# 3-SAT $\in RTIME(\mathcal{O}(1.32793^n))$

## Improving Randomized Local Search by Initializing Strings of 3-Clauses

Daniel Rolf

Humboldt-Universität zu Berlin
Institut für Informatik
Unter den Linden 6
10099 Berlin, GERMANY
rolf@informatik.hu-berlin.de

**Abstract**

This paper establishes a randomized algorithm that finds a satisfying assignment for a satisfiable formula $F$ in 3-CNF in $\mathcal{O}(1.32793^n)$ expected running time. The algorithms is based on the analysis of so-called *strings*, which are sequences of 3-clauses where non-succeeding clauses do not share a variable and succeeding clauses share one or two variables. One the one hand, if there are not many independent strings, we can solve $F$ with a decent success probability, but on the other hand, if there are many strings, we use them to improve the running time of Schöning's 3-SAT algorithm. Within a string, propagation of unit clauses is used to find successors.

# 1 Introduction

Firstly, we make some common definitions. A *literal* is a variable or its negation. An assignment $a$ for a set of variables $X$ maps each variable in $X$ to 0 or 1. A literal $l$ is satisfied by $a$ if $X(l) = 1$ if $l$ is not negated resp. $X(\bar{l}) = 0$ if $l$ is negated. A *clause* is a set of literals based on different variables. A clause is satisfied by some assignment $a$ if at least one literal is satisfied by $a$. Two clauses are called *independent* if they do not have any variable in common. A *formula* is a set of clauses. A formula is satisfied by $a$ if each clause is satisfied by $a$. A $k-$clause is a clause of size $k$ and a $k-$formula is a set of clauses of size at most $k$. The empty clause is denoted by $\perp$, which is not satisfiable. Note, the empty formula is satisfiable, and a formula containing $\perp$ is not satisfiable.

The problem of deciding whether a $k-$formula $F$ has a satisfying assignment is well known as the k-SAT problem, which is NP-complete for $k > 2$. Hence, if $NP \neq P$ holds (which is widely assumed), there is no hope to find a polynomial time algorithm for the k-SAT problem for $k > 2$. $poly(n)$ is used to denote some polynomial over $n$ with $poly(n) \geq 1$. We will not consider polynomial factors in complexity calculations because we always expect an exponential expression which outweighs all polynomials for large problems, and because the number of clauses is $\mathcal{O}(n^3)$, polynomials that depend on the number of clauses can also be replaced by $poly(n)$. This paper deals with the 3-SAT problem, so we will use "formula" to stand for a $3-$formula.

A naive approach is to enumerate all possible assignments and to check for each one whether it satisfies $F$. This algorithm has $\mathcal{O}\left(poly(n) \cdot 2^n\right)$ running time at most. The evolution of expected running time bounds, which are somewhat below the deterministic ones, is given as [10, 11, 9, 1] with bounds of $\mathcal{O}(1.334^n)$, $\mathcal{O}(1.3302^n)$, $\mathcal{O}(1.32971^n)$, and $\mathcal{O}(1.3290^n)$. In 1999 in [10], Schöning established a beautiful randomized algorithm which we will discuss in Section 2, as we will then need it in Section 3. Using this algorithm, Schöning proved that 3-SAT can be solved in $\mathcal{O}\left((4/3 + \epsilon)^n\right)$ expected running time.

In Section 3, we show how to combine a randomized solver with Schöning's algorithm in a general way by exploiting information extracted during the solving process. This is based on the initial idea given in [11], which describes an $\mathcal{O}(1.3302^n)$ time randomized algorithm for 3-SAT. In Section 5, we apply this to Algorithm $\Psi_{strings}$ and establish an $\mathcal{O}(1.32793^n)$ expected running time bound, which is the currently best upper bound for 3-SAT. Although it was independently invented, the approach we use inin Section 4 is similar to the one given in [1].

## 2 Schöning's Algorithm

In 1999 in [10], Schöning established the following beautiful randomized algorithm.

**Algorithm 1:** $RW$(**formula** $F$, **assignment** $a$)

```
1   repeat for 3n steps {
2       if F(a) = 1 then return a
3       C := arbitrary clause in F that is not satisfied by a
4       Flip one literal in C uniformly at random in the assignment a
5   }
6   return null
```

To bound the running time of this algorithm, Schöning proved the following theorem, which bounds the success probability of this algorithm in terms of the hamming distance $d(a, a^*)$ of the initial assignment $a$ to some satisfying assignment $a^*$.

**Theorem 2.** *Let $F$ be a satisfiable formula on $n$ variables and $a^*$ be a satisfying assignment for $F$. For each initial assignment $a$, the probability that algorithm $RW(a)$ finds a satisfying assignment is at least $(1/2)^{d(a,a^*)}/poly(n)$.*

Immediately from Theorem 2, we have the following corollary.

**Corollary 3.** *Let $F$ be a satisfiable formula on $n$ variables and $a^*$ be a satisfying assignment for $F$. Let $p_a$ be a probability distribution that maps each assignment $a$ to some probability. The probability that algorithm $RW(F, a)$, where $a$ is an assignment selected at random according to $p_a$, finds a satisfying assignemnt is at least $\mathbb{E}[(1/2)^{d(a,a^*)}]/poly(n)$, where the expectation is computed with respect to $p_a$.*

Schöning used this to show that, if we draw some assignment uniformly at random and call Algorithm $RW$ with this assignment, we find a satisfying assignment with probability at least $(3/4)^n/poly(n)$, which immediately yields the $\mathcal{O}\left(poly(n) \cdot (4/3)^n\right)$ expected running time bound.

In Section 3, we will see how we can achieve better bounds using an optimized probability distribution $p_a$.

# 3 Combining *RW* and a Randomized 2-SAT Solver

A *local pattern* $\mathcal{P}$ is a tuple $(F_\mathcal{P}, n_\mathcal{P}, \mu_\mathcal{P}, p_\mathcal{P}, \lambda_\mathcal{P})$ where $F_\mathcal{P}$ is a formula on $n_\mathcal{P} > 0$ variables, $0 < \mu_\mathcal{P} < 1$ and $(3/4)^{n_\mathcal{P}} < \lambda_\mathcal{P} < 1$ are arbitrary reals, and $p_\mathcal{P}$ is a probability distribution that maps each assignment of $F_\mathcal{P}$ to some probability so that

$$\lambda_\mathcal{P} \geq \mathbb{E}[(1/2)^{d(a,a^*)}]/poly(n)$$

holds for all satisfying assignments $a^*$ of $F_\mathcal{P}$, where the expectation is computed with respect to $p_\mathcal{P}$. We define

$$o_\mathcal{P} := \frac{\ln \mu_\mathcal{P}}{\ln \mu_\mathcal{P} - \ln \lambda_\mathcal{P} \ + n_\mathcal{P} \ln(3/4)}$$

for a local pattern $\mathcal{P}$. How to compute such a probability distribution is a somewhat technical issue we deal with in Appendix 6. Moreover, we defer the description of $\mu_\mathcal{P}$ after the next algorithm and its lemma.

