of these calls returns yes. Otherwise, return false.

Theorem 16.3.3 ([DGH+02]). *The cube-covering-based algorithm solves k-SAT in time*

$$|F|^{O(1)} \cdot (2 - 2/(k+1))^n$$

and within exponential space if Method A is used, or in time

$$|F|^{O(1)} \cdot (2 - 2/(k+1) + \varepsilon)^n$$

and within polynomial space if Method B is used.

In the next section we refer to a weakened version of these bounds:

Proposition 16.3.4. *The cube-covering-based algorithm solves k-SAT in time*

$$|F|^{O(1)} \cdot 2^{n(1-1/k)}.$$

16.3.4. Improvements for restricted versions of *k*-SAT

3-SAT. Like the general case of k-SAT, the currently best bounds for randomized 3-SAT algorithms are better than those for deterministic 3-SAT algorithms.

- *Randomized algorithms.* Schöning's algorithm solves 3-SAT formulas in time $O(1.334^n)$. It was shown in [IT04] that this bound can be improved by combining Schöning's algorithm with the PPSZ algorithm. The currently best bound obtained using such a combination is $O(1.323^n)$ [Rol06].
- *Deterministic algorithms.* The cube-covering-based algorithm solves 3-SAT in time $O(1.5^n)$. This bound can be improved using a pruning technique for search inside a ball [DGH+02]. The currently best bound based on this method is $O(1.473^n)$ [BK04].

Unique *k*-SAT. The currently best bounds for Unique k-SAT are obtained using the approach based on critical clauses.

- *Randomized algorithms.* For $k > 4$, the best known bound for Unique k-SAT is the bound given by the PPSZ algorithm (Theorem 16.3.1). For Unique 3-SAT and Unique 4-SAT, the PPSZ bounds can be improved: $O(1.308^n)$ and $O(1.470^n)$ respectively [PPSZ05].
- *Deterministic algorithms.* The PPSZ algorithms can be derandomized for Unique k-SAT using the technique of limited independence [Rol05]. This derandomization yields bounds that can be made arbitrarily close to the bounds for the randomized case (see above) by taking appropriate values for the parameters of the derandomized version.

16.4. Upper bounds for General SAT

Algorithms with the currently best upper bounds for General SAT are based on the following two approaches:

- Clause shortening (Section 16.4.1). This method gives the currently best bound of the form $|F|^{O(1)} \cdot 2^{\alpha n}$ where α depends on n and m.
- Branching (Section 16.4.2, see also Part 1, Chapter 8). Algorithms based on branching have the currently best upper bounds of the form $|F|^{O(1)} \cdot 2^{\beta m}$ and $|F|^{O(1)} \cdot 2^{\gamma l}$, where β and γ are constants.

16.4.1. Algorithms based on clause shortening

The first nontrivial upper bound for SAT was given by Pudlak in [Pud98]. His randomized algorithm used the critical clauses method [PPZ97] to solve SAT with the $|F|^{O(1)} \cdot 2^{\alpha n}$ bound where $\alpha = 1 - 1/(2\sqrt{n})$. A slightly worse bound was obtained using a deterministic algorithm based on the cube-covering technique [DHW04].

Further better bounds were obtained using the *clause-shortening* approach proposed by Schuler in [Sch05]. This approach is based on the following dichotomy: *For any "long" clause (longer than some $k = k(n, m)$), either we can shorten this clause by choosing any k literals in the clause and dropping the other literals, or we can substitute* false *for these k literals in the entire formula.*

In other words, if a formula F contains a "long" clause C then F is equivalent to $F' \vee F''$ where F' is obtained from F by shortening C to a subclause D of length k and F'' is obtained from F by assigning false to all literals in D. This dichotomy is used to shorten all "long" clauses in the input formula F and apply a k-SAT algorithm afterwards.

The clause-shortening approach was described in [Sch05] as a randomized algorithm. Its derandomized version was given in [DHW06].

Algorithm 16.4.1 (The clause-shortening algorithm)**.**

Input: CNF formula F consisting of clauses $C_1, \ldots, C_m$.
Output: yes if F is satisfiable, no otherwise.
Parameter: integer k.

1. Change each clause C_i to a clause D_i as follows: If $|C_i| > k$ then choose any k literals in C_i and drop the other literals; otherwise leave C_i as is. Let F' denote the resulting formula $D_1 \vee \ldots \vee D_m$.
2. Test satisfiability of F' using the cube-covering-based algorithm (Section 16.3).
3. If F' is satisfiable, output yes and halt. Otherwise, for each i, do the following:
   (a) Convert F to F_i as follows:
      i. Replace C_j by D_j for all $j < i$;
      ii. Assign false to all literals in D_i.
   (b) Recursively invoke this algorithm on F_i.
4. Return no.

