

First report on an adaptive density based  
branching rule for DLL-like SAT solvers, using a  
database for mixed random conjunctive normal  
forms created using the Advanced Encryption  
Standard (AES)\*

Oliver Kullmann

Computer Science Department  
University of Wales Swansea  
Swansea, SA2 8PP, UK

e-mail: O.Kullmann@Swansea.ac.uk

<http://cs-svr1.swan.ac.uk/~csoliver/>

March 15, 2002

**Abstract**

We introduce an *adaptive density-based heuristics*  $h_{\mathcal{A}}$  for a given (DLL-like, otherwise arbitrary) SAT solver  $\mathcal{A}$ , leading to a (hopefully) improved SAT solver  $\mathcal{A}'$ . The determination of  $h_{\mathcal{A}}$  is motivated by a *generalised threshold conjecture* for random formulas, and exploits a *database* for satisfiability and hardness of random formulas. To build up such a (large) database, a new reliable *pseudo-random formula generator* **OKgenerator**, based on AES (Advanced Encryption Standard), the successor of DES, is introduced.

## Contents

### 1 Introduction

---

\*Extended version of [12]

2

<b>2</b>	<b>Mixed random formulas</b>	<b>5</b>
2.1	Mixed random clause-sets and their densities . . . . .	5
2.2	Example: The $(2 + \lambda)$ -model . . . . .	6
2.3	Guaranteeing the existence of thresholds in a general setting . . . . .	6
<b>3</b>	<b>A generic branching rule</b>	<b>8</b>
<b>4</b>	<b>The random formulas generator OKgenerator, based on AES</b>	<b>10</b>
<b>A</b>	<b>AES — from bits to numbers</b>	<b>14</b>
<b>B</b>	<b>Generalised clause-sets</b>	<b>15</b>
<b>C</b>	<b>The implementation</b>	<b>16</b>
C.1	Standardised densities . . . . .	17
C.2	Other uses of the generator . . . . .	18
<b>D</b>	<b>An example calculation</b>	<b>19</b>
<b>E</b>	<b>A tool for simplified creation of large experiments</b>	<b>21</b>

## 1 Introduction

Let us consider a boolean formula  $F$  and the problem of finding a good (binary) branching for a given SAT solver  $\mathcal{A}$ . There are  $2 \cdot n(F)$  many choices, where  $n(F)$  is the number of variables, namely for each variable  $v \in \text{var}(F)$  the branchings  $v^{(0)} = \text{“first } \langle v \rightarrow 0 \rangle, \text{ then } \langle v \rightarrow 1 \rangle\text{”}$  and  $v^{(1)} = \text{“first } \langle v \rightarrow 1 \rangle, \text{ then } \langle v \rightarrow 0 \rangle\text{”}$ . An obvious criterion for choosing a good branching  $v^{(\varepsilon)}$  is to consider some sort of “expected run time” when processing the branching  $v^{(\varepsilon)}$ , and to choose a branching with minimal “expected run time”. I propose the following measure  $h_{\mathcal{A}}(v^{(\varepsilon)})$  of such a kind of expected run time:

$$h_{\mathcal{A}}(v^{(\varepsilon)}) := \Pr_{SAT}(F_{\varepsilon}) \cdot \text{av\_steps}_{SAT}^{\mathcal{A}}(F_{\varepsilon}) + \Pr_{USAT}(F_{\varepsilon}) \cdot (\Pr_{SAT}(F_{\bar{\varepsilon}}) \cdot \text{av\_steps}_{SAT}^{\mathcal{A}}(F_{\bar{\varepsilon}}) + \Pr_{USAT}(F_{\bar{\varepsilon}}) \cdot \text{av\_steps}_{USAT}^{\mathcal{A}}(F_{\bar{\varepsilon}}))$$

where

- $F_{\varepsilon} := \langle v \rightarrow \varepsilon \rangle * F$  is the result of the application of the partial assignment  $\langle v \rightarrow \varepsilon \rangle$  to  $F$ ;

- $\text{Pr}_{(U)SAT}(F_\varepsilon)$  is (an estimation of) the *probability* that a formula “like”  $F_\varepsilon$  is (un-)satisfiable;
- $\text{av\_steps}_{(U)SAT}^A(F_\varepsilon)$  is (an estimation of) the *average number of steps* it takes for  $\mathcal{A}$  to process a formula “like”  $F_\varepsilon$  *under the assumption* that  $F_\varepsilon$  is (un)satisfiable.