Let $\Psi$ be a non-empty but finite set of local patterns, then $\Psi$ is called a *local scheme*. Let $I$ be a mapping which maps each local pattern $\mathcal{P} \in \Psi$ to a set of formulas which have to be isomorphic to $F_\mathcal{P}$ with respect to arbitrary variable renaming and flipping of all signs of arbitrary variables. Furthermore, do not let each two different formulas drawn from $\bigcup_{\mathcal{P}\in\Psi} I(\mathcal{P})$ share variables, i.e. they are mutually independent. Then $I$ is a called an *instance* of $\Psi$. With $n_I$ we denote the total number of variables involved in the instance $I$.

The following algorithm uses an instance of a local scheme to run Algorithm *RW* with better initial assignments. The relevant properties are stated in the following lemma.

**Algorithm 4:** *IRWSolve*(**formula** $F$, **local scheme** $\Psi$, **instance** $I$)

1 $a :=$ uninitialized assignment for $F$

2 **for each** $\mathcal{P}$ in $\Psi$ and each formula $G \in I(\mathcal{P})$ **do** {

3 In $a$ draw the variables involved in $G$ at random using the assignment probability distribution $p_\mathcal{P}$ with respect to the isomorphic mapping from $F_\mathcal{P}$ to $G$

4 }

5 **for each** variable $x$ of $F$ that is not initialized yet **do**

6 Uniformly at random in $a$ assign 0 or 1 to $x$

7 **return** $RW(F, a)$

**Lemma 5.** *Let $F$ be a satisfiable formula on $n$ variables, $\Psi$ a local schema, and $I$ an instance of $\Psi$ so that each formula in $I(\mathcal{P})$ is a sub formula of $F$. Then, with probability at least $\lambda/poly(n)$ where $\lambda$ is*

$$\prod_{\mathcal{P}\in\Psi} {\lambda_{\mathcal{P}}}^{|I(\mathcal{P})|} \cdot (3/4)^{n-n_I}$$

*Algorithm $IRWSolve(F, \Psi, I)$ returns a satisfying assignment for $F$.*

*Proof.* Fix some satisfying assignemnt $a^*$. From Corollary 3, we know that the success probability is at least $\mathbb{E}[(1/2)^{d(a,a^*)}]/poly(n)$ computed with respect to to the assignment probability distribution that has been used to setup the assignment. $IRWSolve$ initializes the assignment in blocks which are mutually independent to each other. Because the formulas in $I(\mathcal{P})$ are sub formulas of $F$, the partial assignment of $a^*$ on the variables involved in a certain sub formula also satisfies this sub formula. Thus, each block in step 3 has its own distribution $p_{\mathcal{P}}$ and expectation $\lambda_{\mathcal{P}}$ where $\mathcal{P}$ is the local pattern of that block, whereas the expectation in step 5 is $\frac{1}{2}2^0 + \frac{1}{2}2^{-1} = 3/4$. Because of independence, the entire expectation is computed by multiplying the expectations of all blocks, i.e. each expectation $\lambda_{\mathcal{P}}$ occurs $|I(\mathcal{P})|$ times, and $3/4$ occurs for the remaining variables. □

A randomized solver $(a^*, \mu, I) := \Psi_{sol}(F)$ is a polynomial time algorithm that makes a polynomial number of random decisions to solve a 3-SAT formula. This algorithm can be viewed as as if it would explore a top-down binary decision tree until it hits a leaf. If $a^*$ is *null* the leaf explored does not yield a satisfying assignment for $F$, otherwise $a^*$ is a satisfying assignment for $F$. A constant which has to be at most the path probability to that leaf, i.e. the product of the probabilities of the choice made to get to the leaf, is returned in $\mu$. The probability of the choice made at some decision is called the *cost* of that choice. Meaning that $\mu$ is the probability that this leaf is reached in some run of $\Psi_{sol}$. Moreover, $I$ is a local instance of $\Psi$, which is not only a byproduct of the exploration, but will be used for $IRWSolve$. However, $\prod_{\mathcal{P}\in\Psi} {\mu_{\mathcal{P}}}^{|I(\mathcal{P})|}$ must always be a lower bound for $\mu$. Furthermore, if $F$ is satisfiable, then at least one possible leaf reachable by $\Psi_{sol}(F)$ must contain a satisfying assignment for $F$.

The algorithmic idea is, if the decision tree is not "deep," then a satisfying leaf can be found quickly, yet if the decision tree is "deep," then a "long" path can be found quickly which will yield a good local instance for $IRWSolve$. The following polynomial time algorithm is a template for some algorithm

that combines Algorithm $IRWSolve$ and a randomized solver $\Psi_{sol}(F)$. This algorithm is a generalization of the one given in [1].

**Algorithm 6:** $Combine$**(formula** $F$**, local scheme** $\Psi$**)**

```
1    I := ∅, μ := 1
2    repeat {
3        (a, μ', I') := Ψ_sol(F)
4        if a ≠ null then return a
5        if μ' < μ then I := I', μ := μ'
6        a := IRWSolve(F, Ψ, I)
7        if a ≠ null then return a
8    }
```

We will now settle an important result for this algorithm.

**Proposition 7.** *Let $\Psi$ be a local scheme and $\Psi_{sol}$ a randomized reduction strategy. Then Algorithm $Combine(F, \Psi)$ finds a satisfying assignment for a satisfiable formula $F$ on $n$ variables in expected running time*

$$1/o \cdot poly(n) \tag{1}$$

*with*

$$o := e^{(\max_{\mathcal{P} \in \Psi} o_{\mathcal{P}}) \cdot \ln(3/4) n}.$$

*Proof.* For convenience, we write $i(\mathcal{P})$ to stand for $|I(\mathcal{P})|$. Let $\mu_m$ denote the minimum of the path probabilities to all leaves. Note, that this must not necessarily be the minimum of all $\mu$ returned by $\Psi_{sol}(F)$, since the $\mu$ returned by $\Psi_{sol}(F)$ is a lower bound and thus could be smaller. We have to deal with two cases.

Firstly, assume that $\mu_m \geq o$. At each repetition, $\Psi_{sol}(F)$ is called. Because $F$ is satisfiable, there is at least one $G$ which is also satisfiable and that has path probability $\mu_G \geq \mu_m$. So, in expected time at most $1/o$, a formula $G$ is found that is a complete reduction of $F$.

Secondly, assume that $\mu_m \leq o$. Then there is at least one possible result of $\Psi_{sol}$ with $\mu \leq o$. So, in expected running time at most $1/o$, $\mu \leq o$ will hold. After that, repeatedly calling $IRWSolve$ will return a satisfying assignment for $F$ after expected time at most $1/\lambda \cdot poly(n)$ with

$$\lambda := \prod_{\mathcal{P} \in \Psi} {\lambda_{\mathcal{P}}}^{i(\mathcal{P})} \cdot (3/4)^{n - n_I}$$

due to Lemma 5. Hence, the expected running time in that case is $1/o \cdot poly(n)$.