What value of k is optimal? If k is small then each k-SAT instance can be solved fast, however we have a large number of k-SAT instances. If k is large then the k-SAT subroutine takes much time. Schuler's bound and further better bounds for the clause-shortening approach were improved by Calabro, Impagliazzo, and Paturi in [CIP06]. Using this approach, they reduce a SAT instance to an exponential number of k-SAT instances and prove the following "conditional" bound:

Theorem 16.4.1 ([CIP06]). *If k-SAT can be solved in time $|F|^{O(1)} \cdot 2^{\alpha n}$ then SAT can be solved in time*

$$|F|^{O(1)} \cdot 2^{\alpha n + \frac{4m}{2^{\alpha k}}}$$

for any n, m and k such that $m \geq n/k$.

Although this theorem is not used in [CIP06] to give an improved bound for SAT explicitly, such a bound can be easily derived as follows. Since there is an algorithm that solves k-SAT with the $2^{n(1-1/k)}$ bound (Corollary 16.3.4), we can take $\alpha = 1-1/k$. Now we would like to choose k such as to minimize the exponent

$$\alpha n + \tfrac{4m}{2^{\alpha k}}$$

in the bound given by Theorem 16.4.1 above. Assuming that $m > n$, we take $k = 2\log(m/n) + c$ where c is a constant that will be chosen later. Then

$$\begin{aligned} \alpha n + \tfrac{4m}{2^{\alpha k}} &= n\left(1 - \tfrac{1}{k}\right) + \tfrac{4m}{2^{k-1}} \\ &= n\left(1 - \tfrac{1}{2\log(m/n)+c} + \tfrac{1}{(m/n)\,2^{c-3}}\right) \end{aligned}$$

The constant c can be chosen so that

$$2\log(m/n) + c < (m/n)\,2^{c-3}$$

for all $m > n$. Therefore, the exponent is bounded by $n(1 - 1/O(k))$, which gives us the currently best upper bound for SAT:

Proposition 16.4.2. *SAT can be solved (for example, by Algorithm 16.4.1) in time*

$$|F|^{O(1)} \cdot 2^{n\left(1 - \frac{1}{O(\log(m/n))}\right)}$$

where $m > n$.

16.4.2. Branching-based algorithms

Branching heuristics (aka DPLL algorithms) gave the first nontrivial upper bounds for k-SAT in the form $|F|^{O(1)} \cdot 2^{\alpha n}$ where α depends on k (Section 16.3). This approach was also used to obtain the first nontrivial upper bounds for SAT as functions of other parameters on input formulas: Monien and Speckenmeyer proved the $|F|^{O(1)} \cdot 2^{m/3}$ bound [MS80]; Kullmann and Luckhardt proved the $|F|^{O(1)} \cdot 2^{l/9}$ bound [KL97]. Both these bounds are improved by Hirsch in [Hir00]:

Theorem 16.4.3 ([Hir00]). *SAT can be solved in time*

1. $|F|^{O(1)} \cdot 2^{0.30897\,m}$;
2. $|F|^{O(1)} \cdot 2^{0.10299\,l}$.

To prove these bounds, two branching-based algorithms are built. They split the input formula F into either two formulas $F[x], F[\neg x]$ or four formulas $F[x,y], F[x,\neg y], F[\neg x, y], F[\neg x, \neg y]$. Then the following transformation rules are used (Section 16.1.2):

- *Elimination of unit clauses.*

- *Subsumption.*
- *Resolution with subsumption.*
- *Elimination of variables by resolution.* The rule is being applied as long as it does not increase m (resp., l).
- *Elimination of blocked clauses.* Given a formula F, a clause C, and a literal a in C, we say that C is *blocked* [Kul99] for a with respect to F if the literal $\neg a$ occurs only in those clauses of F that contain the negation of at least one of the literals occurring in $C \setminus \{a\}$. In other words, C is blocked for a if there are no resolvents by a of the clause C and any other clause in F. Given F and a, we define the assignment $I(a, F)$ to be consisting of the literal a plus all other literals b of F such that
  - b is different from a and $\neg a$;
  - the clause $\{\neg a, b\}$ is blocked for $\neg a$ with respect to F.

  The following two facts about blocked clauses are proved by Kullmann in [Kul99]:
  - If C is blocked for a with respect to F, then F and $F \setminus \{C\}$ are equisatisfiable.
  - For any literal a, the formula F is satisfiable iff at least one of the formulas $F[\neg a]$ and $F[I(a, F)]$ is satisfiable.

