Chapter 12

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, the possibility of solving SAT in subexponential time, etc.

In Section 12.1 we define terminology and notation used throughout the chapter. Section 12.2 addresses the question of which special cases of SAT are polynomial-time tractable and which ones remain $\text{NP}$-complete. The first non-trivial upper bounds for testing satisfiability were obtained for algorithms that solve $k$-SAT; such algorithms also form the core of general SAT algorithms. Section 12.3 surveys the currently fastest algorithms for $k$-SAT. Section 12.4 shows how to use bounds for $k$-SAT to obtain the currently best bounds for SAT. Section 12.5 addresses structural questions like “what else happens if $k$-SAT is solvable in time $\langle \ldots \rangle$?”. Finally, Section 12.6 summarizes the currently best bounds for the main cases of the satisfiability problem.

12.1. Preliminaries

12.1.1. 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 ($\text{CNF formula}$) 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 $k$-$\text{CNF}$ 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 $\{\text{true}, \text{false}\}$. 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.

$\text{SAT}$ denotes the language of all satisfiable CNF formulas (we sometimes refer to this language as General SAT to distinguish it from its restricted versions), $k$-$\text{SAT}$ is its subset consisting of $k$-CNF formulas, $\text{SAT}$-$f$ is the subset of $\text{SAT}$ containing formulas with each variable occurring at most $f$ times, $k$-$\text{SAT}$-$f$ is the intersection of the last two languages. Unique $k$-$\text{SAT}$ 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 ($O$, $\Omega$, $o$, 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

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

12.1.2. 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 \(\text{DP}_a(F)\) 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 \(\text{DP}_a(F)\) is smaller than in \(F\), replace \(F\) by \(\text{DP}_a(F)\).
- **Bounded resolution.** Add all resolvents of size bounded by some function of the formula parameters.

12.2. 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 \(\text{SAT}(\mathcal{C})\) 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: \(\text{SAT}(\mathcal{C})\) 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 \text{true} to all variables;
2. Every formula in the class is satisfied by assigning \text{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 12.2.2.

12.2.1. “Easy” classes

**Trivially satisfiable formulas.** A CNF formula \(F\) is satisfied by assigning \text{true} (resp. \text{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 $2^n$ vertices corresponding to all variables occurring in $F$ and their negations. Each clause $a \lor b$ in $F$ can be viewed as two implications $\neg a \rightarrow b$ and $\neg b \rightarrow a$. 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 $\delta$ 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.

12.2.2. Schaefer’s dichotomy theorem

Definition 12.2.1. A function $\phi : \{\text{true, false}\}^k \rightarrow \{\text{true, false}\}$ is called a Boolean constraint of arity $k$.

Definition 12.2.2. Let $\phi$ be a constraint of arity $k$ and $x_1, \ldots, x_k$ be a sequence of Boolean variables (possibly with repetitions). The pair $(\phi, (x_1, \ldots, x_k))$ is called a constraint application. Such a pair is typically written as $\phi(x_1, \ldots, x_k)$.

Definition 12.2.3. 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 $\phi$ evaluates to true on the truth values assigned by $A$. 

Definition 12.2.4. Let $\Phi = \{C_1, \ldots, C_m\}$ be a set of constraint applications and $A$ be a truth assignment to all variables occurring in $\Phi$. We say that $A$ is a satisfying assignment for $\Phi$ if $A$ satisfies every constraint application in $\Phi$.

Definition 12.2.5. Let $C$ be a set of Boolean constraints. We define $\text{SAT}(C)$ to be the following decision problem: given a finite set $\Phi$ of applications of constraints from $C$, is there a satisfying assignment for $\Phi$?

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

\[
\begin{align*}
\phi_1(x, y) &= x \lor y; \\
\phi_2(x, y) &= x \lor \neg y; \\
\phi_3(x, y) &= \neg x \lor \neg y; \\
\phi_4(x, y) &= \text{false}.
\end{align*}
\]

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

Theorem 12.2.1 (Schaefer’s dichotomy theorem [Sch78]). Let $C$ be a set of Boolean constraints. If $C$ satisfies at least one of the conditions (1)–(6) below then $\text{SAT}(C)$ is in P. Otherwise $\text{SAT}(C)$ is NP-complete.