We claim that $o \geq \max\{\lambda, \mu\}$ for all possible instances $I$ of $\Psi$. Then $\mu \leq o$ implies $\lambda \geq o$. Thus in expected time at most $1/o \cdot poly(n)$, a satisfying assignment of $F$ is found.

Unfortunately, $\max\{\lambda, \mu\}$ depends on an actual instance $I$. Thus, to obtain what claimed, we have to establish a lower bound of $\max\{\lambda, \mu\}$ which does not care for $I$, i.e. we have to minimize $\max\{\lambda, \mu\}$ with respect to all possible instances $I$ of $\Psi$.

Firstly, we take the logarithm of both $\mu$ and $\lambda$ to obtain

$$\begin{aligned} l_\mu := \ln \mu &= \sum_{\mathcal{P} \in \Psi} (\ln \mu_\mathcal{P} \cdot i(\mathcal{P})) \text{ and} \\ l_\lambda := \ln \lambda &= \sum_{\mathcal{P} \in \Psi} ((\ln \lambda_\mathcal{P} - n_\mathcal{P} \ln(3/4)) \cdot i(\mathcal{P})) \\ &\quad + \ln(3/4) \cdot n. \end{aligned}$$

Since $\lambda_\mathcal{P} > (3/4)^{n_\mathcal{P}}$, $0 < \lambda_\mathcal{P} < 1$, and $0 < \mu_\mathcal{P} < 1$ hold for all $\mathcal{P} \in \Psi$, we observe that all coefficients of $i(\mathcal{P})$ in $l_\mu$ resp. $l_\lambda$ are negative resp. positive. $l_\mu = l_\lambda$ can be considered as a plane equation. So, fix some arbitrary point one that plane and vary $i(\mathcal{P})$ away from the plane in some direction. Because of the signs of the coefficients in $l_\mu$ resp. $l_\lambda$, depending on the direction, $l_\mu$ or $l_\lambda$ will not decrease. Thus the minimum of $l_\mu$ constrained to $l_\mu = l_\lambda$ is the global minimum of $\ln \max\{\lambda, \mu\}$ with respect to all possible instance $I$. Finally, consider the following linear program.

$$\begin{gathered} \text{Minimize } l_\mu \\ \text{with respect to } i(\mathcal{P}) \geq 0 \text{ for all } \mathcal{P} \in \Psi \\ \text{constrained to } l_\mu = l_\lambda. \end{gathered} \tag{2}$$

We know from the theory of linear programming (cf. [12]) that the minimum will be attained on some intersection of the border planes of the solution space. So, let $\mathcal{R} \in \Psi$ be a local pattern that maximizes $o_\mathcal{P}$ with respect to $\mathcal{P} \in \Psi$, and set $i(\mathcal{P})$ to 0 for all $\mathcal{P} \in \Psi - \mathcal{R}$. Moreover, we solve constraint (2) and set

$$i(\mathcal{R}) := \frac{\ln(3/4)}{\ln \mu_\mathcal{R} - \ln \lambda_\mathcal{R} + n_\mathcal{R} \ln(3/4)} n = o_\mathcal{R} \frac{\ln(3/4) \cdot n}{\ln \mu_\mathcal{R}}.$$

Observe that this is a feasible basic solution to the linear program, i.e. one that satisfies all constraints. We will rewrite the objective function $l_\mu$ using

the null (non-basic) variables (by replacing $i(\mathcal{R})$) and verify that all coefficients of non-basic variables are at least 0. Furthermore, we know that $l_\mu$ is a minimal solution if all non-basic variables in this rewritten form have coefficients at least 0. So, with $d_\mathcal{P}$ we denote the coefficient of $i(\mathcal{P})$ in the rewritten form and with $c^\mu_\mathcal{P}$ resp. $c^\lambda_\mathcal{P}$ in the original form of $l_\mu$ resp. $l_\lambda$. Furthermore, we obtain for all $\mathcal{P} \in \Psi - \mathcal{R}$

$$d_\mathcal{P} = c^\mu_\mathcal{P} - c^\mu_\mathcal{R} \frac{c^\lambda_\mathcal{P} - c^\mu_\mathcal{P}}{c^\lambda_\mathcal{R} - c^\mu_\mathcal{R}}.$$

To show that $d_\mathcal{P} \geq 0$ holds, we have to prove

$$c^\mu_\mathcal{P} \geq c^\mu_\mathcal{R} \frac{c^\lambda_\mathcal{P} - c^\mu_\mathcal{P}}{c^\lambda_\mathcal{R} - c^\mu_\mathcal{R}}.$$

So, we insert the actual values to obtain equivalently

$$\ln \mu_\mathcal{P} \geq \ln \mu_\mathcal{R} \frac{\ln \lambda_\mathcal{P} - \ln \mu_\mathcal{P} - n_\mathcal{P} \ln(3/4)}{\ln \lambda_\mathcal{R} - \ln \mu_\mathcal{R} - n_\mathcal{R} \ln(3/4)}$$

Observe that the numerator of the fraction is greater than 0 because of the definition of local patterns and the precondition for $\lambda_\mathcal{P}$. Thus we can divide by the numerator and by $-1$ to equivalently obtain

$$\frac{\ln \mu_\mathcal{P}}{\ln \mu_\mathcal{P} - \ln \lambda_\mathcal{P} + n_\mathcal{P} \ln \frac{3}{4}} \leq \frac{\ln \mu_\mathcal{R}}{\ln \mu_\mathcal{R} - \ln \lambda_\mathcal{R} + n_\mathcal{R} \ln \frac{3}{4}}, \text{ i.e.}$$
$$o_\mathcal{P} \leq o_\mathcal{R},$$

which is true due to the maximality of $o_\mathcal{R}$.

Finally, insert the minimal solution in $l_\mu$ for a lower bound on $\ln \min\{\lambda, \mu\}$, and obtain what claimed. □

A special randomized solver called $\Psi_{strings}$ will be introduced in Section 5. But before, we have to establish a little framework and prove some interesting properties of it in Section 4.

# 4 Unit Clause Propagation

Let $F$ be a formula and $l$ a literal. Let $F|_l$ be obtained by removing all clauses in $F$ that contain $l$ (they are satisfied by $l = 1$) and removing $\bar{l}$ from all clauses that contain $\bar{l}$ (they are not satisfied by $l = 1$, but lose one literal. Furthermore we call $l$ a *fixed* literal of $F|_l$, and we say $F|_l$ is obtained by *fixing* $l$ to 1 in $F$. Let $L = \{l_1, .., l_s\}$ be a finite set of literals. Then $F|_L$ is defined by fixing all literals of $L$ in $F$. A satisfying assignment for some formula $G$ that has been obtained by fixing some literals in a formula $F$ can be extending to satisfy $F$ using the fixed literals of $G$. Observe, that we may fix all literals in $F$ and obtain a formula $G$. Then $G$ is either empty or contains $\bot$. In the first case, the extended assignment of $G$ satisfies $F$. In the second case, the extended assignment does not satisfy $F$. Trying to fix a literal of $F$ that is already fixed to a different value will result in a contradiction, so to signal this, we automatically add $\bot$ to $F$ if this happens.