  The rule is based on the first statement: if C is blocked for a with respect to F, replace F by $F \setminus \{C\}$.
- *Black and white literals.* This rule [Hir00] generalizes the obvious fact that if all clauses in F contain the same literal then F is satisfiable. Let P be a binary relation between literals and formulas such that for a variable v and a formula F, at most one of $P(v, F)$ and $P(\neg v, F)$ holds. Suppose that each clause in F that contains a "white" literal w satisfying $P(w, F)$ also contains a "black" literal b satisfying $P(\neg b, F)$. Then, for any literal a such that $P(\neg a, F)$, the formula F can be replaced by $F[a]$.

Remark. The second bound is improved to $2^{0.0926\,l}$ using another branching-based algorithm [Wah05].

Formulas with constant clause density. Satisfiability of formulas with constant clause density ($m/n \leq \text{const}$) can be solved in time $O(2^{\alpha n})$ where $\alpha < 1$ and α depends on the clause density [AS03, Wah05]. This fact also follows from Theorem 16.4.3 for $m \leq n$ and from Corollary 16.4.2 for $m > n$. The latter also shows how α grows as m/n increases:

$$\alpha = 1 - \tfrac{1}{O(\log(m/n))}.$$

16.5. How large is the exponent?

Given a number of $2^{\alpha n}$-time algorithms for various variants of SAT, it is natural to ask how the exponents αn are related to each other (and how they are related to exponents for other combinatorial problems of similar complexity). Another natural question is whether α can be made arbitrarily small.

Recall that a language in **NP** can be identified with a polynomially balanced relation $R(x, y)$ that checks that y is indeed a solution to the instance x. (We

refer the reader to [Pap94] or some other handbook for complexity theory notions such as polynomially balanced relation or oracle computation.)

Definition 16.5.1. *Parameterized* **NP** *problem* (L, q) consists of a language $L \in \mathbf{NP}$ (defined by a polynomially-time computable and polynomially balanced relation R such that $x \in L \iff \exists y R(x, y)$) and a complexity parameter q, which is a polynomial-time computable[4] integer function that bounds the size of the shortest solution to R: $x \in L \iff \exists y(|y| \leq q(x) \wedge R(x, y))$.

For $L = \text{SAT}$ or k-SAT, natural complexity parameters studied in the literature are the number n of variables, the number m of clauses, and the total number l of literal occurrences. We will be interested whether these problems can be solved in time bounded by a *subexponential* function of the parameter according to the following definition.

Definition 16.5.2 ([IPZ01])**.** A parameterized problem $(L, q) \in \mathbf{SE}$ if for every positive integer k there is a deterministic Turing machine deciding the membership problem $x \in L$ in time $|x|^{O(1)} \cdot 2^{q(x)/k}$.

Our ultimate goal is then to figure out which versions of the satisfiability problem belong to **SE**. An *Exponential Time Hypothesis* (*ETH*) for a problem (L, q) says that $(L, q) \notin \mathbf{SE}$. A natural conjecture saying that the search space for k-SAT must be necessarily exponential in the number of variables is $(3\text{-SAT}, n) \notin \mathbf{SE}$.

It turns out, however, that the only thing we can hope for is a kind of completeness result relating the complexity of SAT to the complexity of other combinatorial problems. The completeness result needs the notion of a reduction. However, usual polynomial-time reductions are not tight enough for this. Even linear-time reductions are not always sufficient. Moreover, it is by far not straightforward even that ETH for $(k\text{-SAT}, n)$ is equivalent to ETH for $(k\text{-SAT}, m)$, see Section 16.5.1 for a detailed treatment of these questions.

Even if one assumes ETH for 3-SAT to be true, it is not clear how the exponents for different versions of k-SAT are related to each other, see Section 16.5.2.

16.5.1. SERF-completeness

The first attempt to investigate systematically ETH for k-SAT has been made by Impagliazzo, Paturi, and Zane [IPZ01] who introduced reductions that preserve the fact that the complexity of the parameterized problem is subexponential.

Definition 16.5.3 ([IPZ01])**.** A series $\{T_k\}_{k \in \mathbb{N}}$ of oracle Turing machines forms a *subexponential reduction family* (*SERF*) reducing parameterized problem (A, p) to problem (B, q) if T_k^B solves the problem $F \in A$ in time $|F|^{O(1)} \cdot 2^{p(F)/k}$ and queries its oracle for instances $I \in B$ with $q(I) = O(p(F))$ and $|I| = |F|^{O(1)}$. (The constants in $O(\cdot)$ do not depend on k.)

