Chain, Generalization of Covering Code, and Deterministic Algorithm for k-SAT

Sixue Liu
Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ 08540, USA
`http://www.cs.princeton.edu/~sixuel`
sixuel@cs.princeton.edu

Abstract

We present the current fastest deterministic algorithm for k-SAT, improving the upper bound $(2-2/k)^{n+o(n)}$ due to Moser and Scheder in STOC 2011. The algorithm combines a branching algorithm with the derandomized local search, whose analysis relies on a special sequence of clauses called chain, and a generalization of covering code based on linear programming.

We also provide a more intelligent branching algorithm for 3-SAT to establish the upper bound 1.32793^n, improved from 1.3303^n.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms

Keywords and phrases Satisfiability, derandomization, local search

Digital Object Identifier 10.4230/LIPIcs.ICALP.2018.88

Related Version A full version of the paper is available at `https://arxiv.org/abs/1804.07901`.

Acknowledgements I want to thank Yuping Luo, S. Matthew Weinberg and Periklis A. Papakonstantinou for helpful discussions, and the anonymous reviewers for their valuable comments.

1 Introduction

As the fundamental NP-complete problems, k-SAT and especially 3-SAT have been extensively studied for decades. Numerous conceptual breakthroughs have been put forward via continued progress of exponential-time algorithms, including randomized and deterministic ones.

The first provable algorithm solving k-SAT on n variables in less than 2^n steps is presented by Monien and Speckenmeyer, using the concept of autark assignment [10]. Later their bound 1.619^n for 3-SAT is improved to 1.579^n and 1.505^n respectively [15, 8]. One should note that these algorithms follow a branching manner, i.e., recursively reducing the formula size by branching and fixing variables deterministically, thus are called branching algorithms.

As for randomized algorithm, two influential ones are PPSZ and Schöning's local search [12, 16]. There has been a long line of research improving the bound $(4/3)^n$ of local search for 3-SAT, including HSSW and combining with PPSZ [5, 6], until Hertli closes the gap between unique and general cases for PPSZ [4] (by unique it means the formula has only one satisfying assignment). In a word, considering randomized algorithm, PPSZ for k-SAT is currently the fastest, although with one-sided error (see PPSZ in Table 1). Unfortunately, general PPSZ seems tough to derandomize due to the excessive usage of random bits [13].

In contrast to the hardness in derandomizing PPSZ, local search can be derandomized using the so-called covering code [2]. Subsequent deterministic algorithms focus on boosting local search for 3-SAT to the bounds 1.473^n and 1.465^n [1, 14]. In 2011, Moser and Scheder fully derandomize Schöning's local search with another covering code for the choice of flipping

45th International Colloquium on Automata, Languages, and Programming (ICALP 2018).
Editors: Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella;
Article No. 88; pp. 88:1–88:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 The rounded up base c in the upper bound c^n of our deterministic algorithm for k-SAT and the corresponding upper bound in previous results [9, 11, 2] and randomized algorithm [12, 4].

k	Our Results	Makino et al.	Moser&Scheder	Dantsin et al.	PPSZ(randomized)
3	**1.32793**	1.3303	1.33334	1.5	1.30704
4	**1.49857**	-	1.50001	1.6	1.46895
5	**1.59946**	-	1.60001	1.66667	1.56943
6	**1.66646**	-	1.66667	1.71429	1.63788

variables within the unsatisfied clauses, which is immediately improved by derandomizing HSSW for 3-SAT, leading to the current best upper bounds for k-SAT (see Table 1) [11, 9]. Since then, all random coins in Schöning's local search are replaced by deterministic choices, and the bounds remain untouched. How to break the barrier?

The difficulty arises in both directions. If attacking this without local search, one has to derandomize PPSZ or propose radically new algorithm. Else if attacking this from derandomizing local search-based algorithm, one must greatly reduce the searching space.

Our method is a combination of a branching algorithm and the derandomized local search. As we mentioned in the second paragraph of this paper, branching algorithm is intrinsically deterministic, therefore it remains to leverage the upper bounds for both of them by some tradeoff. The tradeoff we found is the weighted size of set of chains, where a chain is a sequence of clauses sharing variable with the clauses next to them only, such that a branching algorithm either solves the formula within desired time or returns a large enough set of chains. The algorithm is based on the study of autark assignment from [10] with further refinement, whose output can be regarded as a generalization of maximal independent clauses set from HSSW [5], which reduces the k-CNF to a $(k-1)$-CNF. The searching space equipped with chains is rather different from those in previous derandomizations [2, 9, 11]: It is a Cartesian product of finite number of non-uniform spaces. Using linear programming, we prove that such space can be perfectly covered, and searched by derandomized local search within aimed time. Additionally, unlike the numerical upper bound in HSSW, we give the closed-form.

The rest of the paper is organized as follows. In §2 we give basic notations, definitions related to chain and algorithmic framework. We show how to generalize covering code to cover any space equipped with chains in §3. Then we use such code in derandomized local search in §4. In §5, we prove upper bound for k-SAT, followed by an intelligent branching algorithm for 3-SAT in §6 for improvement. Some upper bound results are highlighted in Table 1, with main results formally stated in Theorem 14 of §5 and Theorem 21 of §6.

2 Preliminaries

2.1 Notations

We study formula in Conjunctive Normal Form (CNF). Let $V = \{v_i | i \in [n]\}$ be a set of n boolean variables. For all $i \in [n]$, a literal l_i is either v_i or $\bar{v}_i$. A clause C is a disjunction of literals and a CNF F is a conjunction of clauses. A k-clause is a clause that consists of exactly k literals, and an $\leq k$-clause consists of at most k literals. If every clause in F is $\leq k$-clause, then F is a k-CNF.

An *assignment* is a function $\alpha : V \mapsto \{0, 1\}$ that maps each $v \in V$ to truth value $\{0, 1\}$. A *partial assignment* is the function restricted on $V' \subseteq V$. We use $F|\alpha(V')$ to denote the formula derived by fixing the values of variables in V' according to partial assignment $\alpha(V')$.

A clause C is said to be *satisfied* by α if α assigns at least one literal in C to 1. F is *satisfiable* iff there exists an α satisfying all clauses in F, and we call such α a *satisfying assignment* of F. The k-SAT problem asks to find a satisfying assignment of a given k-CNF F or to prove its non-existence if F is unsatisfiable.

Let X be a literal or a clause or a collection of either of them, we use $V(X)$ to denote the set of all the variables appear in X. We say that X and X' are *independent* if $V(X) \cap V(X') = \emptyset$, or X *overlaps* with X' if otherwise.