A *unit clause* is a clause that consists of exactly one literal. A formula that does not contain a unit clause is called *unit-free*. The following proposition shapes the kernel of our devised algorithms. But before, we define $\chi(F, G)$ to be the set of pairs $(C, D) \in F \times G$ with $D \subset C$ and $|D| = 2$.

**Proposition 8.** *Let $F$ be formula and $L$ a set of literals so that $F|_L$ is a unit-free formula. Then at least one of the following holds.*

*(1) If $F$ is satisfiable, then $F|_L$ is satisfiable.*

*(2) $F|_L$ contains $\bot$ and is thus not satisfiable.*

*(3) $\chi(F, F|_L)$ is not empty.*

*Proof.* If $F$ is not satisfiable, then obviously (1) holds. So, let $F$ be satisfiable. Assume that neither (3) nor (2) holds. We will show that (1) holds.

Let $a^*$ be a satisfying assignment of $F$. Assume that there is a clause $D \in F|_L$ that is not satisfied by $a^*$. Firstly, observe that, if $D \in F$ held, $D$ would be satisfied by $a^*$. However, $D \in F|_L$ holds, and thus there is a clause $C \in F$ with $D \subset C$, i.e. $D$ is obtained by removing at least one literal $\bar{l}$ with $l \in L$ from some clause $C \in F$. Firstly, $D$ cannot be $\bot$ since (2) is assumed to be wrong, but secondly, $D$ cannot be a unit clause since $F|_L$ is unit-free, and finally third, $D$ cannot be a 3-clause since then $C$ would have to be a 4-clause. Thus $D$ has to be a 2-clause, yet this violates our assumption, so such a clause $D \in F|_L$ cannot exist, and hence, we conclude that $F|_L$ is also satisfied by $a^*$, which completes the proof. □

A pair of formulas $(F, G)$ is said to be Proposition 8-conform, if $G$ is unit-free and there exists a set of literals $L$ with $G = F|_L$. We can apply this proposition to any case where a unit-free formula $G$ is obtained from a formula $F$ by fixing arbitrary literals in any order. Proposition 8 requires a unit-free formula $F|_L$, and we now discuss how to handle unit clauses. Observe that a unit clause forces its literal to a fixed value. So, it is quite easy to remove all unit clauses from some formula $F$, and that is done by the following trivial polynomial time algorithm.

**Algorithm 9:** $Simplify$**(formula** $F$**)**

```
1   while there exists a unit clause l ∈ F do
2       F := F|_l
3   return F
```

As stated before, a unit clause allows only one value for its literal, so we see that fixing does not alter satisfiability. We will use this behavior called unit clause propagation in Section 5 for reduction purposes. However, this also leads to the following.

**Observation 10.** *Let $F$ be a formula. Set $G := Simplify(F)$. Then $G$ is unit-free, and there exists a set of literals $L$ with $G = F|_L$.*

So, we can conclude the following.

**Lemma 11.** *Let $F$ be a unit-free formula, $L$ a set of literals, and $G := Simplify(F|_L)$. Then $(F, G)$ is Proposition 8-conform.*

*Proof.* Extend $L$ by all literals that have been fixed during $Simplify(F|_L)$, so $(F, G)$ is Proposition 8-conform. □

Furthermore, we present a simple yet powerful lemma.

**Lemma 12.** *Let $F$ be a formula and $ab$ a 2-clause in $F$. Set $G := Simplify(F|_a)$. Let $(C, D)$ be an arbitrary pair in $\chi(F, G)$. If $b \in D$, then $F$ is equivalent to $H := F - C + D$.*

*Proof.* Let $d$ be the literal of $C$ that is missing in $D$. Observe that $d$ can even be $\overline{a}$. Let $a^*$ be a satisfying assignment of $F$. If $a^*$ assigns 1 to $a$, then 0 must be assigned to $d$. If $a^*$ assigns 0 to $a$, then 1 must be assigned to $b$. In both cases $a^*$ satisfies $D$, i.e. $a^*$ satisfies $H$. Now, let $a^*$ be a satisfying assignment of $G$. Assume that $D$ is satisfied, but not $C$. That means that 0 is assigned to $d$. This implies that $a^*$ has to assign 0 to $a$ and thus has to assign 1 to $b$. But, $C$ contains $b$ and is thus satisfied by $a^*$, yielding a contradiction. □

A formula is *clean* if Lemma 12 is not applicable to any 2-clause in $F$. To *clean* a formula means to apply Lemma 12 as long as possible. For example, $\{ab, \overline{a}bc\}$ or $\{ab, bc, a\overline{c}d\}$ cannot occur in a clean formula.

The following lemma shows that cleaning preserves the properties of Proposition 8.

**Lemma 13.** *Let $F$ be a clean formula and $L$ a set of literals. Set $G := Simplify(F|_L)$ and let $H$ be the cleaned version of $G$. Then the following holds.*

*(1) If $G$ contains an empty clause, $H$ also contains an empty clause, and vice versa.*

*(2) If $\chi(F, G)$ is empty, then $\chi(F, H)$ is empty.*

*(3) If $\chi(F, G)$ is not empty, then $\chi(F, H)$ is not empty.*

*Proof.* (1) Obviously.

(2) Since $\chi(F, G)$ is empty, there is no 2-clause in $G$ that has not been already a 2-clause in $F$. Thus Lemma 12 is not applicable because $F$ is already clean and new 3-clauses cannot emerge, showing $G = H$.

(3) Cleaning does not decrease the number of 2-clauses of a formula. So, all 2-clauses in $G$ are also 2-clauses in $H$. That means $\chi(F, G) \subseteq \chi(F, H)$. □

This yields the following extension of $Simplify$.

**Algorithm 14:** $SimplifyClean$**(formula** $F$**)**

1 $F := Simplify(F)$

2 Clean $F$

3 **return** $F$

We will exploit these facts in our randomized solver presented in Section 5.

# 5 Randomized Solver Using Strings

Let $(C_1, .., C_l)$ be a sequence of 3-clauses so that successive clauses share no more than two, but at least one variable so and that non-successive clauses are independent. Then we call $(C_1, .., C_l)$ a *string* of length $l$. The type of a string $S$, denoted with $type(S)$, is built as follows. $type(S)$ is a sequence of the the same length like $S$. Each item in $type(S)$ describes how the corresponding succeeding clauses in $S$ are connected. $p$ means both clauses share exactly one variable and that with the same sign. $n$ means both clauses share exactly one variable and that with the different sign. $nn$ means both clauses share exactly two variables and both with different signs. These types are sufficient for our intentions. For example, the string $(abc, \overline{b}\overline{c}d, \overline{d}ef, fgh)$ has type $(nn, n, p)$. The types of strings are interesting because all strings of the same type are isomorphic with respect to arbitrary variable renaming and flipping of all signs of arbitrary variables. So, one example string of some type can be used as $F_{\mathcal{P}}$ in a local pattern $\mathcal{P}$. We want to establish a randomized solver which which will output a couple of string of types given in $\Psi$ as byproduct. These will serve as a local instance passed to $IRWSolve$. At first, we will present the algorithm, and then will look which strings could arise and what local patterns to create for them in $\Psi$. This is a bit messy and mostly deferred to Appendix 6.