[4] Note that the definition is different from the one being used in the parameterized complexity theory (Chapter 17).

It is easy to see that SERF reductions are transitive and preserve subexponential time. In particular, once a SERF-complete problem for a class of parameterized problems is established, the existence of subexponential-time algorithms for the complete problem is equivalent to the existence of subexponential time algorithms for every problem in the class. It turns out that the class **SNP** defined by Kolaitis and Vardi [KV87] (cf. [PY91]) indeed has SERF-complete problems.

Definition 16.5.4 (parameterized version of **SNP** from [IPZ01])**.** A parameterized problem $(L, q) \in \mathbf{SNP}$ if L can be defined by a series of second-order existential quantifiers followed by a series of first-order universal quantifiers followed by a quantifier-free first-order formula in the language of the quantified variables and the quantified relations $f_1, \ldots, f_s$ and input relations $h_1, \ldots, h_u$ (given by tables of their values on a specific universum of size n):

$$(h_1, \ldots, h_u) \in L \iff \exists f_1, \ldots, f_s \forall p_1, \ldots, p_t \Phi(p_1, \ldots, p_t),$$

where Φ uses f_i's and h_i's. The parameter q is defined as the number of bits needed to describe all relations f_i's, i.e., $\sum_i n^{\alpha_i}$, where α_i is the arity of f_i.

Theorem 16.5.1 ([IPZ01])**.** *$(3\text{-}SAT, m)$ and $(3\text{-}SAT, n)$ are SERF-complete for* **SNP***.*

SERF-completeness of SAT with the parameter n follows straightforwardly from the definitions (actually, the reduction queries only k-SAT instances with k depending on the problem to reduce). In order to reduce $(k\text{-SAT}, n)$ to $(k\text{-SAT}, m)$, one needs a *sparsification procedure* which is interesting in its own right. Then the final reduction of $(k\text{-SAT}, m)$ to $(3\text{-SAT}, m)$ is straightforward.

Algorithm 16.5.5 (The sparsification procedure [CIP06])**.**

`Input:` k-CNF formula F.
`Output:` sequence of k-CNF formulas that contains a satisfiable formula if and only if F is satisfiable.
`Parameters:` integers θ_i for $i \geq 0$.

1. For $c = 1, \ldots, n$, for $h = c, \ldots, 1$, search F for $t \geq \theta_{c-h}$ clauses $C_1, C_2, \ldots, C_t \in F$ of length c such that $|\bigcap_{i=1}^{t} C_i| = h$. Stop after the first such set of clauses is found and let $H = \bigcap_{i=1}^{t} C_i$. If nothing is found, output F and exit.
2. Make a recursive call for $F \setminus \{C_1, \ldots, C_t\} \cup \{H\}$.
3. Make a recursive call for $F[\neg H]$.

Calabro, Impagliazzo, and Paturi [CIP06] show that, given k and sufficiently small ε, for

$$\theta_i = O\left(\left(\tfrac{k^2}{\varepsilon} \log \tfrac{k}{\varepsilon}\right)^{i+1}\right)$$

this algorithm outputs at most $2^{\varepsilon n}$ formulas such that

- for every generated formula, every variable occurs in it at most $(k/\varepsilon)^{3k}$ times;

- F is satisfiable if and only if at least one of the generated formulas is satisfiable.

Since the number of clauses in the generated formulas is bounded by a linear function of the number of variables in F, Algorithm 16.5.5 defines a SERF reduction from $(k\text{-SAT}, n)$ to $(k\text{-SAT}, m)$.

16.5.2. Relations between the exponents

The fact of SERF-completeness, though an important advance in theory, does not give per se the exact relation between the exponents for different versions of k-SAT if one assumes ETH for these problems. Let us denote the exponents for the most studied versions of k-SAT (here $\mathbf{RTime}[t(n)]$ denotes the class of problems solvable in time $O(t(n))$ by a randomized one-sided bounded error algorithm):