A *word* of length n is a vector from $\{0,1\}^n$. The *Hamming space* $H \subseteq \{0,1\}^n$ is a set of words. Given two words $\alpha_1, \alpha_2 \in H$, the *Hamming distance* $d(\alpha_1, \alpha_2) = \|\alpha_1 - \alpha_2\|_1$ is the number of bits α_1 and α_2 disagree. The reason of using α for word as same as for assignment is straightforward: Giving each variable an index $i \in [n]$, a word of length n naturally corresponds to an assignment, which will be used interchangeably.

Throughout the paper, n always denotes the number of variables in the formula and will be omitted if the context is clear. We use $O^*(f(n)) = \text{poly}(n) \cdot f(n)$ to suppress polynomial factor, and use $\mathcal{O}(f(n)) = 2^{o(n)} \cdot f(n)$ to suppress sub-exponential factor.

2.2 Preliminaries for Chain

In this subsection, we propose our central concepts, which are the basis of our analysis.

▶ **Definition 1.** Given integers $k \geq 3$ and $\tau \geq 1$, a *τ-chain* $\mathcal{S}^{(k)}$ is a sequence of τ k-clauses $\langle C_1, \ldots, C_\tau \rangle$ satisfies that $\forall i, j \in [\tau]$, $V(C_i) \cap V(C_j) = \emptyset$ iff $|i - j| > 1$.

If the context is clear, we will use $\mathcal{S}$, τ-chain or simply chain for short.

▶ **Definition 2.** A set of chains $\mathcal{I}$ is called an *instance* if $\forall \mathcal{S}, \mathcal{S}' \in \mathcal{I}$, $V(\mathcal{S}) \cap V(\mathcal{S}') = \emptyset$ for $\mathcal{S} \neq \mathcal{S}'$.

In other words, each clause in chain only and must overlap with the clauses next to it (if exist), and chains in an instance are mutually independent.

▶ **Definition 3.** Given chain $\mathcal{S}$, define the *solution space* of $\mathcal{S}$ as $A \subseteq \{0,1\}^{|V(\mathcal{S})|}$ such that partial assignment α on $V(\mathcal{S})$ satisfies all clauses in $\mathcal{S}$ iff $\alpha(V(\mathcal{S})) \in A$. [1]

We define vital algebraic property of chain, which will play a key role in the construction of covering code.

▶ **Definition 4.** Let A be the solution space of chain $\mathcal{S}^{(k)}$, define $\lambda \in \mathbb{R}$ and $\pi : A \mapsto [0,1]$ as the *characteristic value* and *characteristic distribution* of $\mathcal{S}^{(k)}$ respectively, where λ and π are feasible solution to the following linear programming LP_A:

$$\sum_{a \in A} \pi(a) = 1$$

$$\lambda = \sum_{a \in A} \left(\pi(a) \cdot (\frac{1}{k-1})^{d(a,a^*)} \right) \qquad \forall a^* \in A$$

$$\pi(a) \geq 0 \qquad \forall a \in A$$

▶ Remark. The variables in LP_A are λ and $\pi(a)$ ($\forall a \in A$). There are $|A| + 1$ variables and $|A| + 1$ equality constraints in LP_A. One can work out the determinant of the coefficient matrix to see it has full rank, so the solution is unique if feasible. Specifically, $\lambda \in (0, 1)$.

[1] This essentially defines the set of all satisfying assignments for a chain. As a simple example in 3-CNF, 1-chain $\langle x_1 \vee x_2 \vee x_3 \rangle$ has solution space $A = \{0,1\}^3 \backslash 0^3$.

Algorithm 1: Algorithmic Framework.

Input: k-CNF F
Output: a satisfying assignment or `Unsatisfiable`

1: $\mathsf{BR}(F)$ solves F or returns an instance $\mathcal{I}$
2: **if** F is not solved **then**
3: $\quad\mathsf{DLS}(F, \mathcal{I})$
4: **end if**

2.3 Algorithmic Framework

Our algorithm (Algorithm 1) is a combination of a branching algorithm called BR, and a derandomized local search called DLS. BR either solves F or provides a large enough instance to DLS for further use, which essentially reduces the Hamming space exponentially.

3 Generalization of Covering Code

First of all, we introduce the covering code, then show how to generalize it for the purpose of our derandomized local search.

3.1 Preliminaries for Covering Code

The *Hamming ball* of *radius* r and *center* α $B_\alpha(r) = \{\alpha' | d(\alpha, \alpha') \leq r\}$ is the set of all words with Hamming distance at most r from α. A *covering code* of *radius* r for Hamming space H is a set of words $C(r) \subseteq H$ satisfies $\forall \alpha' \in H, \exists \alpha \in C(r)$, such that $d(\alpha, \alpha') \leq r$, i.e., $H \subseteq \bigcup_{\alpha \in C(r)} B_\alpha(r)$, and we say $C(r)$ *covers* H.

Let ℓ be a non-negative integer and set $[\ell]^* = [\ell] \cup \{0\}$, a set of covering codes $\{C(r) | r \in [\ell]^*\}$ is an *ℓ-covering code* for H if $\forall r \in [\ell]^*, C(r) \subseteq H$ and $H \subseteq \bigcup_{r \in [\ell]^*} \bigcup_{\alpha \in C(r)} B_\alpha(r)$, i.e., $\{C(r) | r \in [\ell]^*\}$ covers H.

The following lemma gives the construction time and size of covering codes for the uniform Hamming spaces $\{0,1\}^n$.

▶ **Lemma 5** ([2]). *Given $\rho \in (0, \frac{1}{2})$, there exists a covering code $C(\rho n)$ for Hamming space $\{0,1\}^n$, such that $|C(\rho n)| \leq O^*(2^{(1-h(\rho))n})$ and $C(\rho n)$ can be deterministically constructed in time $O^*(2^{(1-h(\rho))n})$, where $h(\rho) = -\rho \log \rho - (1-\rho) \log (1-\rho)$ is the* binary entropy function.

3.2 Generalized Covering Code

In this subsection we introduce our generalized covering code, including its size and construction time.

First of all we take a detour to define the *Cartesian product* of σ sets of words as $X_1 \times \cdots \times X_\sigma = \prod_{i \in [\sigma]} X_i = \{\uplus_{i \in [\sigma]} \alpha_i | \forall i \in [\sigma], \alpha_i \in X_i\}$, where $\uplus_{i \in [\sigma]} \alpha_i$ is the concatenation from α_1 to α_σ. Then we claim that the Cartesian product of covering codes is also a good covering code for the Cartesian product of the Hamming spaces they covered separately. The proof of this general result can be found in the full version of the paper.