At first, we describe how to find strings which allow a bound of $\mathcal{O}(poly(n) \cdot 1.32793^n)$ for the expected running time for 3-SAT. Secondly, we will use these ideas to create a set of algorithms that actually find those strings. The algorithms request a set of types $\mathcal{T}$ which has the following meaning. A string may grow very long, but we will see that we only have to deal with string of length at most 5. So, in $\mathcal{T}$, we provide a set of forced stop types, i.e. if a string has type in $\mathcal{T}$ it is not continued anymore. Beside these, there are cases where the algorithms internally decide to stop the string, these are called the internal stop types. We will made a number of choices to find the string each having some cost.

We split the whole algorithm in a number of tiny algorithms to make the analysis easier. Despite the fact that the algorithms call each other, they do not branch, i.e. will finish after polynomial time. We will make a lot of decisions and say that a decision propagates satisfiability if at least one choice yields a new formula that is a satisfiable if the old formula was satisfiable. Observe that all algorithms propagate satisfiability. So, if the input formula is satisfiable, there is a positive probability to find a satisfying assignment, i.e. there exists a satisfying leaf. Because we also want to return the path probability if a leaf is reached (cf. Section 3), we keep track of the cost.

All algorithms make us of some global variables: $F$ is the current formula,

$I$ containing the current local instance, $S$ contains the current sequence of clauses, $\mathcal{T}$ contains a set of stop string types, and $\mu$ is the current lower bound of the cost.

Moreover, we have to ensure that all clause sequences added to $I$ are really strings and all strings added to $I$ are independent. This is guaranteed by finding successors in $S$ using an advancing strategy as follows. We have some $S$ and all literals but at most the ones in the last clause are already fixed. In the next step we will always fix the last clause to get the next clause. So, step by step, we advance the string and are sure that only succeeding clauses can share variables. For example, if we have $(abc, def, fgh)$ as current clause sequence, then only $g$ and $h$ are not fixed, i.e. any existing 3-clause can share at most $f$ or $g$ with the string. Finally, we will only choose 3-clauses to continue with, which ensures independence to all strings already found because a 3-clause is always independent to all literals already fixed, otherwise it would not be a 3-clause.

The following algorithm is the is the algorithm we are going to use in *Combine*. It does some initialization and essentially calls *Start*. Finally, it returns a satisfying assignment if found, the path probability to the leaf explored, and the local instance containing all strings found.

**Algorithm 15:** $\Psi_{strings}$**(formula** $F$**)**

1 For all $\mathcal{P} \in \Psi$ set $I(\mathcal{P}) := \emptyset$

2 $S := ()$

3 *Start*

4 $\mu :=$ product of the probabilities of all choices

5 **if** $\bot \in F$ **then** $a := null$

6 **else** $a :=$ current assignment

7 **return** $(a, \mu, I)$

The next algorithm firstly the current clause sequence is splitted in mutually independent strings and cleared. This can be done by popping clauses from $S$ until the next clause would be independent to the already popped clauses. Then the clauses popped form a string because only variables succeeding in $S$ can share variables. This is repeated while $S$ is not empty. Finally, if there is a literal left, then control is passed to $Next2$ with the 2 possible assignments for this literal. Obviously, deciding among $c$ and $\overline{c}$ propagates satisfiability, which is a precondition of $Next2$ (see below).

**Algorithm 16:** *Start*()

1 **if** $S \neq ()$ **then** {

2 Split $S$ in mutually independent strings $\mathcal{S}$
3 **for each** $S' \in \mathcal{S}$ **do**
4 $I(type(S')) := I(type(S')) + S'$
5 $S := ()$
6 }
7 **if** $\exists$ literal $c$ not fixed in $F$ **then goto** $Next2(\{c\}, \{\overline{c}\})$

The following algorithm checks whether the last string in $S$ is a forced stop type. We will use this always after we extended $S$, and if we encounter a forced stop type, we will select an satisfying assignment for $G$ and will pass control to *Start* being going to flush $S$. In that case we have cost $1/g$ where $g$ is the number of satisfying assignments for $G$. Furthermore, $G$ has to contain all variables in $S$ that are not fixed yet, so that after fixing $G$ to some assignment, all 3-clauses are independent to $S$. On the other hand, this also implies that at most the last string in $S$ is a forced stop type. All other strings are internal stop types.

**Algorithm 17:** *CheckStop*(**sub-formula** $G$)
1 **if** $S \neq ()$ **then** {
2 $S' :=$ last string of $S$
3 **if** $type(S') \in \mathcal{T}$ **then** {
4 Uniformly at random, select $L$ from the set satisfying assignments for $G$
5 $F := F|_L$
6 **return** *true*
7 }
8 }
9 **return** *false*

The following algorithm takes two sets of literals. It relies on that deciding among $L_1$ and $L_2$ is satisfiability propagating. At first, it checks if one of the 2 sets can be excluded using Lemma 11 and Lemma 13. This is done by checking for $\perp$ firstly, which would imply that the other assignment must preserve satisfiability and thus can be fixed, and after that, by checking for an empty $\chi$, which would imply that this assignment preserves satisfiability and thus can be fixed. If a check is true, then the clause sequence is stopped with no cost. If all checks failed, then we can safely select among $L_1$ and $L_2$ and will find a 3-clause that has shortened to a 2-clause. In this cases the string is extended with cost 1/2. This way, the new clause will extend

$S$ using an $n-$, $p-$, or independent connection. If $S$ ends with a stop type now, then we set $D$ with cost 1/3.

**Algorithm 18:** $Next2$**(set of literals** $L_1$, $L_2$ **)**

```
1   Clean F
2   F_1 := SimplifyClean(F|_{L_1})
3   F_2 := SimplifyClean(F|_{L_2})
4   if ⊥ ∈ F_1 then F := F_2, goto Start
5   if ⊥ ∈ F_2 then F := F_1, goto Start
6   if χ(F, F_1) = ∅ then F := F_1, goto Start
7   if χ(F, F_2) = ∅ then F := F_2, goto Start
8   select at random {
9       w.p. 1/2: {
10          (C, D) := arbitrary pair in χ(F, F_1)
11          F := F_1
12          S := S ⊙ C
13          if CheckStop(D) then goto Start
14          goto Next3(D)
15      }
16      w.p. 1/2: same as previous case, but use F_2 instead of F_1
17  }
```

The following algorithm takes a 2-clause $ab$ that has to be in $F$. Then a number of checks similar to those in $Next2$ are done. If a check is true, then the clause sequence is stopped with no cost or control is passed to $Next2$ because one assignment could be excluded. Moreover, we check whether there is a clause $\overline{a}\overline{b}c$ with some appropriate $c$ is found in $F$. If that is true, we pass control to $Next5$, which deals with this special case. If all checks failed, then both $F_a$ and $F_b$ contain at least one 2-clause that was a 3-clause in $F$. So, we select among the 3 satisfying assignments for $ab$, use clause $C$ to extend $S$ with cost 1/3 and pass control to $Next3$. We show that $C$ neither contains $a$, $b$, nor $\overline{a}\overline{b}$. This way $C$ extends $S$ with an $n-$ or independent connection. If $\overline{a}\overline{b}$ is in $C$ then we would have passed control to $Next5$. Assume that $C$ contains $a$ or $b$. Without loss of generality, we say that this is $a$. So, we obtained $(C, D)$ from $\chi(F, F_b)$. But then we can apply Lemma 12 meaning that $F_b$ is not clean. But this cannot be true since $F_b$ was cleaned by $SimplifyClean$.