$$\begin{aligned}
s_k &= \inf\{\delta \geq 0 \,|\, k\text{-SAT} \in \mathbf{RTime}[2^{\delta n}]\},\\
s_k^{\text{freq.}f} &= \inf\{\delta \geq 0 \,|\, k\text{-SAT-}f \in \mathbf{RTime}[2^{\delta n}]\},\\
s_k^{\text{dens.}d} &= \inf\{\delta \geq 0 \,|\, k\text{-SAT with at most } dn \text{ clauses} \in \mathbf{RTime}[2^{\delta n}]\},\\
s^{\text{freq.}f} &= \inf\{\delta \geq 0 \,|\, \text{SAT-}f \in \mathbf{RTime}[2^{\delta n}]\},\\
s^{\text{dens.}d} &= \inf\{\delta \geq 0 \,|\, \text{SAT with at most } dn \text{ clauses} \in \mathbf{RTime}[2^{\delta n}]\},\\
\sigma_k &= \inf\{\delta \geq 0 \,|\, \text{Unique } k\text{-SAT} \in \mathbf{RTime}[2^{\delta n}]\},\\
s_\infty &= \lim_{k\to\infty} s_k,\\
s^{\text{freq.}\infty} &= \lim_{f\to\infty} s^{\text{freq.}f},\\
s^{\text{dens.}\infty} &= \lim_{d\to\infty} s^{\text{dens.}d},\\
\sigma_\infty &= \lim_{k\to\infty} \sigma_k.
\end{aligned}$$

In the remaining part of this section we survey the dependencies between these numbers. Note that we are interested here in randomized (as opposed to deterministic) algorithms, because some of our reductions are randomized.

k-SAT vs Unique k-SAT

Valiant and Vazirani [VV86] presented a randomized polynomial-time reduction of SAT to its instances having at most one satisfying assignment (Unique SAT). The reduction, however, neither preserves the maximum length of a clause nor is capable to preserve subexponential complexity.

As a partial progress in the above problem, it is shown in [CIKP03] that $s_\infty = \sigma_\infty$.

Lemma 16.5.2 ([CIKP03]). *$\forall k\ \forall \varepsilon \in (0, \frac{1}{4})$ $s_k \leq \sigma_{k'} + O(H(\varepsilon))$, where $k' = \max\{k, \frac{1}{\varepsilon}\ln\frac{2}{\varepsilon}\}$.*

Proposition 16.5.3 ([CIKP03]). *$s_k \leq \sigma_k + O(\frac{\ln^2 k}{k})$. Thus $s_\infty = \sigma_\infty$.*

In order to prove Lemma 16.5.2 [CIKP03] employs an *isolation procedure*, which is a replacement for the result of [VV86]. The procedure starts with *concentrating* the satisfying assignments of the input formula in a Hamming ball of small radius εn. This is done by adding a bunch of "size k'" random hyperplanes $\bigoplus_{i\in R} a_i x_i = b$ (encoded in CNF by listing the $2^{k'-1}$ k'-clauses that are implied by this equality), where the subset R of size k' and bits a_i, b are chosen at random, to the formula. For $k' = \max\{k, \frac{1}{\varepsilon}\ln\frac{2}{\varepsilon}\}$ and a linear number of hyperplanes, this yields a "concentrated" k'-CNF formula with substantial probability of success.

Once the assignments are concentrated, it remains to isolate a random one. To do that, the procedure guesses a random set of variables in order to approximate (from the above) the set of variables having different values in different satisfying assignments. Then the procedure guesses a correct assignment for the chosen set. Both things are relatively easy to guess, because the set is small.

k-SAT for different values of k

Is the sequence $s_3, s_4, \ldots$ of the k-SAT complexities an increasing sequence (as one may expect from the definition)? Impagliazzo and Paturi prove in [IP01] that the sequence s_k is strictly increasing infinitely often. More exactly, if there exists $k_0 \geq 3$ such that ETH is true for k_0-SAT, then for every k there is $k' > k$ such that $s_k < s_{k'}$. This result follows from the theorem below, which can be proved using both critical clauses and sparsification techniques.

Theorem 16.5.4 ([IP01]). $s_k \leq (1 - \Omega(k^{-1}))s_\infty$.

Note that [IP01] gives an explicit constant in $\Omega(k^{-1})$.

k-SAT vs SAT-f

The sparsification procedure gives immediately

$$s_k \leq s_k^{\mathrm{freq}.((k/\varepsilon)^{3k})} + \varepsilon \leq s_k^{\mathrm{dens}.((k/\varepsilon)^{3k})} + \varepsilon.$$

In particular, for $\varepsilon = k^{-O(1)}$ this means that k-SAT could be only slightly harder than (k-)SAT of exponential density, and $s_\infty \leq s^{\mathrm{freq}.\infty} \leq s^{\mathrm{dens}.\infty}$.