▶ **Lemma 6.** *Given integer $\chi > 1$, for each $i \in [\chi]$, let H_i be a Hamming space and $C_i(r_i)$ be a covering code for H_i. If $C_i(r_i)$ can be deterministically constructed in time $O^*(f_i(n))$ and $|C_i(r_i)| \leq O^*(g_i(n))$ for all $i \in [\chi]$, then there exists covering code $\mathfrak{C}$ of radius $\sum_{i \in [\chi]} r_i$ for Hamming space $\prod_{i \in [\chi]} H_i$ such that $\mathfrak{C}$ can be deterministically constructed in time $O^*(\sum_{i \in [\chi]} f_i(n) + \prod_{i \in [\chi]} g_i(n))$ and $|\mathfrak{C}| \leq O^*(\prod_{i \in [\chi]} g_i(n))$.*

Algorithm 2: Derandomized Local Search: DLS.

Input: k-CNF F, instance $\mathcal{I}$
Output: a satisfying assignment or `Unsatisfiable`

```
1: construct covering code 𝔆 for Hamming space H(F, I) (Definition 9)
2: for every word α ∈ 𝔆 do
3:    if searchball-fast(F, α, r) find a satisfying assignment α* for F then
4:       return α*
5:    end if
6: end for
7: return Unsatisfiable
```

Our result on generalized covering code is given below. We give its proof sketch here, and the detailed proof can be found in full version of the paper.

▶ **Lemma 7.** *Let A be the solution space of chain $\mathcal{S}^{(k)}$ whose characteristic value is λ, for any $\nu = \Theta(n)$, there exists an ℓ-covering code $\{C(r)|r \in [\ell]^*\}$ for Hamming space $H = A^\nu$ where $\ell = \lfloor -\nu \log_{k-1} \lambda + 2 \rfloor$, such that $|C(r)| \leq O^*(\lambda^{-\nu}/(k-1)^r)$ and $C(r)$ can be deterministically constructed in time $O^*(\lambda^{-\nu}/(k-1)^r)$, for all $r \in [\ell]^*$.*

Proof Sketch. Firstly, we show the existence of such ℓ-covering code by a probabilistic method. For each $r \in [l]^*$, we build $C(r)$ from $\emptyset$ by repeating the following for $O^*(\lambda^{-\nu}/(k-1)^r)$ times independently: Choose ν words independently from A according to characteristic distribution π (Definition 4) and concatenate them to get a word $\alpha \in A^\nu$, then add α to $C(r)$ with replacement. Clearly, $|C(r)| \leq O^*(\lambda^{-\nu}/(k-1)^r)$. By deliberately choosing the repeating rounds, we can prove that the probability of the event that any code in A^ν is not covered by $C(r)$ is extremely small, such that a union bound for all codes is strictly less than 1, therefore proved the existence.

Secondly, we construct the code deterministically. W.l.o.g., let $d \geq 2$ be a constant divisor of ν. By partitioning H into d blocks and applying the approximation algorithm for the set covering problem in [2], we obtain a good covering code for each block within affordable time. Then we concatenate all covering code by taking their Cartesian product, and the size and construction time follows from Lemma 6. ◀

4 Derandomized Local Search

In this section, we present our derandomized local search (DLS), see Algorithm 2.

The algorithm first constructs the generalized covering code and stores it (Line 1), then calls searchball-fast (Line 3) to search inside each Hamming ball, where searchball-fast refers to the same algorithm proposed in [11], whose running time is stated in the following lemma.

▶ **Lemma 8** ([11]). *Given k-CNF F, if there exists a satisfying assignment α^* for F in $B_\alpha(r)$, then α^* can be found by searchball-fast in time $(k-1)^{r+o(r)}$.*

Our generalized covering code is able to cover the following Hamming space.

▶ **Definition 9.** Given k-CNF F and instance $\mathcal{I}$, the Hamming space for F and $\mathcal{I}$ is defined as $H(F, \mathcal{I}) = H_0 \times \prod_i H_i$, where:

- $H_0 = \{0,1\}^{n'}$ where $n' = n - |V(\mathcal{I})|$.
- $H_i = {A_i}^{\nu_i}$ for all i, where A_i is a solution space and $\nu_i = \Theta(n)$ is the number of chains in $\mathcal{I}$ with solution space A_i. [2]

[2] As we shall see in §5 and §6, there are only finite number of different solution spaces and finite elements

Apparently all satisfying assignments of F lie in $H(F,\mathcal{I})$, because $\prod_i H_i$ contains all assignments on $V(\mathcal{I})$ which satisfy all clauses in $\mathcal{I}$ and H_0 contains all possible assignments of variables outside $\mathcal{I}$. Therefore to solve F, it is sufficient to search the entire $H(F,\mathcal{I})$.

▶ **Definition 10.** Given $\rho \in (0, \frac{1}{2})$ and Hamming space $H(F,\mathcal{I})$ as above, for $L \in \mathbb{Z}^*$, define covering code $\mathfrak{C}(L)$ for $H(F,\mathcal{I})$ as a set of covering codes $\{C(r)|(r-\rho n') \in [L]^*\}$ satisfies that $C(r) \subseteq H(F,\mathcal{I})$ for all r and $H(F,\mathcal{I}) \subseteq \bigcup_{(r-\rho n')\in[L]^*} \bigcup_{\alpha\in C(r)} B_\alpha(r)$, i.e., $\mathfrak{C}(L)$ covers $H(F,\mathcal{I})$.

▶ **Lemma 11.** *Given Hamming space $H(F,\mathcal{I})$ and A_i, ν_i as above, let $L = \sum_i \ell_i$ where $\ell_i = \lfloor -\nu_i \log \lambda_i + 2 \rfloor$ and λ_i is the characteristic value of chain with solution space A_i. Given $\rho \in (0, \frac{1}{2})$, covering code $\mathfrak{C}(L) = \{C(r)|(r-\rho n') \in [L]^*\}$ for $H(F,\mathcal{I})$ can be deterministically constructed in time $O^*(2^{(1-h(\rho))n'} \prod_i {\lambda_i}^{-\nu_i})$ and $|C(r)| \leq O^*(2^{(1-h(\rho))n'}/(k-1)^{r-\rho n'} \prod_i {\lambda_i}^{-\nu_i})$ for all $(r-\rho n') \in [L]^*$.*