**Algorithm 19:** $Next3$**(2-clause** $ab$ **)**

1 Clean $F$

2 $F_a := SimplifyClean(F|_a)$

3 $F_b := SimplifyClean(F|_b)$

4 $F_{ab} := SimplifyClean(F|_{a,b})$

5 $F_{a\bar{b}} := SimplifyClean(F|_{a,\bar{b}})$

6 $F_{b\bar{a}} := SimplifyClean(F|_{b,\bar{a}})$

7 **if** $\bot \in F_a$ **then** $F := F_{b\bar{a}}$, **goto** $Start$

8 **if** $\chi(F, F_a) = \emptyset$ **then goto** $Next2(\{a, \bar{b}\}, \{a, b\})$

9 **if** $\bot \in F_b$ **then** $F := F_{a\bar{b}}$, **goto** $Start$

10 **if** $\chi(F, F_b) = \emptyset$ **then goto** $Next2(\{b, \bar{a}\}, \{b, a\})$

11 **if** $\bot \in F_{ab}$ **then goto** $Next2(\{a, \bar{b}\}, \{\bar{a}, b\})$

12 **if** $\chi(F, F_{ab}) = \emptyset$ **then** $F := F_{ab}$, **goto** $Start$

13 **if** $\bot \in F_{a\bar{b}}$ **then goto** $Next2(\{a, b\}, \{\bar{a}, b\})$

14 **if** $\chi(F, F_{a\bar{b}}) = \emptyset$ **then** $F := F_{a\bar{b}}$, **goto** $Start$

15 **if** $\bot \in F_{b\bar{a}}$ **then goto** $Next2(\{b, a\}, \{\bar{b}, a\})$

16 **if** $\chi(F, F_{b\bar{a}}) = \emptyset$ **then** $F := F_{b\bar{a}}$, **goto** $Start$

17 **if** $\exists c : \bar{a}\bar{b}c \in F$ **then goto** $Next5(ab, \bar{a}\bar{b}c)$

18 **select** at random {

19 **w.p.** 1/3: {

20 $(C, D) :=$ arbitrary pair in $\chi(F, F_a)$

21 $F := F_{a\bar{b}}$

22 $S := S \odot C$

23 **if** $CheckStop(D)$ **then goto** $Start$

24 **goto** $Next3(D)$

25 }

26 **w.p.** 1/3: same as previous cause, but swap $a$ and $b$

27 **w.p.** 1/3: {

28 $(C, D) :=$ arbitrary pair in $\chi(F, F_a)$

29 $F := F_{ab}$

30 $S := S \odot C$

31 **if** $CheckStop(D)$ **then goto** $Start$

32 **goto** $Next3(D)$

33 }

34 }

The following algorithm takes a 2-clause $ab$ and a 3-clause $\overline{a}\overline{b}c$ that has to be in $F$. It uses some probability constants $p_0$, $p_1$, $q_0$, and $q_1$ with $2p_0+p_1 = 1$ and $q_0 + q_1 = 1$, which are subject to optimization and given in Appendix 6. At first, we extend $S$ by $C$ resulting a $nn-$connection. If $S$ ends with a stop type now, then we set $ab$, $\overline{a}\overline{b}c$ with cost $1/5$ because there are only 5 satisfying assignments involving $a, b, c$. In the outer **select** statement we select one of the 3 satisfying assignments for $ab$, so this decision is satisfiability propagating. Selecting $a = 1$ and $b = 1$ implies $c = 1$, so we can use fix $c$ and pass control to $Start$ if we choose $a = 1$ and $b = 1$ as being done in the first case. This case has cost $p_1$. In the second (analogous third case), we choose $a = 1$ and $b = 0$. Then we check, whether choosing $c = 1$ may result in $\perp$ or empty $\chi$. If this is true, we can immediately set $c$ and pass control to $Start$ having cost $p_0$. If the checks fail, we choose setting $c = 0$ or setting $c = 1$. So, this decision also propagates satisfiability. If we set $c = 0$, we have cost $p_0q_0$ and pass control to $Start$. Otherwise, we extend $S$ by $C$ having cost $p_0q_1$ and resulting an $n-$ or independent connection, and we pass control to $Next3$. This is a $n-$ or independent connection because shortened clauses may contain $\overline{c}$, but not $c$. So, if we pass control to $Start$, we have cost at least $\min\{p_1, p_0q_0\}$, and if we hand over to $Next3$, we have cost $p_0q_1$.

**Algorithm 20:** $Next5$**(2-clause** $ab$**, 3-clause** $\overline{a}\overline{b}c$**)**

```
1   S := S ⊙ āb̄c
2   if CheckStop(ab, āb̄c) then goto Start
3   Clean F
4   select at random {
5       w.p. p1 : F := F|a,b,c, goto Start
6       w.p. p0 : {
7           F := F|a,b̄
8           Clean F
9           Fc := SimplifyClean(F|c)
10          Fc̄ := SimplifyClean(F|c̄)
11          if ⊥ ∈ Fc then F := Fc̄, goto Start
12          if χ(F, Fc) = ∅ then F := Fc, goto Start
13          select at random {
14              w.p. q0 : F := Fc̄, goto Start
15              w.p. q1 : {
16                (C, D) = arbitrary pair in χ(F, Fc)
17                F := Fc
```

18 $S := S \odot C$
19 **goto** $Next3(D)$
20 }
21 }
22 }
23 **w.p.** $p_0$ : same as previous case, but swap $a$ and $b$
24 }

We use $\Psi_{strings}$ as $\Psi_{red}$ and call $Combine(F, \Psi)$ to obtain our final algorithn. In Appendix 6, we explain how $\Psi$ and $\mathcal{T}$ are set up. These are rather technical issues and are omitted here. Yet, they show that the worst cases is determined by the strings of type () (i.e. single clauses), in that cases $\lambda_{()} = 3/7$, and $\mu_{()} = 1/3$. By inserting these values in the equation in Proposition 7, we obtain the following theorem, which is the main result of this paper.

**Theorem 21.** *Let $F$ be a satisfiable formula. Algorithm $Combine(F, \Psi)$ will find a satisfying assignment of $F$ in expected running time at most $\mathcal{O}(1.32793^n)$.*

Interestingly, the bound does not decrease if we try to seek for longer strings because the case () may always arise and thus will always be the limit, at least using our approach.