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

The main part of this theorem is to prove that $\text{SAT}(C)$ is NP-hard if none of the conditions (1)–(6) holds for $C$. 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 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 $\#\text{SAT}$, the quantified satisfiability problem $\text{QSAT}$, the maximum satisfiability problem $\text{MAX SAT}$ (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 $C$ of constraints, how difficult is it to recognize whether $\text{SAT}(C)$ is “easy” or “hard”? That is, we are interested in the complexity of the following “meta-problem”: Given $C$, is $\text{SAT}(C)$ in P? As shown in [CKS01], the complexity of the meta-problem depends on how the constraints are specified:

- If each constraint in $C$ is specified by its set of satisfying assignments then the meta-problem is in P.
- If each constraint in $C$ is specified by a CNF formula then the meta-problem is co-NP-hard.
• If each constraint in $C$ is specified by a DNF formula then the meta-problem is co-NP-hard.

12.3. 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 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 7 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 12.4.2. 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 12.3.1), multistart random walks (Section 12.3.2), and cube covering proposed in an attempt to derandomize the random-walk technique (Section 12.3.3). Finally, we go briefly through recent improvements for particular cases (Section 12.3.4).

12.3.1. 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 $2^{-\binom{n-j}{j/k}}$.

Such SAT algorithms are usually called DPLL algorithms since the approach originates from [DP69, 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

\[
\sum_j 2^{-n(\lceil j/k \rceil)} \cdot \text{(number of } j \text{-isolated assignments)},
\]

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.

\textbf{Algorithm 12.3.1 (The PPSZ algorithm, \[PPSZ05\]).}

**Input:** \( k \)-CNF formula \( F \).

**Output:** \textit{yes}, if \( F \) is satisfiable, \textit{no} otherwise.\footnote{This algorithm and the other randomized algorithms described below may incorrectly say “no” on satisfiable formulas.}

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 \( 2^n\left(1-\frac{\mu_k}{k}+o(1)\right) \) times, where \( \mu_k = \sum_{j=1}^\infty \frac{1}{j(j+1/k)} \):
   
   (a) Apply unit clause elimination to \( F \).
   
   (b) Pick randomly a literal \( a \) still occurring in \( F \) and replace \( F \) by \( F[a] \).\footnote{Contrary to \[PPSZ05\], we choose a literal not out of all \( 2^n \) 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.}

\textbf{Theorem 12.3.1 \[PPSZ05\].} The PPSZ algorithm solves \( k \)-SAT in time

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

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

\subsection{12.3.2. 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,
the number of satisfied clauses. Another method was used in Papadimitriou’s algorithm for 2-SAT \cite{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 12.3.2 (The multistart random-walk algorithm).

Input: \( k \)-CNF formula \( F \).
Output: satisfying assignment \( A \) if \( F \) is satisfiable, \( \text{no} \) 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 \( \text{no} \) and halt.

This multistart random-walk algorithm with \( t = 1 \) and \( w = 2n^2 \) is Papadimitriou’s polynomial-time algorithm for 2-SAT \cite{Pap91}. The algorithm with \( t = (2 - 2/k)^n \) and \( w = 3n \) is known as Schöning’s algorithm for \( k \)-SAT where \( k \geq 3 \) \cite{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 \cite{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 \cite{Sch02}. Therefore, Schöning’s algorithm solves \( k \)-SAT with the following upper bound:

Theorem 12.3.2 (\cite{Sch02}). Schöning’s algorithm solves \( k \)-SAT in time

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

with error probability \( o(1) \).
12.3.3. 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 \cite{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 \( C \) 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 \( C \). When thinking of the centers \( w_1, \ldots, w_t \) as codewords, we identify the covering \( C \) with a covering code \cite{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 \textit{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 \cite{DGH+02} that for the particular procedure \textit{Search} the overall running time is minimum when \( R = n/(k + 1) \).

\textbf{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 \( 2^n H(R/n) \) 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 \textit{sphere covering bound} \cite{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. 12.1). More exactly, the cube-covering-based algorithm in \cite{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 \( 2^n \) 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 \( \text{Search}(F, A, R) \). 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 \( \text{Search}(F[l], A, R - 1) \). 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, \( \text{Search}(F, A, R) \) runs in time \( k^R \) up to a polynomial factor.

**Algorithm 12.3.3** (The cube-covering-based algorithm, \([\text{DGH}^+02]\)).

**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 \( \text{Search}(F, A, R) \) where \( A \) is the center of the ball. Return yes if at least one of these calls returns yes. Otherwise, return false.
Theorem 12.3.3 (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 12.3.4. The cube-covering-based algorithm solves \( k \)-SAT in time
\[
|F|^{O(1)} \cdot 2^{n(1-1/k)}.
\]

12.3.4. 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 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 12.3.3). 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.