Proof. To construct $\mathfrak{C}(L)$ for $H(F,\mathcal{I})$, we construct covering code $C_0(\rho n')$ for $H_0 = \{0,1\}^{n'}$ and ℓ_i-covering code for $H_i = {A_i}^{\nu_i}$ for all i, then take a Cartesian product of all the codes. By Lemma 5, the time taken for constructing $C_0(\rho n')$ is $O^*(2^{(1-h(\rho))n'})$, and $|C_0(\rho n')| \leq O^*(2^{(1-h(\rho))n'})$. By Lemma 7, for each i, the time taken for constructing $C(r_i)$ for each $r_i \in [\ell_i]^*$ is $O^*({\lambda_i}^{-\nu_i}/(k-1)^{r_i})$ and $|C(r_i)| \leq O^*({\lambda_i}^{-\nu_i}/(k-1)^{r_i})$. So by Lemma 6, we have that $|C(r)|$ can be upper bounded by:

$$2^{(1-h(\rho))n'} \cdot \sum_{\sum_i r_i = r-\rho n'} \left(\prod_i O^*({\lambda_i}^{-\nu_i}/(k-1)^{r_i}) \right) = O^*(2^{(1-h(\rho))n'}/(k-1)^{r-\rho n'} \prod_i {\lambda_i}^{-\nu_i}).$$

The equality holds because L is a linear combination of ν_i with constant coefficients and $\nu_i = \Theta(n)$, thus there are $O(1)$ terms in the product since $\sum_i \nu_i \leq n$. Meanwhile, there are $O^*(1)$ ways to partition $(r-\rho n')$ into constant number of integers, thus the outer sum has $O^*(1)$ terms. Together we get an $O^*(1)$ factor in RHS.

The construction time includes constructing each covering code for H_i ($i \geq 0$) and concatenating each of them by Lemma 6, which is dominated by the concatenation time. As a result, the time taken to construct $C(r)$ for all $(r-\rho n') \in [L]^*$ is:

$$\sum_{(r-\rho n')\in[L]^*} O^*(2^{(1-h(\rho))n'}/(k-1)^{r-\rho n'} \prod_i {\lambda_i}^{-\nu_i}) = O^*(2^{(1-h(\rho))n'} \prod_i {\lambda_i}^{-\nu_i}),$$

because it is the sum of a geometric series. Therefore conclude the proof. ◀

Using our generalized covering code and applying Lemma 8 for searchball-fast (Line 3 in Algorithm 2), we can upper bound the running time of DLS.

▶ **Lemma 12.** *Given k-CNF F and instance $\mathcal{I}$, DLS runs in time $T_{DLS} = \mathcal{O}((\frac{2(k-1)}{k})^{n'} \cdot \prod_i {\lambda_i}^{-\nu_i})$, where $n' = n - |V(\mathcal{I})|$, λ_i is the characteristic value of chain $\mathcal{S}_i$ and ν_i is number of chains in $\mathcal{I}$ with the same solution space to $\mathcal{S}_i$.*

Proof. The running time includes the construction time for $\mathfrak{C}(L)$ and the total searching time in all Hamming balls. It is easy to show that the total time is dominated by the

in each solution space. Thus for those $\nu_i = o(n)$, we can enumerate all possible combinations of assignments on them and just get a sub-exponential slowdown, i,e., an $\mathcal{O}(1)$ factor in the upper bound.

Algorithm 3: Branching Algorithm BR for k-SAT.

Input: k-CNF F

Output: a satisfying assignment or `Unsatisfiable` or an instance $\mathcal{I}$

staring from $\mathcal{I} \leftarrow \emptyset$, **for** 1-chain $\mathcal{S} : V(\mathcal{I}) \cap V(\mathcal{S}) = \emptyset$, **do** $\mathcal{I} \leftarrow \mathcal{I} \cup \mathcal{S}$
if $|\mathcal{I}| < \nu n$ **then**
  for each assignment $\alpha \in \{\{0,1\}^k \backslash 0^k\}$ of $\mathcal{I}$ **do**
    solve $F|\alpha$ by deterministic $(k-1)$-SAT algorithm
    return the satisfying assignment if satisfiable
  end for
  return `Unsatisfiable`
else
  return $\mathcal{I}$
end if

searching time using Lemma 11, thus we have the following equation after multiplying a sub-exponential factor $\mathcal{O}(1)$ for the other $o(n)$ chains not in $\mathcal{I}$:

$$\begin{aligned} T_{\mathrm{DLS}} &= \mathcal{O}(1) \cdot \sum_{(r-\rho n') \in [L]^*} \Big(|C(r)| \cdot (k-1)^{r+o(r)} \Big) \\ &= \mathcal{O}(1) \cdot \sum_{(r-\rho n') \in [L]^*} \Big(O^*(2^{(1-h(\rho))n'} / (k-1)^{r-\rho n'} \prod_i \lambda_i{}^{-\nu_i}) \cdot (k-1)^{r+o(r)} \Big) \\ &= \mathcal{O}(2^{(1-h(\rho)+\rho \log(k-1))n'} \cdot \prod_i \lambda_i{}^{-\nu_i}) = \mathcal{O}((\frac{2(k-1)}{k})^{n'} \cdot \prod_i \lambda_i{}^{-\nu_i}). \end{aligned}$$

The first equality follows from Lemma 8, the second inequality is from Lemma 11, and the last equality follows by setting $\rho = \frac{1}{k}$. Therefore we proved this lemma. ◀

5 Upper Bound for k-SAT

In this section, we give our main result on upper bound for k-SAT.

A simple branching algorithm BR for general k-SAT is given in Algorithm 3: Greedily construct a maximal instance $\mathcal{I}$ consisting of independent 1-chains and branch on all satisfying assignments of it if $|\mathcal{I}|$ is small. [3] After fixing all variables in $V(\mathcal{I})$, the remaining formula is a $(k-1)$-CNF due to the maximality of $\mathcal{I}$. Therefore the running time of BR is at most:

$$T_{\mathsf{BR}} = \mathcal{O}((2^k-1)^{|\mathcal{I}|} \cdot c_{k-1}{}^{n-k|\mathcal{I}|}), \tag{1}$$

where $\mathcal{O}(c_{k-1}{}^n)$ is the worst-case upper bound of a deterministic $(k-1)$-SAT algorithm.

On the other hand, since there are only 1-chains in $\mathcal{I}$, by Lemma 12 we have:

$$T_{\mathrm{DLS}} = \mathcal{O}((\frac{2(k-1)}{k})^{n-k|\mathcal{I}|} \cdot \lambda^{-|\mathcal{I}|}). \tag{2}$$

It remains to calculate the characteristic value λ of 1-chain $\mathcal{S}^{(k)}$. We prove the following lemma for the unique solution of linear programming LP_A in Definition 4.

[3] W.l.o.g., one can negate all negative literals in $\mathcal{I}$ to transform the solution space of 1-chain to $\{0,1\}^k \backslash 0^k$.