The opposite inequality is shown using the clause-shortening algorithm: substituting an $O(2^{s_k+\varepsilon n})$-time k-SAT algorithm and $m \leq dn$ into Theorem 16.4.1 we get (after taking the limit as $\varepsilon \to 0$)

$$s_k^{\mathrm{dens}.d} \leq s_k + \frac{4d}{2^{ks_k}}. \tag{16.1}$$

Choosing k as a function of d so that the last summand is smaller than the difference between s_k and s_∞, [CIP06] shows that taking (16.1) to the limit gives $s^{\mathrm{dens}.\infty} \leq s_\infty$, i.e.,

$$s_\infty = s^{\mathrm{freq}.\infty} = s^{\mathrm{dens}.\infty}.$$

16.6. Summary table

The table below summarizes the currently best upper bounds for 3-SAT, k-SAT for $k > 3$, and SAT with no restriction on clause length (the bounds are given up

to polynomial factors). Other "record" bounds are mentioned in previous sections, for example bounds for Unique k-SAT (Section 16.3.4), bounds as functions in m or l (Section 16.4.2), etc.

	randomized algorithms	deterministic algorithms
3-SAT	1.323^n [Rol06]	1.473^n [BK04]
k-SAT	$2^{n\left(1-\frac{\mu_k}{k-1}+o(1)\right)}$ [PPSZ98]	$(2-2/(k+1))^n$ [DGH+02]
SAT	$2^{n\left(1-\frac{1}{O(\log(m/n))}\right)}$ [CIP06][5]	$2^{n\left(1-\frac{1}{O(\log(m/n))}\right)}$ [CIP06][5]

16.7. Addendum for the 2nd Edition: Connections to Circuit Complexity

To prove an upper bound for a computational problem Π, it suffices to design a *single* algorithm that solves Π using resources (like time or space) that do not exceed this bound. To prove a lower bound for Π, one needs to show that *every* algorithm for solving Π requires a certain amount of resources. Ryan Williams showed in [Wil10, Wil11] that there is a nontrivial connection between these somewhat opposing tasks: a slight improvement over exhaustive search for Boolean satisfiability implies a nontrivial lower bound for circuit complexity.

In circuit complexity, the basic model of computation is a family $\{C_n\}_{n\in\mathbb{N}}$ where C_n is a circuit with n input bits. Such a family *decides* a language $L \subseteq \{0,1\}^*$ if for every n-bit string x, the circuit C_n outputs 1 on x if and only if $x \in L$. Since there may be no single algorithm producing every circuit C_n, complexity classes defined in terms of circuits are commonly referred to as "non-uniform" classes in contrast to "uniform" complexity classes, like **P** or **NP**, defined in terms of algorithms.

The most natural example of a non-uniform class is **P/poly** that consists of all languages decidable by families $\{C_n\}_{n\in\mathbb{N}}$ where the size of C_n is polynomial in n. It is clear that each polynomial-time algorithm can be simulated using a family of polynomial-size circuits and, therefore, **P** is a subset of **P/poly**. What about nondeterministic polynomial-time algorithms, can they be simulated by non-uniform families of polynomial-size circuits? It is commonly believed that **NP** $\not\subset$ **P/poly** but the current state of complexity theory is far from proving this (such a proof would imply **P** $\neq$ **NP**). Moreover, it is open whether much larger classes such as **NEXP** or $\mathbf{E}^{\mathbf{NP}}$ are contained in **P/poly**; the current best lower bound is **MAEXP** $\not\subset$ **P/poly** [BFT98].

A possible approach to proving lower bounds for non-uniform computations would be to restrict types of circuits. For example, **ACC** denotes the class of languages decidable by constant-depth polynomial-size circuits with the following gates: NOT gates, AND gates and OR gates of unbounded fan-in, MOD-m gates of unbounded fan-in (such a gate outputs 0 if the number of ones in the input is divisible by m and outputs 1 otherwise). The question of strong lower bounds for **ACC** was open until 2011 when Williams proved the bound **NEXP** $\not\subset$ **ACC**

[5]This bound can be derived from Lemma 5 in [CIP06], see Corollary 16.4.2 above.

building on connections between upper bounds and lower bounds [Wil10, Wil11]. These exciting results can be summarized as follows.

Let $\mathcal{C}$ be a class of circuits. For example, $\mathcal{C}$ can consists of all polynomial-size circuits over AND, OR, NOT, or $\mathcal{C}$ can be a class of circuits used in the definition of **ACC** above. Let $\mathcal{C}$-SAT denote the satisfiability problem for circuits of $\mathcal{C}$: given a circuit $C \in \mathcal{C}$, determine whether there is an input on which C outputs 1.