A branching  $v^{(\varepsilon)}$  is chosen with minimal  $h_{\mathcal{A}}(v^{(\varepsilon)})$ . The basic problem with the heuristic  $h_{\mathcal{A}}$  is to determine the meaning of “*like*”, and depending on this choice to determine the approximations of the probability of being satisfiable and the approximations of the average number of steps it takes to process an unsatisfiable or a satisfiable formula. The probabilities here are solver-independent, and we compute running times separately for unsatisfiable and satisfiable formulas (whatever the probability is that the instance is unsatisfiable resp. satisfiable).

I propose to choose “have the same *mixed density*” for “like” when speaking of the probability of being satisfiable, and to choose “have the same mixed density and the same number of variables” when speaking of the running times, where we now restrict our attention to clause-sets  $F$ , and the mixed density  $\rho(\mathbf{F})$  of  $F$  is the function with domain the different clause-sizes  $k$  occurring in  $F$ , mapping  $k \mapsto \rho(F)(k) = \rho_k(F) := \frac{c_k(F)}{n(F)}$ , using  $c_k(F)$  for the number of clauses in  $F$  of size  $k$ . We assume a “generalised threshold conjecture”, which asserts that for  $n$  going to infinity the probability of being satisfiable for clause-sets with fixed mixed density  $\varrho$  either goes to 0 or 1, except of the exceptional case when we are at a threshold (where the probability might be arbitrary). To determine the limit-probability  $\gamma(\varrho)$  for being unsatisfiable<sup>1)</sup> thus we have to determine the thresholds  $\Gamma(\varrho, k)$  for arbitrary mixed densities  $\varrho$  and clause-sizes  $k > \text{dom}(\varrho)$  (that is  $k > k'$  for all  $k' \in \text{dom}(\varrho)$ ), where we have  $\gamma(\varrho_{k,\varepsilon}^-) = 0$  and  $\gamma(\varrho_{k,\varepsilon}^+) = 1$  for  $\varepsilon > 0$ , using  $\varrho_{k,\varepsilon}^\pm$  for the mixed density obtained from  $\varrho$  by mapping additionally  $k$  to  $\Gamma(\varrho, k) \pm \varepsilon$ .

I want to conduct a (large-scale) computational experiment for determining these threshold functions, based upon a (large) database for mixed random clause-sets. For this study I want to rely on a precisely defined and strong random formula generator:

1. The range of parameters must be *known, reasonably large*, and the behaviour of the generator must be *stable* over this whole range.<sup>2)</sup>
2. In case we finally realise (after *many* hours of computation time) that the formulas produced by our (pseudo)-random generator are easier for some

---

<sup>1)</sup>rather than for being satisfiable;  $\gamma$  is non-decreasing, with values 0 for “small” densities and values 1 for “large” densities

<sup>2)</sup>Once I studied random clause-sets with low density but large number of variables, and I found quite interesting new effects there — until I found out that the generator I used could handle only up to 65535 variables!

algorithms than “real” random formulas, this should be something very interesting.

Using the usual linear congruence generators, when we finally will realise that these formulas are easy (and that’s for sure), this for my understanding would be just another anecdote on bad choices of random generators. Therefore I have chosen to write a random generator (“**OKgenerator**”, available from my home page) based on a *strong cryptographic standard*, namely the “Advanced Encryption Standard” (AES) — here we can hope that the generator is much better than the usual ones, and that in case we find some strange regularities then this should be of high relevance for AES and the field of cryptology in general!