12.4. 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 12.4.1).** This method gives the currently best bound of the form \( |F|^{O(1)} \cdot 2^{\alpha n} \) where \( \alpha \) depends on \( n \) and \( m \).

- **Branching (Section 12.4.2, see also Part 1, Chapter 7).** 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 \( \beta \) and \( \gamma \) are constants.


12.4.1. 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 $F' \lor F''$ 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 12.4.1 (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 $D_1 \lor \cdots \lor D_m$.
2. Test satisfiability of $F'$ using the cube-covering-based algorithm (Section 12.3).
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 12.4.1. 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 12.3.4), we can take $\alpha = 1 - \frac{1}{k}$. Now we would like to choose $k$ such as to minimize the exponent

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

in the bound given by Theorem 12.4.1 above. Assuming that $m > n$, we take $k = 2 \log(m/n) + c$ where $c$ is a constant that will be chosen later. Then

$$\alpha n + \frac{4m}{2^{\alpha k}} = n \left(1 - \frac{1}{k}\right) + \frac{4m}{2^{\alpha k}} = n \left(1 - \frac{1}{2 \log(m/n) + c} + \frac{1}{n/m \cdot 2^{\alpha k}}\right)$$

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 $n(1 - O(1))$, which gives us the currently best upper bound for SAT:

Proposition 12.4.2. SAT can be solved (for example, by Algorithm 12.4.1) in time $|F|^{O(1)} \cdot 2^{n(1 - O\left(\frac{\log(m/n)}{\log(m/n) + c}\right))}$ where $m > n$.

12.4.2. 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 $\alpha$ depends on $k$ (Section 12.3). 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 12.4.3. SAT can be solved in time

1. $|F|^{O(1)} \cdot 2^{0.30897 m}$;
2. $|F|^{O(1)} \cdot 2^{0.10294 l}$.

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 $F[x, y], F[\neg x, y], F[\neg x, \neg y], F[x, \neg y]$. Then the following transformation rules are used (Section 12.1.2):

- Elimination of unit clauses.
Chapter 12. Worst-Case Upper Bounds

- **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 equi-satisfiable.
  - 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 $2^{0.0926l}$ 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 $\alpha$ depends on the clause density [AS03, Wah05]. This fact also follows from Theorem 12.4.3 for $m \leq n$ and from Corollary 12.4.2 for $m > n$. The latter also shows how $\alpha$ grows as $m/n$ increases:

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

12.5. 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 $\alpha n$ are related to each other (and how they are related to exponents for other combinatorial problems of similar complexity). Another natural question is whether $\alpha$ can be made arbitrarily small.

Recall that a language in $\text{NP}$ can be identified with a polynomially balanced relation $R(x, y)$ that checks that $y$ is indeed a solution to the instance $x$. (We
Chapter 12. Worst-Case Upper Bounds

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

Definition 12.5.1. Parameterized NP problem \((L, q)\) consists of a language \(L \in \text{NP}\) (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 integer function that bounds the size of the shortest solution to \(R: x \in L \iff \exists y (|y| \leq q(x) \land R(x, y))\).

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

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

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 12.5.1 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 \(k\text{-SAT}\) are related to each other, see Section 12.5.2.

12.5.1. SERF-completeness

The first attempt to investigate systematically ETH for \(k\text{-SAT}\) 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 12.5.3 ([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 \(T_k^A\) 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 13).
Chapter 12. Worst-Case Upper Bounds

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 12.5.4 (parameterized version of SNP from [IPZ01]). A parameterized problem \((L, q) \in \text{SNP}\) if

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

where \(\Phi\) 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., \(\sum \alpha_i n^{\alpha_i}\), where \(\alpha_i\) is the arity of \(f_i\).

Theorem 12.5.1 ([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 \(k\text{-SAT}\) 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 12.5.5 (The sparsification procedure [CIP06]).

Input: \(k\text{-CNF}\) formula \(F\).
Output: sequence of \(k\text{-CNF}\) formulas that contains a satisfiable formula if and only if \(F\) is satisfiable.

Parameters: integers \(\theta_i\) 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 \(\varepsilon\), for

\[
\theta_i = O \left( \left( \frac{k^2}{\varepsilon} \log \frac{k}{\varepsilon} \right)^{i+1} \right)
\]

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 12.5.5 defines a SERF reduction from \((k\text{-SAT}, n)\) to \((k\text{-SAT}, m)\).