Theorem 16.7.1 ([Wil10, Wil11]). *Suppose there is a $c > 0$ such that $\mathcal{C}$-SAT can be solved on circuits with n inputs and n^k size in $O(2^n/n^c)$ time for every k. Then* **NEXP** *contains a language that cannot be decided by a family of polynomial-size circuits from the class $\mathcal{C}$.*

This theorem is proved by contradiction, assuming that $\mathcal{C}$-SAT can be solved by a $O(2^n/n^c)$ algorithm and **NEXP** has polynomial-size circuits from $\mathcal{C}$. These two assumptions together imply an algorithm that is "too good to exist": this algorithm could be used to simulate every nondeterministic algorithm running in $O(2^n)$ time by a nondeterministic algorithm running in $o(2^n)$ time, which contradicts the nondeterministic hierarchy theorem [SFM78].

Theorem 16.7.2 ([Wil11]). *Let $\mathcal{C}$ be the class of circuits with NOT gates and unbounded fan-in AND, OR, MOD-m gates (such circuits are used in the definition of the class* **ACC***). For every integer $d > 0$, there is an $\epsilon \in (0, 1)$ and there is an algorithm that solves $\mathcal{C}$-SAT on circuits with n inputs of depth d and size at most 2^{n^ϵ} in 2^{n-n^ϵ} time.*

The proof of this theorem relies on simulating **ACC** circuits by low degree polynomials [Yao90, BT91].

Proposition 16.7.3 ([Wil11]). $\mathbf{NEXP} \not\subset \mathbf{ACC}$.

There is also a bunch of results relating modest upper bounds for satisfiability to modest lower bounds for non-uniform classes. Here are two examples of such results.

Theorem 16.7.4 ([Wil10], Theorem 6.1). *If the Exponential Time Hypothesis is false, then $\mathbf{E}^{\mathbf{NP}}$ does not have linear size circuits.*

Theorem 16.7.5 ([JMV18], Corollary 7). *If 3-SAT is in time c^n for any $c < 2^{1/7} = 1.10\ldots$, then there exists a (non-Boolean) function $f\colon \{0,1\}^n \to \{0,1\}^2$ in $\mathbf{E}^{\mathbf{NP}}$ such that any circuit over the full basis computing it requires at least $3n$ (non-input) gates.*

References

[ABI+05] E. Allender, M. Bauland, N. Immerman, H. Schnoor, and H. Vollmer. The complexity of satisfiability problems: Refining Schaefer's theorem. In *Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science, MFCS 2005*, volume 3618 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2005.

[APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–132, 1979.

[AS03] V. Arvind and R. Schuler. The quantum query complexity of 0-1 knapsack and associated claw problems. In *Proceedings of the 14th Annual International Symposium on Algorithms and Computation, ISAAC 2003*, volume 2906 of *Lecture Notes in Computer Science*, pages 168–177. Springer, December 2003.

[BFT98] H. Buhrman, L. Fortnow, and T. Thierauf. Nonrelativizing separations. In *Proceedings of the 13th Annual IEEE Conference on Computational Complexity, Buffalo, New York, USA, June 15-18, 1998*, pages 8–12. IEEE Computer Society, 1998.

[BK04] T. Brueggemann and W. Kern. An improved local search algorithm for 3-SAT. *Theoretical Computer Science*, 329(1–3):303–313, December 2004.

[BT91] R. Beigel and J. Tarui. On ACC. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 1991*, pages 783–792, 1991. Journal version: Computational Complexity 4: 350-366 (1994).

[CHLL97] G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein. *Covering Codes*, volume 54 of *Mathematical Library*. Elsevier, Amsterdam, 1997.

[CIKP03] C. Calabro, R. Impagliazzo, V. Kabanets, and R. Paturi. The complexity of unique k-SAT: An isolation lemma for k-CNFs. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity, CCC 2003*, pages 135–141. IEEE Computer Society, 2003.

[CIP06] C. Calabro, R. Impagliazzo, and R. Paturi. A duality between clause width and clause density for SAT. In *Proceedings of the 21st Annual IEEE Conference on Computational Complexity, CCC 2006*, pages 252–260. IEEE Computer Society, 2006.

[CKS01] N. Creignou, S. Khanna, and M. Sudan. *Complexity Classifications of Boolean Constraint Satisfaction Problems*. Society for Industrial and Applied Mathematics, 2001.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[Dan81] E. Dantsin. Two propositional proof systems based on the splitting method. *Zapiski Nauchnykh Seminarov LOMI*, 105:24–44, 1981. In Russian. English translation: *Journal of Soviet Mathematics*, 22(3):1293–1305, 1983.

[DG84] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1:267–284, 1984.