▶ **Lemma 13.** *For* 1*-chain* $\mathcal{S}^{(k)}$*, let* A *be its solution space, then the characteristic distribution* π *satisfies*

$$\pi(a) = \frac{(k-1)^k}{(2k-2)^k - (k-2)^k} \cdot (1 - (\frac{-1}{k-1})^{d(a,0^k)}) \text{ for all } a \in A,$$

and the characteristic value $\lambda = \frac{k^k}{(2k-2)^k-(k-2)^k}$.

Proof. We prove that this is a feasible solution to LP_A. Constraint $\pi(a) \geq 0$ ($\forall a \in A$) is easy to verify. To show constraint $\sum_{a\in A} \pi(a) = 1$ holds, let $y = d(a, 0^k)$ and note there are $\binom{k}{y}$ different $a \in A$ with $d(a, 0^k) = y$, then multiply $\frac{(2k-2)^k-(k-2)^k}{(k-1)^k}$ on both sides:

$$\frac{(2k-2)^k - (k-2)^k}{(k-1)^k} \cdot \sum_{a\in A} \pi(a) = \sum_{1\leq y\leq k} \left((1 - (\frac{-1}{k-1})^y) \cdot \binom{k}{y} \right)$$

$$= \sum_{0\leq y\leq k} \binom{k}{y} - \sum_{0\leq y\leq k} \binom{k}{y} (\frac{-1}{k-1})^y = 2^k - (\frac{k-2}{k-1})^k.$$

Thus $\sum_{a\in A} \pi(a) = 1$ holds.

To prove $\lambda = \sum_{a\in A} \left(\pi(a) \cdot (\frac{1}{k-1})^{d(a,a^*)}\right)$, similar to the previous case, we multiply $\frac{(2k-2)^k-(k-2)^k}{(k-1)^k}$ on both sides. Note that adding the term at $a = 0^k$ does not change the sum, then for all $a^* \in A$, we have:

$$\text{RHS} = \sum_{a\in\{0,1\}^k} (1 - (\frac{-1}{k-1})^{d(a,0^k)}) \cdot (\frac{1}{k-1})^{d(a,a^*)}$$

$$= \sum_{a\in\{0,1\}^k} (\frac{1}{k-1})^{d(a,a^*)} - \sum_{a\in\{0,1\}^k} (-1)^{d(a,0^k)} (\frac{1}{k-1})^{d(a,0^k)+d(a,a^*)}.$$

The first term is equal to $(\frac{k}{k-1})^k = \text{LHS}$. To prove the second term is 0, note that $\exists i \in [k]$ such that some bit $a_i^* = 1$. Partition $\{0,1\}^k$ into two sets $S_0 = \{a \in \{0,1\}^k | a_i = 0\}$ and $S_1 = \{a \in \{0,1\}^k | a_i = 1\}$. We have the following bijection: For each $a \in S_0$, negate the i-th bit to get $a' \in S_1$. Then $d(a, 0^k) + d(a, a^*) = d(a', 0^k) + d(a', a^*)$ and $(-1)^{d(a,0^k)} = -(-1)^{d(a',0^k)}$, so the sum is 0. Therefore we verified the constraint and proved the lemma. ◀

Observe from (1) and (2) that T_{BR} is an increasing function of $|\mathcal{I}|$, while T_{DLS} is a decreasing function of it, so $T_{\mathsf{BR}} = T_{\mathsf{DLS}}$ gives the worst-case upper bound for k-SAT. We solve this equation by plugging in λ from Lemma 13 to get νn as the worst-case $|\mathcal{I}|$, and obtain the following theorem as our main result on k-SAT.

▶ **Theorem 14.** *Given* $k \geq 3$*, if there exists a deterministic algorithm for* $(k-1)$*-SAT that runs in time* $\mathcal{O}({c_{k-1}}^n)$*, then there exists a deterministic algorithm for* k*-SAT that runs in time* $\mathcal{O}({c_k}^n)$*, where* $c_k = (2^k-1)^\nu \cdot {c_{k-1}}^{1-k\nu}$ *and* $\nu = \frac{\log(2k-2)-\log k-\log c_{k-1}}{\log(2^k-1)-\log(1-(\frac{k-2}{2k-2})^k)-k\log c_{k-1}}$.

Note that the upper bound for 3-SAT implied by this theorem is $O(1.33026^n)$, but we can do better by applying Theorem 21 (presented later) for $c_3 = 3^{\log\frac{4}{3}/\log\frac{64}{21}} < 1.32793$ to prove all upper bounds for k-SAT ($k \geq 4$) in Table 1 of §1.

6 Upper Bound for 3-SAT

We provide a better upper bound for 3-SAT by a more intelligent branching algorithm.

First of all, we introduce some additional notations in 3-CNF simplification, then we present our branching algorithm for 3-SAT from high-level to all its components. Lastly we show how to combine it with derandomized local search to achieve a tighter upper bound.

Algorithm 4: Branching Algorithm BR for 3-SAT.

Input: 3-CNF F, clause sequence $\mathcal{C}$
Output: a satisfying assignment or `Unsatisfiable` or a clause sequence $\mathcal{C}$

1: simplify F by *procedure* $\mathcal{P}$
2: **if** $\perp \in F$ **then**
3:     **return** `Unsatisfiable`
4: **else if** *condition* Φ holds **then**
5:     stop the recursion, *transform* $\mathcal{C}$ to an instance $\mathcal{I}$ and **return** $\mathcal{I}$
6: **else if** F is 2-CNF **then**
7:     deterministically solve F in polynomial time
8:     **if** F is satisfiable **then**
9:         stop the recursion and **return** the satisfying assignment
10:     **else**
11:         **return** `Unsatisfiable`
12:     **end if**
13: **else**
14:     choose a clause C according to *rule* Υ
15:     for every satisfying assignment α_C of C, call $\mathsf{BR}(F|\alpha_C, \mathcal{C} \cup C^{\mathcal{F}})$
16:     **return** `Unsatisfiable`
17: **end if**

6.1 Additional Notations

For every clause $C \in F$, if partial assignment α satisfies C, then C is removed in $F|\alpha$. Otherwise, the literals in C assigned to 0 under α are removed from C. If all the literals in C are removed, which means C is unsatisfied under α, we replace C by $\perp$ in $F|\alpha$. Let $G = F|\alpha$, for every $C \in F$, we use C^F to denote the clause C in F and $C^G \in G$ the new clause derived from C by assigning variables according to α. We use $\mathcal{F}$ to denote the original input 3-CNF without instantiating any variable, and $C^{\mathcal{F}}$ is called the *original form* of clause C.

Let $\mathsf{UP}(F)$ be the CNF derived by running *Unit Propagation* on F until there is no 1-clause in F. Clearly F is satisfiable iff $\mathsf{UP}(F)$ is satisfiable, and UP runs in polynomial time [3].