# 6 Local Scheme for Strings

We present a local schema $\Psi$ containing a couple of string type that are used to establish an $\mathcal{O}(1.32793^n)$ bound on the expected running time of our 3-SAT algorithm. We have splitted the strings into two different tables. The ones in the first table are those we put in $\mathcal{T}$, which will force the algorithm to stop a string if the type of the string is in $\mathcal{T}$. The other ones in the second table are those which occur if the algorithms themselves decide to stop a string. Both together form $\Psi$. Observe, that mirroring a string yields a different type, e.g. $(nn, n, n, p)$ compared to $(p, n, n, nn)$, but yields the same $F_{\mathcal{P}}$, so in that case that we have a type and its mirror, we take the one with the lower $\mu-$value into $\Psi$ so that this $\mu$ serves as a lower bound for both types.

The last column in the tables shows $e^{-o_{\mathcal{P}} \cdot \ln(3/4)}$. Due to Proposition 7, the largest of these determines the constant $c$ for the expected running time $poly(n) \cdot c^n$ of our algorithm. Instead of writing formulas for $F_{\mathcal{P}}$ we write only the types, but each string of such a type can stand for $F_{\mathcal{P}}$.

$\mu_{\mathcal{P}}$ can be calculated as follows. Assume we have a clause sequence $S$ that is being flushed in $Start$. Then $S$ has being found resulting a number of costs that we will split in individual costs for each string. The first choice always has cost 1/2 in $Next2$ called by $Start$. This will be postponed to the last string in $S$. At first, we consider all but the last string in $S$. A $n-$connection not preceded by a $nn-$connection has cost at least 1/3. A $p-$connection has cost 1/2. A $nn, n-$connection has cost $p_0 q_1$. The independent connection to the next string has cost at least 1/3. So, the cost of a non-ending string of $S$ is at least $p_0 q_1$ times the number of $nn, n-$connections multiplied by 1/3 times the number of $n-$connections not preceded by a $nn-$connection multiplied by 1/2 times the number of $p-$connections multiplied by 1/3 for the independent connection to the next string. Observe, these strings are internal stop types. Now, we consider the last string. At first, assume that the last string is also an internal stop type. If it ends with an $nn-$connection, this has cost $\min\{p_1, p_0 q_0\}$. In other cases, the last clause could be fixed with no cost. All other connections are similar to the previous non-ending string case. But, we have to include the initial 1/2. So, the cost of an ending, internal stop type string of $S$ is at least 1/2 multiplied by $p_0 q_1$ times the number of $nn, n-$connections multiplied by 1/3 times the number of $n-$connections not preceded by a $nn-$connection multiplied by 1/2 times the number of $p-$connections multiplied by $\min\{p_1, p_0 q_0\}$ if the string ends with an $nn-$connection. Finally, assume that the last string is a forced stop type. The the analysis is similar to the previous case, but we have to use

cost 1/5 for the last connection if it is an $nn-$connection (for fixing the $ab$, $\overline{a}\overline{b}c$), and we have to additionally multiply cost 1/3 if the last connection is not an $nn-$connection (for fixing $ab$).

Some cases may yield similar string patterns, in that cases we use the lowest $\mu_{\mathcal{P}}$.

To obtain the values for $\lambda_{\mathcal{P}}$ and the appropriate assignment probability distributions $p_{\mathcal{P}}$, maximize, with respect to all possible assignment distributions $p_{\mathcal{P}}$,

$$\min \left\{ \mathbb{E}\left[(1/2)^{d(a,a^*)}\right] | a^* \text{ satisfies } F_{\mathcal{P}} \right\}, \tag{3}$$

where the expectation is computed with respect to $p_{\mathcal{P}}$. Using the maximizing $p_{\mathcal{P}}$, assign to $\lambda_{\mathcal{P}}$ the minimum of $\mathbb{E}\left[(1/2)^{d(a,a^*)}\right]$ with respect to all possible satisfying assignments for $F_{\mathcal{P}}$, then $\lambda_{\mathcal{P}}$ is a lower bound whatever the real assignment of $F_{\mathcal{P}}$ is.

To calculate $\lambda_{\mathcal{P}}$ we have to find a probability distribution that maximizes equation (3). So, we have a classical max-min optimization problem and could form a linear program and solve it using the Simplex-Method constraining the $p_{\mathcal{P}}$ to be a probability distribution, i.e. summing to one and being at least 0. Although, we will use a different approach (cf. [9]). Instead of computing max-min we compute max-equal, i.e. find a probability distribution so that $\mathbb{E}\left[(1/2)^{d(a,a^*)}\right]$ is equal for all $a^*$ satisfying $F_{\mathcal{P}}$ and maximal. This can be done using any linear equation solver. But, beware this may yield an invalid $p_{\mathcal{P}}$, i.e. with some negative values, so we have to check that the $p_{\mathcal{P}}$ computed. Fortunately, this does not occur in our cases.

We have built the stop types in a way that a string is stopped as soon as it is yielding a bound that is below the worst case bound, as mentioned, already determined by type (). The probability constants $p_{\cdot}$ and $q_{\cdot}$, which affect the path probabilities of strings involving $nn-$connections, are set to the following values.

$$\begin{aligned} p_1 &= 61083/250000 \\ p_0 &= 188917/500000 \\ q_1 &= 44167/125000 \\ q_0 &= 80833/125000 \end{aligned}$$

We set them this way to get those strings below the worst case bound.