[DGH+02] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, October 2002.

[DHW04] E. Dantsin, E. A. Hirsch, and A. Wolpert. Algorithms for SAT based on search in Hamming balls. In *Proceedings of the 21st Annual Sym-*

posium on Theoretical Aspects of Computer Science, STACS 2004, volume 2996 of *Lecture Notes in Computer Science*, pages 141–151. Springer, March 2004.

[DHW06] E. Dantsin, E. A. Hirsch, and A. Wolpert. Clause shortening combined with pruning yields a new upper bound for deterministic SAT algorithms. In *Proceedings of the 6th Conference on Algorithms and Complexity, CIAC 2006*, volume 3998 of *Lecture Notes in Computer Science*, pages 60–68. Springer, May 2006.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[Hir00] E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.

[IP01] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

[IPZ01] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity. *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[IT04] K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, page 328, January 2004.

[JMV18] H. Jahanjou, E. Miles, and E. Viola. Local reduction. *Inf. Comput.*, 261(Part 2):281–295, 2018.

[KL97] O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity. Technical report, Fachbereich Mathematik, Johann Wolfgang Goethe Universität, 1997.

[Kul99] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1–2):1–72, 1999.

[KV87] P. G. Kolaitis and M. Y. Vardi. The decision problem for the probabilities of higher-order properties. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC 1987*, pages 425–435. ACM, 1987.

[Luc84] H. Luckhardt. Obere Komplexitätsschranken für TAUT-Entscheidungen. In *Proceedings of Frege Conference 1984, Schwerin*, pages 331–337. Akademie-Verlag Berlin, 1984.

[MS79] B. Monien and E. Speckenmeyer. 3-satisfiability is testable in $O(1.62^r)$ steps. Technical Report Bericht Nr. 3/1979, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn, 1979.

[MS80] B. Monien and E. Speckenmeyer. Upper bounds for covering problems. Technical Report Bericht Nr. 7/1980, Reihe Theoretische Informatik, Universität-Gesamthochschule-Paderborn, 1980.

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, March 1985.

[Pap91] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 1991*, pages 163–169, 1991.

[Pap94] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PPSZ98] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1998*, pages 628–637, 1998.

[PPSZ05] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. *Journal of the ACM*, 52(3):337–364, May 2005.

[PPZ97] R. Paturi, P. Pudlák, and F. Zane. Satisfiability coding lemma. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1997*, pages 566–574, 1997.

[Pud98] P. Pudlák. Satisfiability — algorithms and logic. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science, MFCS 1998*, volume 1450 of *Lecture Notes in Computer Science*, pages 129–141. Springer, 1998.

[PY91] C. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[Rol05] D. Rolf. Derandomization of PPSZ for Unique-k-SAT. In *Proceedings of the 8th International Conference on Theory and Applications on Satisfiability Testing, SAT 2005*, volume 3569 of *Lecture Notes in Computer Science*, pages 216–225. Springer, June 2005.

[Rol06] D. Rolf. Improved bound for the PPSZ/Schöning algorithm for 3-SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:111–122, November 2006.

[Sch78] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, STOC 1978*, pages 216–226, 1978.

[Sch99] U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1999*, pages 410–414, 1999.

[Sch02] U. Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, 2002.

[Sch05] R. Schuler. An algorithm for the satisfiability problem of formulas in conjunctive normal form. *Journal of Algorithms*, 54(1):40–44, January 2005.

[SFM78] J. I. Seiferas, M. J. Fischer, and A. R. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25(1):146–167, 1978.

[VV86] L. Valiant and V. Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, 47:85–93, 1986.

[Wah05] M. Wahlström. An algorithm for the SAT problem for formulae of linear length. In *Proceedings of the 13th Annual European Symposium on Algorithms, ESA 2005*, volume 3669 of *Lecture Notes in Computer Science*, pages 107–118. Springer, October 2005.

[Wil10] R. Williams. Improving exhaustive search implies superpolynomial lower bounds. In *Proceedings of the 42nd Annual ACM Symposium on Theory of Computing, STOC 2010*, pages 231–240, 2010. Journal version: SIAM J. Comput., 42(3):1218–1244, 2013.

[Wil11] R. Williams. Non-uniform ACC circuit lower bounds. In *Proceedings of the 26th Annual IEEE Conference on Computational Complexity, CCC 2011*, pages 115–125, 2011. Journal version: J. ACM, 61(1):2:1–2:32, 2014.

[Yao90] A. C. Yao. On ACC and threshold circuits. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science, FOCS 1990*, pages 619–627, 1990.