We will also use the set definition of CNF, i.e., for a CNF $F = \bigwedge_{i \in [m]} C_i$, it is equivalent to write $F = \{C_i | i \in [m]\}$. Define $\mathcal{T}(F), \mathcal{B}(F), \mathcal{U}(F)$ as the set of all the 3-clauses, 2-clauses and 1-clauses in F respectively. We have that any 3-CNF $F = \mathcal{T}(F) \cup \mathcal{B}(F) \cup \mathcal{U}(F)$.

6.2 Branching Algorithm for 3-SAT

In this subsection, we give our branching algorithm for 3-SAT (Algorithm 4). The algorithm is recursive and follows a depth-first search manner:

- Stop the recursion when certain conditions are met (Line 4 and Line 8).
- Backtrack when the current branch is unsatisfiable (Line 3, Line 11 and Line 16).
- Branch on all possible satisfying assignments on a clause and recursively call itself (Line 15). Return `Unsatisfiable` if all branches return `Unsatisfiable`.
- Clause sequence $\mathcal{C}$ stores all the branching clauses from root to the current node.

It is easy to show this algorithm is correct as long as *procedure* $\mathcal{P}$ maintains satisfiability.

In what follows, we introduce (i) the *procedure* $\mathcal{P}$ for simplification (Line 1); (ii) the clause choosing *rule* Υ (Line 14); (iii) the *transformation* from clause sequence to instance (Line 5); (iv) the termination *condition* Φ (Line 4). All of them are devoted to analyzing the running time of BR as a function of instance.

6.2.1 Simplification Procedure

The simplification relies on the following two lemmas.

▶ **Lemma 15** ([10]). *Given* 3*-CNF* F *and partial assignment* α*, define* $\mathcal{TB}(F,\alpha) = \{C|C \in \mathcal{B}(UP(F|\alpha), C^F \in \mathcal{T}(F)\}$*. If* $\perp \notin UP(F|\alpha)$ *and* $\mathcal{TB}(F,\alpha) = \emptyset$*, then* F *is satisfiable iff* $UP(F|\alpha)$ *is satisfiable and* α *is called an* autark.

We refer our readers to Chapter 11 in [7] for a simple proof (also see full version of the paper). We also provide the following stronger lemma to further reduce the formula size.

▶ **Lemma 16.** *Given* 3*-CNF* F *and* $(l_1 \vee l_2) \in \mathcal{B}(F)$*, if* $\exists C \in \mathcal{TB}(F, l_1 = 1)$ *such that* $l_2 \in C$*, then* F *is satisfiable iff* $F \backslash C^F \cup C$ *is satisfiable.*

Proof. Clearly F is satisfiable if $F \backslash C^F \cup C$ is. Suppose $C = l_2 \vee l_3$ and let α be a satisfying assignment of F. If $\alpha(l_1) = 1$, then $\mathsf{UP}(F|l_1 = 1)$ is satisfiable, thus $F \backslash C^F \cup C$ is also satisfiable since $C \in \mathsf{UP}(F|l_1 = 1)$. Else if $\alpha(l_1) = 0$, then $\alpha(l_2) = 1$ due to $l_1 \vee l_2$, so α satisfies C and the conclusion follows. ◀

As a result, 3-CNF F can be simplified by the following polynomial-time *procedure* $\mathcal{P}$: for every $(l_1 \vee l_2) \in \mathcal{B}(F)$, if $l_1 = 1$ or $l_2 = 1$ is an autark, then apply Lemma 15 to simplify F; else apply Lemma 16 to simplify F if possible.

▶ **Lemma 17.** *After running* $\mathcal{P}$ *on* 3*-CNF* F*, for any* $(l_1 \vee l_2) \in \mathcal{B}(F)$ *and for any* 2*-clause* $C \in \mathcal{TB}(F, l_1 = 1)$*, it must be* $l_2 \notin C$*. This also holds when switching* l_1 *and* l_2*.*

Proof. If $\mathcal{TB}(F, l_1 = 1) = \emptyset$, then $l_1 = 1$ is an autark and F can be simplified by Lemma 15. If $C \in \mathcal{TB}(F, l_1 = 1)$ and $l_2 \in C$, then F can be simplified by Lemma 16. ◀

6.2.2 Clause Choosing Rule

Now we present our clause choosing *rule* Υ. By Lemma 15 we can always begin with branching on a 2-clause with a cost of factor 2 in the upper bound: Choose an arbitrary literal in any 3-clause and branch on its two assignments $\{0, 1\}$. This will result in a new 2-clause otherwise it is an autark and we fix it and continue to choose another literal.

Now let us show the overlapping cases between the current branching clause to the next branching clause. Let C_0 be the branching clause in the father node where $C_0^{\mathcal{F}} = l_0 \vee l_1 \vee l_2$, and let F_0 be the formula in the father node. The *rule* Υ works as follows: if $\alpha_{C_0}(l_1) = 1$, choose arbitrary $C_1 \in \mathcal{TB}(F_0, l_1 = 1)$; else if $\alpha_{C_0}(l_2) = 1$, choose arbitrary $C_1 \in \mathcal{TB}(F_0, l_2 = 1)$.

We only discuss the case $\alpha_{C_0}(l_1) = 1$ due to symmetry. We enumerate all the possible forms of $C_1^{\mathcal{F}}$ by discussing what literal is eliminated followed by whether l_2 or $\bar{l}_2$ is contained:

1. $C_1^{\mathcal{F}} \backslash C_1 = l_3$. C_1 becomes a 2-clause due to elimination of l_3. There are three cases: (i) $C_1 = l_2 \vee l_4$, (ii) $C_1 = \bar{l}_2 \vee l_4$ or (iii) $C_1 = l_4 \vee l_5$.
2. $C_1^{\mathcal{F}} \backslash C_1 = \bar{l}_1$. C_1 becomes a 2-clause due to elimination of $\bar{l}_1$. There are three cases: (i) $C_1 = l_2 \vee l_3$, (ii) $C_1 = \bar{l}_2 \vee l_3$ or (iii) $C_1 = l_3 \vee l_4$.
3. $C_1^{\mathcal{F}} \backslash C_1 = l_2$. This means $l_1 = 1 \Rightarrow l_2 = 0$, and $\alpha_{C_0}(l_1 l_2) = 11$ can be excluded.
4. $C_1^{\mathcal{F}} \backslash C_1 = \bar{l}_2$. This means $l_1 = 1 \Rightarrow l_2 = 1$, and $\alpha_{C_0}(l_1 l_2) = 10$ can be excluded.