12.5.2. 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 \( k\text{-SAT} \) if one assumes ETH for these problems. Let us denote the exponents for the most studied versions of \( k\text{-SAT} \) (here \( \text{R Time}[t(n)] \) denotes the class of problems solvable in time \( O(t(n)) \) by a randomized one-sided bounded error algorithm):

\[
\begin{align*}
s_k &= \inf \{ \delta \geq 0 \mid k\text{-SAT} \in \text{R Time}[2^{\delta n}] \}, \\
s_{\text{freq}, k} &= \inf \{ \delta \geq 0 \mid k\text{-SAT-f} \in \text{R Time}[2^{\delta n}] \}, \\
s_{\text{dens}, k} &= \inf \{ \delta \geq 0 \mid k\text{-SAT with at most } dn \text{ clauses} \in \text{R Time}[2^{\delta n}] \}, \\
s_{\text{freq}, \text{freq}, k} &= \inf \{ \delta \geq 0 \mid k\text{-SAT-f} \in \text{R Time}[2^{\delta n}] \}, \\
s_{\text{dens}, \text{freq}, k} &= \inf \{ \delta \geq 0 \mid k\text{-SAT-f with at most } dn \text{ clauses} \in \text{R Time}[2^{\delta n}] \}, \\
s_{\text{dens}, \text{dens}, k} &= \inf \{ \delta \geq 0 \mid k\text{-SAT with at most } dn \text{ clauses} \in \text{R Time}[2^{\delta n}] \}, \\
\sigma_k &= \inf \{ \delta \geq 0 \mid k\text{-Unique SAT} \in \text{R Time}[2^{\delta n}] \}, \\
s_\infty &= \lim_{k \to \infty} s_k, \\
s_{\text{freq}, \infty} &= \lim_{f \to \infty} s_{\text{freq}, f}, \\
s_{\text{dens}, \infty} &= \lim_{d \to \infty} s_{\text{dens}, d}, \\
\sigma_\infty &= \lim_{k \to \infty} \sigma_k.
\end{align*}
\]

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.

\( k\text{-SAT} \) vs Unique \( k\text{-SAT} \)

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

**Proposition 12.5.3** ([CIKP03]). \( s_k \leq \sigma_k + O(\frac{\ln^2 k}{k}) \). Thus \( s_\infty = \sigma_\infty \).
In order to prove Lemma 12.5.2 \cite{CIKP03} employs an isolation procedure, which is a replacement for the result of \cite{VV86}. The procedure starts with concentrating the satisfying assignments of the input formula in a Hamming ball of small radius $\varepsilon n$. This is done by adding a bunch of “size $k'\varepsilon n$” 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 \cite{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 12.5.4** (\cite{IP01}). \(s_k \leq (1 - \Omega(k^{-1})) s_\infty\).

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

$k$-SAT vs SAT-$f$

The sparsification procedure gives immediately

\[ s_k \leq s_k^{\text{freq.}} ((k/\varepsilon)^{3k}) + \varepsilon \leq s_k^{\text{dens.}} ((k/\varepsilon)^{3k}) + \varepsilon. \]

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

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 12.4.1 we get (after taking the limit as $\varepsilon \to 0$)

\[ s_k^{\text{dens.}} \leq s_k + \frac{4d}{2^k s_k}. \quad (12.1) \]

Choosing $k$ as a function of $d$ so that the last summand is smaller than the difference between $s_k$ and $s_\infty$, \cite{CIP06} shows that taking \ref{12.1} to the limit gives $s^{\text{dens.}} \leq s_\infty$, i.e.,

\[ s_\infty = s^{\text{freq.}} = s^{\text{dens.}}. \]

12.6. 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 12.3.4), bounds as functions in $m$ or $l$ (Section 12.4.2), etc.

<table>
<thead>
<tr>
<th></th>
<th>randomized algorithms</th>
<th>deterministic algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-SAT</td>
<td>$1.323^n$ [Rol06]</td>
<td>$1.473^n$ [BK04]</td>
</tr>
<tr>
<td>$k$-SAT</td>
<td>$2^n\left(1 - \frac{1}{2^{k-1}} + o(1)\right)$ [PPSZ98]</td>
<td>$(2 - 2/(k + 1))^n$ [DGH+02]</td>
</tr>
<tr>
<td>SAT</td>
<td>$2^n\left(1 - \frac{1}{\log(m/n)}\right)$  [CIP06]</td>
<td>$2^n\left(1 - \frac{1}{\log(m/n)}\right)$  [CIP06]</td>
</tr>
</tbody>
</table>

References


\footnote{This bound can be derived from Lemma 5 in CIP06, see Corollary 12.4.2 above.}
Chapter 12. Worst-Case Upper Bounds


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