## 6.1 Forced Stop Types

| $F_{\mathcal{P}}$ | $n_{\mathcal{P}}$ | $\mu_{\mathcal{P}}$ | $\lambda_{\mathcal{P}}$ | $c \leq$ |
|---|---|---|---|---|
| $(p, p)$ | 7 | 1/24 | 243/1739 | 1.32790 |
| $(p, n, p)$ | 9 | 1/72 | 2187/27334 | 1.32773 |
| $(p, n, n, p)$ | 11 | 1/216 | 729/15904 | 1.32760 |
| $(p, n, n, n)$ | 11 | 1/324 | 729/15848 | 1.32777 |
| $(p, n, n, nn)$ | 10 | 1/180 | 1215/19894 | 1.32745 |
| $(p, n, nn)$ | 8 | 1/60 | 405/3799 | 1.32755 |
| $(p, nn)$ | 6 | 1/20 | 27/145 | 1.32765 |
| $(n, p, p)$ | 9 | 1/72 | 729/9110 | 1.32772 |
| $(n, p, n, p)$ | 11 | 1/216 | 2187/47732 | 1.32763 |
| $(n, p, n, n)$ | 11 | 1/324 | 729/15856 | 1.32780 |
| $(n, p, n, nn)$ | 10 | 1/180 | 405/6634 | 1.32748 |
| $(n, p, nn)$ | 8 | 1/60 | 45/422 | 1.32753 |
| $(n, n, p, p)$ | 11 | 1/216 | 2187/47704 | 1.32759 |
| $(n, n, p, n)$ | 11 | 1/324 | 729/15856 | 1.32780 |
| $(n, n, p, nn)$ | 10 | 1/180 | 1215/19888 | 1.32743 |
| $(n, n, n, p)$ | 11 | 1/324 | 729/15848 | 1.32777 |
| $(n, n, n, n)$ | 11 | 1/486 | 243/5264 | 1.32791 |
| $(n, n, n, nn)$ | 10 | 1/270 | 405/6608 | 1.32764 |
| $(n, n, nn)$ | 8 | 1/90 | 135/1262 | 1.32778 |
| $(n, nn, n, p)$ | 10 | $\frac{8343897139}{2250000000000}$ | 1215/19904 | 1.32790 |
| $(n, nn, n, n, p)$ | 12 | $\frac{8343897139}{6750000000000}$ | 10935/312692 | 1.32776 |
| $(n, nn, n, n, n)$ | 12 | $\frac{8343897139}{101250000000000}$ | 3645/103864 | 1.32789 |
| $(n, nn, n, n, nn)$ | 11 | $\frac{8343897139}{5625000000000}$ | 405/8692 | 1.32765 |
| $(n, nn, n, nn)$ | 9 | $\frac{8343897139}{1875000000000}$ | 675/8299 | 1.32777 |
| $(nn, n, p, p)$ | 10 | $\frac{8343897139}{1500000000000}$ | 405/6653 | 1.32768 |
| $(nn, n, p, n)$ | 10 | $\frac{8343897139}{2250000000000}$ | 405/6634 | 1.32789 |
| $(nn, n, p, nn)$ | 9 | $\frac{8343897139}{1250000000000}$ | 675/8321 | 1.32752 |
| $(nn, n, n, p)$ | 10 | $\frac{8343897139}{2250000000000}$ | 1215/19894 | 1.32787 |
| $(nn, n, n, n, p)$ | 12 | $\frac{8343897139}{6750000000000}$ | 3645/104168 | 1.32773 |
| $(nn, n, n, n, n)$ | 12 | $\frac{8343897139}{101250000000000}$ | 243/6920 | 1.32786 |
| $(n, nn, n, n, nn)$ | 11 | $\frac{8343897139}{5625000000000}$ | 405/8692 | 1.32765 |
| $(n, nn, n, nn)$ | 9 | $\frac{8343897139}{1875000000000}$ | 675/8299 | 1.32777 |
| $(nn, n, p, p)$ | 10 | $\frac{8343897139}{1500000000000}$ | 405/6653 | 1.32768 |
| $(nn, n, p, n)$ | 10 | $\frac{8343897139}{2250000000000}$ | 405/6634 | 1.32789 |
| $(nn, n, p, nn)$ | 9 | $\frac{8343897139}{1250000000000}$ | 675/8321 | 1.32752 |
| $(nn, n, n, p)$ | 10 | $\frac{8343897139}{2250000000000}$ | 1215/19894 | 1.32787 |
| $(nn, n, n, n, p)$ | 12 | $\frac{8343897139}{6750000000000}$ | 3645/104168 | 1.32773 |

| | | | | |
|---|---|---|---|---|
| $(nn, n, n, n, n)$ | 12 | $\frac{8343897139}{10125000000000}$ | 243/6920 | 1.32786 |
| $(nn, n, n, n, nn)$ | 11 | $\frac{8343897139}{5625000000000}$ | 225/4826 | 1.32762 |
| $(nn, n, n, nn)$ | 9 | $\frac{8343897139}{1875000000000}$ | 45/553 | 1.32774 |
| $(nn, n, nn)$ | 7 | $\frac{8343897139}{625000000000}$ | 25/176 | 1.32790 |

## 6.2 Internal Stop Types

| $F_{\mathcal{P}}$ | $n_{\mathcal{P}}$ | $\mu_{\mathcal{P}}$ | $\lambda_{\mathcal{P}}$ | $c \leq$ |
|---|---|---|---|---|
| $()$ | 3 | 1/3 | 3/7 | 1.32793 |
| $(p)$ | 5 | 1/6 | 81/331 | 1.32688 |
| $(p, n)$ | 7 | 1/18 | 81/578 | 1.32700 |
| $(p, n, n)$ | 9 | 1/54 | 729/9080 | 1.32702 |
| $(n)$ | 5 | 1/9 | 27/110 | 1.32755 |
| $(n, p)$ | 7 | 1/18 | 81/578 | 1.32700 |
| $(n, p, n)$ | 9 | 1/54 | 243/3028 | 1.32706 |
| $(n, n)$ | 7 | 1/27 | 9/64 | 1.32738 |
| $(n, n, p)$ | 9 | 1/54 | 729/9080 | 1.32702 |
| $(n, n, n)$ | 9 | 1/81 | 243/3016 | 1.32729 |
| $(n, nn, n, n)$ | 10 | $\frac{8343897139}{1687500000000}$ | 135/2204 | 1.32737 |
| $(n, nn, n)$ | 8 | $\frac{8343897139}{562500000000}$ | 405/3788 | 1.32745 |
| $(n, nn)$ | 6 | $\frac{15270727861}{375000000000}$ | 45/241 | 1.32769 |
| $(nn, n, p)$ | 8 | $\frac{8343897139}{375000000000}$ | 405/3799 | 1.32712 |
| $(nn, n, n, n)$ | 10 | $\frac{8343897139}{1687500000000}$ | 405/6608 | 1.32733 |
| $(nn, n, n)$ | 8 | $\frac{8343897139}{562500000000}$ | 135/1262 | 1.32741 |
| $(nn, n)$ | 6 | $\frac{8343897139}{187500000000}$ | 45/241 | 1.32753 |
| $(nn)$ | 4 | $\frac{15270727861}{125000000000}$ | 15/46 | 1.32793 |

# References


[1] S. Baumer and R. Schuler. Improving a probabilistic 3-sat algorithmy by dynamic search and independent clause pairs. *ECCC Report*, 2003.

[2] R. Beigel and D. Eppstein. 3-coloring in time $O(1.3446^n)$: a no-MIS algorithm. In *36th IEEE Symposium on Foundations of Computer Science*, pages 444–452, 1995.

[3] D. Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. In *Symposium on Discrete Algorithms*, pages 329–337, 2001.

[4] E. A. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg, 2001.

[5] I. Kersten and O. Riedlin. Algebra. Georg-August-Universität Göttingen, http://www.uni-math.gwdg.de/skripten/Algebraskript/algebra.pdf, 2001.

[6] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. pages 628–637.

[7] R. Paturi, P. Pudlak, and F. Zane. Satisfiability coding lemma. *Chicago Journal of Theoretical Computer Science*, 1999.

[8] R. Rodosek. A new approach on solving 3-satisfiability. In *AISMC: International Conference on Artificial Intelligence and Symbolic Mathematical Computing*, 1996.

[9] D. Rolf. 3-SAT $\in RTIME(1.32971^n)$. Diploma thesis, Department Of Computer Science, Humboldt University Berlin, Germany, Jan. 2003. http://www.informatik.hu-berlin.de/~rolf/papers/rolf033sat.html.

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

[11] R. Schuler, U. Schöning, and O. Watanabe. A probabilistic 3-sat algorithm further improved. In *STACS 2002, 19th Annual Symposium on Theoretical Aspects of Computer Science, Proceedings*, volume 2285 of *Lecture Notes in Computer Science*, pages 192–202. Springer, 2002.

[12] R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, Boston, 1996.