Both Case 1.(i) and Case 2.(i) are impossible due to Lemma 17. To sum up, we immediately have the following by merging similar cases with branch number bounded from above:

- Case 1.(iii): it takes at most 3 branches in the father node to get $l_3 \vee l_4 \vee l_5$.
- Case 1.(ii), Case 2.(iii) and Case 4: it takes at most 3 branches in the father node to get $\bar{l}_1 \vee l_3 \vee l_4$ or $\bar{l}_2 \vee l_3 \vee l_4$.
- Case 3: it takes at most 2 branches in the father node to get $l_2 \vee l_3 \vee l_4$.
- Case 2.(ii): it takes at most 3 branches in the father node to get $\bar{l}_1 \vee \bar{l}_2 \vee l_3$.

To fit *rule* Υ, there must be at least one literal assigned to 1 in the branching clause. Except Case 2.(ii), we get a 2-clause C_1, and *rule* Υ still applies.

Now consider the case $C_1^{\mathcal{F}} = \bar{l}_1 \vee \bar{l}_2 \vee l_3$. If $\alpha(l_1 l_2) = 11$, we have $C_1^F = l_3$, otherwise we have $C_1^F = 1 \vee l_3$. In other words, the assignment satisfying $C_0 \wedge C_1$ should be $\alpha(l_1 l_2 l_3) \in \{010, 100, 011, 101, 111\}$. Note that $\alpha(l_3) = 0$ in the first two assignments, which does not fit *rule* Υ. In this case, we do the following: Choose an arbitrary literal in any 3-clause and branch on its two assignments $\{0, 1\}$. Continue this process we will eventually get a new 2-clause (Lemma 15). Now the first two assignments $\alpha(l_1 l_2 l_3) \in \{010, 100\}$ has 4 branches because of the new branched literal, and we have that all 7 branches fit *rule* Υ because either $l_3 = 1$ or there is a new 2-clause. Our key observation is the following: These 7 branches correspond to all satisfying assignments of $C_0 \wedge C_1$, which can be amortized to think that C_1 has 3 branches and C_0 has 7/3 branches. As a conclusion, we modify the last case to be:

- Case 2.(ii): it takes at most 7/3 branches in the father node to get $\bar{l}_1 \vee \bar{l}_2 \vee l_3$.

6.2.3 Transformation from Clause Sequence to Instance

We show how to transform a clause sequence $\mathcal{C}$ to an instance, then take a symbolic detour to better formalize the cost of generating chains, i.e., the running time of BR.

Similar to above, let C_1 be the clause chosen by *rule* Υ and let C_0 be the branching clause in the father node, moreover let C be the branching clause in the grandfather node. In other words, C_1, C_0, C are the last three clauses in $\mathcal{C}$. C_1 used to be a 3-clause in the father node since $C_1 \in \mathcal{T}(F)$, thus C_1 is independent with C because all literals in C are assigned to some values in F, so C_1 can only overlap with C_0. Therefore, clauses in $\mathcal{C}$ can only (but not necessarily) overlap with the clauses next to them.

By the case discussion in §6.2.2, there are only 4 overlapping cases between C_0 and C_1, which we call *independent* for $\langle l_0 \vee l_1 \vee l_2,\ l_3 \vee l_4 \vee l_5 \rangle$, *negative* for $\langle l_0 \vee l_1 \vee l_2,\ \bar{l}_1 \vee l_3 \vee l_4 \rangle$ or $\langle l_0 \vee l_1 \vee l_2,\ \bar{l}_2 \vee l_3 \vee l_4 \rangle$, *positive* for $\langle l_0 \vee l_1 \vee l_2,\ l_2 \vee l_3 \vee l_4 \rangle$ and *two-negative* for $\langle l_0 \vee l_1 \vee l_2,\ \bar{l}_1 \vee \bar{l}_2 \vee l_3 \rangle$. There is a natural mapping from clause sequence to a string.

▶ **Definition 18.** Let $\mathcal{C}$ be a clause sequence, define function $\zeta : \mathcal{C} \mapsto \Gamma^{|\mathcal{C}|}$, where $\Gamma = \{\mathtt{*}, \mathtt{n}, \mathtt{p}, \mathtt{t}\}$, satisfies that the i-th bit of $\zeta(\mathcal{C})$ is $\mathtt{*}$ if $\mathcal{C}_i$ and $\mathcal{C}_{i+1}$ are independent, or $\mathtt{n}$ if negative, or $\mathtt{p}$ if positive, or $\mathtt{t}$ if two-negative for all $i \in [|\mathcal{C}| - 1]$, and the $|\mathcal{C}|$-th bit of $\zeta(\mathcal{C})$ is $\mathtt{*}$. A τ-chain $\mathcal{S}$ is also a clause sequence of length τ, so ζ maps $\mathcal{S}$ to Γ^τ. Two chains $\mathcal{S}_1$ and $\mathcal{S}_2$ are *isomorphic* if $\zeta(\mathcal{S}_1) = \zeta(\mathcal{S}_2)$.

Then the *transformation* from $\mathcal{C}$ to $\mathcal{I}$ naturally follows: Partition $\zeta(\mathcal{C})$ by $\mathtt{*}$, then every substring corresponds to a chain, just add this chain to $\mathcal{I}$. Now we can formalize the cost.

▶ **Lemma 19.** *Given 3-CNF $\mathcal{F}$, let $\mathcal{C}$ be the clause sequence in time T of running BR$(\mathcal{F}, \emptyset)$, it must be $T \le O^*(2^{\kappa_1} \cdot 3^{\kappa_2} \cdot (7/3)^{\kappa_3})$, where κ_1 is the number of $\mathtt{p}$ in $\zeta(\mathcal{C})$, κ_2 is the number of $\mathtt{*}$ and $\mathtt{n}$ in $\zeta(\mathcal{C})$, and κ_3 is the number of $\mathtt{t}$ in $\zeta(\mathcal{C})$.*

Proof. By Definition 18 and case discussion in §6.2.2, the conclusion follows. ◀

6.2.4 Termination Condition

We show how the cost of generating chains implies the termination *condition* Φ. We map every chain to an integer as the *type* of the chain such that isomorphic chains have the same type. Formally, let $\mathcal{I}(\mathcal{C})$ be the instance transformed from $\mathcal{C}$, and let $\Sigma = \{\zeta(\mathcal{S}) | \mathcal{S} \in \mathcal{I}(\mathcal{C})\}$ be the set of distinct strings with no repetition. Define bijective function $g : \Sigma \mapsto [\theta]$ that maps each string $\zeta(\mathcal{S})$ in Σ to a distinct integer as the *type* of chain $\mathcal{S}$, where $\theta = |\Sigma|$ is the number of types of chain in $\mathcal{C}$ and g can be arbitrary fixed bijection. Define *branch number* b_i of type-i chain $\mathcal{S}$ as $b_i = 2^{\kappa_1} \cdot 3^{\kappa_2} \cdot (7/3)^{\kappa_3}$, where κ_1 is the number of `p` in $\zeta(\mathcal{S})$, κ_2 is the number of `*` and `n` in $\zeta(\mathcal{S})$, and κ_3 is the number of `t` in $\zeta(\mathcal{S})$. Also define the *chain vector* $\vec{\nu} \in \mathbb{Z}^\theta$ for $\mathcal{I}(\mathcal{C})$ satisfies $\nu_i = |\{\mathcal{S} \in \mathcal{I}(\mathcal{C}) | (g \circ \zeta)(\mathcal{S}) = i\}|$ for all $i \in [\theta]$, i.e., ν_i is the number of type-i chains in $\mathcal{I}(\mathcal{C})$. We can rewrite Lemma 19 as the following.

▶ **Corollary 20.** *Given 3-CNF $\mathcal{F}$, let $\mathcal{I}$ be the instance in time T of running $\mathsf{BR}(\mathcal{F}, \emptyset)$, it must be $T \le T_{\mathsf{BR}} = O^*(\prod_{i\in[\theta]} b_i^{\nu_i})$, where b_i is the branch number of type-i chain and $\vec{\nu}$ is the chain vector for $\mathcal{I}$.*

To achieve worst-case upper bound $\mathcal{O}(c^n)$ for solving 3-SAT, we must have $T_{\mathsf{BR}} \le \mathcal{O}(c^n)$, which is $\prod_{i=1}^{\theta} b_i^{\nu_i} \le c^n$. This immediately gives us the termination *condition* Φ: $(\sum_{i\in[\theta]} \nu_i \cdot \log b_i)/\log c > n$.

Therefore, we can hardwire such condition into the algorithm to achieve the desired upper bound, as calculated in next subsection.

6.3 Combination of Two Algorithms

By combining BR and DLS as in Algorithm 1, we have that the worst-case upper bound $\mathcal{O}(c^n)$ is attained when $T_{\mathsf{BR}} = T_{\mathsf{DLS}}$, which is:

$$c^n = \prod_{i\in[\theta]} b_i^{\nu_i} = (\frac{4}{3})^{n'} \cdot \prod_{i\in[\theta]} {\lambda_i}^{-\nu_i}, \tag{3}$$

followed by Corollary 20 and Lemma 12. Let η_i be the number of variables in a type-i chain for all $i \in [\theta]$, we have that $n' = n - |V(\mathcal{I})| = n - \sum_{i\in[\theta]} \eta_i \nu_i$. Taking the logarithm and divided by n, (3) becomes:

$$\log c = \sum_{i\in[\theta]} \frac{\nu_i}{n} \log b_i = \log \frac{4}{3} - \sum_{i\in[\theta]} \frac{\nu_i}{n} (\eta_i \log \frac{4}{3} + \log \lambda_i). \tag{4}$$

The second equation is a linear constraint over $\frac{1}{n} \cdot \vec{\nu}$, which gives that $\log c$ is maximized when $\nu_i = 0$ for all $i \ne \arg\max_{i\in[\theta]}\{\log b_i/(\log b_i + \eta_i \log \frac{4}{3} + \log \lambda_i)\}$.

Based on calculation of LP_A (see full version of the paper), we show that chain $\mathcal{S}$ with $\zeta(\mathcal{S}) = *$ (say, type-1 chain) corresponds to the maximum value above, namely:

$$\arg\max_{i\in[\theta]}\{\log b_i/(\log b_i + \eta_i \log \frac{4}{3} + \log \lambda_i)\} = 1.$$

In other words, all chains in $\mathcal{I}$ are 1-chain. Substitute $\lambda_1 = \frac{3}{7}, b_1 = 3, \eta_1 = 3$ and $\nu_i = 0$ for all $i \in [2, \theta]$ into (4), we obtain our main result on 3-SAT as follow.

▶ **Theorem 21.** *There exists a deterministic algorithm for 3-SAT that runs in time $\mathcal{O}(3^{n \log \frac{4}{3} / \log \frac{64}{21}})$.*

This immediately implies the upper bound $O(1.32793^n)$ for 3-SAT in Table 1 of §1.

References

1 Tobias Brüggemann and Walter Kern. An improved deterministic local search algorithm for 3-sat. *Theoretical Computer Science*, 329(1-3):303–313, 2004.

2 Evgeny Dantsin, Andreas Goerdt, Edward A Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic (2-2/(k+1))$^{\mathrm{n}}$ algorithm for k-sat based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002.

3 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

4 Timon Hertli. 3-sat faster and simpler—unique-sat bounds for ppsz hold in general. *SIAM Journal on Computing*, 43(2):718–729, 2014.

5 Thomas Hofmeister, Uwe Schöning, Rainer Schuler, and Osamu Watanabe. A probabilistic 3-sat algorithm further improved. In *19th Annual Symposium on Theoretical Aspects of Computer Science, STACS 2002*, pages 192–202. Springer, 2002.

6 Kazuo Iwama and Suguru Tamaki. Improved upper bounds for 3-sat. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, volume 4, pages 328–328, 2004.

7 Hans Kleine Büning and Oliver Kullmann. Minimal unsatisfiability and autarkies. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 339–401. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-339.

8 Oliver Kullmann. New methods for 3-sat decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.

9 Kazuhisa Makino, Suguru Tamaki, and Masaki Yamamoto. Derandomizing the HSSW algorithm for 3-sat. *Algorithmica*, 67(2):112–124, 2013.

10 Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10(3):287–295, 1985.

11 Robin A. Moser and Dominik Scheder. A full derandomization of schöning's k-sat algorithm. In *Proceedings of the Forty-third Annual ACM Symposium on Theory of Computing, STOC 2011*, pages 245–252, 2011.

12 Ramamohan Paturi, Pavel Pudlák, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. *J. ACM*, 52(3):337–364, 2005. doi:10.1145/1066100.1066101.

13 Daniel Rolf. Derandomization of PPSZ for unique- k-sat. In *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005*, pages 216–225, 2005.

14 Dominik Scheder. Guided search and a faster deterministic algorithm for 3-sat. In *the 3rd Latin American Theoretical Informatics Symposium, LATIN 2008*, pages 60–71, 2008.

15 Ingo Schiermeyer. Solving 3-satisfiability in less than 1.579^n steps. *Computer Science Logic*, pages 379–394, 1970.

16 Uwe Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999*, pages 410–414, 1999.