For the implementation **OKgenerator** of this mathematically defined generator I considered two possible programming languages, C and C++, since for these two programming languages we have international standards (ISO/IEC 9899 for C, ISO/IEC 14882 for C++), good compilers on many platforms, and these languages allow efficient implementations. I have chosen C++ since in fact “C++ is the better C.” Given the parameters, the generated formula must be *reproducible* on any system having an ISO/IEC 14882 compliant C++ compiler. Since in general C++ programs are allowed to depend on the implementation (in order to allow extensions, which are necessary to fully exploit a great diversity of hardware), it must be guaranteed that for **OKgenerator** implementation defined behaviour has no effect on the output (except of abortion in case of missing resources), and undefined behaviour is not possible; given what is guaranteed on integer types, in fact we can establish this.<sup>3)</sup>

## A few notations

Let  $\mathcal{MCLS}(\mathcal{VA})$  be the set of all multi-clause-sets over the set  $\mathcal{VA}$  of variables (except of the elements of  $\mathcal{MCLS}$  all other sets in the paper are “real” sets). For  $F \in \mathcal{MCLS}$  let  $n(F)$  be the number of variables,  $c(F)$  the number of clauses (with multiplicities) and  $c_i(F)$  the number of clauses of size  $i$ . The (relative) density  $\rho_i(F) := \frac{c_i(F)}{n(F)}$  is defined when  $n(F) \neq 0$ . By  $\mathbb{N} = \{1, 2, 3, \dots\}$  we denote the set of positive integers, while  $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ .

---

<sup>3)</sup>Different to existing generators — changing the environment may change the behaviour for all formula generators I have seen yet.

## 2 Mixed random formulas

### 2.1 Mixed random clause-sets and their densities

Consider  $n \in \mathbb{N}_0$ ,  $p \in \mathbb{N}_0$ ,  $p \leq n$ , and  $c \in \mathbb{N}_0$ , and define

$$\mathcal{MCLS}(n, p, c) := \{ F \in \mathcal{MCLS}(\{1, \dots, n\}) : c(F) = c \wedge \forall C \in F : |C| = p \}$$

as the set of all multi-clause-sets with  $n$  variables, constant clause length  $p$  and  $c$  clauses. More generally, for  $n \in \mathbb{N}_0$ ,  $m \in \mathbb{N}$ ,  $p_1, \dots, p_m \in \mathbb{N}$ ,  $c_1, \dots, c_m \in \mathbb{N}_0$  and  $p_1 < \dots < p_m \leq n$  let

$$\begin{aligned} \mathcal{MCLS}(n, (p_1, \dots, p_m), (c_1, \dots, c_m)) := \\ \{ F \in \mathcal{MCLS}(\{1, \dots, n\}) \mid c(F) = \sum_{i=1}^m c_i \wedge \forall i \in \{1, \dots, m\} : c_{p_i}(F) = c_i \} \end{aligned}$$

be the set of all multi-clause-sets with  $m$  different clause-lengths  $p_i$  and  $c_i$  many clauses of length  $p_i$  for  $1 \leq i \leq m$ . Considering all elements of the set  $\mathcal{MCLS}(n, (p_1, \dots, p_m), (c_1, \dots, c_m))$  as having the same probability, we define the probability of the event of unsatisfiability in this (finite) probability space by

$$P_0(n, (p_1, \dots, p_m), (c_1, \dots, c_m)) := \frac{|\{F \in \mathcal{MCLS}(n, (p_1, \dots, p_m), (c_1, \dots, c_m)) : F \notin \text{SAT}\}|}{|\mathcal{MCLS}(n, (p_1, \dots, p_m), (c_1, \dots, c_m))|}.$$

Let  $r : \mathbb{R}_{\geq 0} \rightarrow \mathbb{N}_0$  be the rounding to the nearest integer, that is  $r(x) := \lfloor x \rfloor$  in case of  $x < \lfloor x \rfloor + \frac{1}{2}$  while otherwise we set  $r(x) := \lceil x \rceil$ .

**Strong Threshold Conjecture Part A** For all  $m \in \mathbb{N}$ ,  $p_1, \dots, p_m \in \mathbb{N}$  with  $p_1 < \dots < p_m$  and for all  $\varrho_1, \dots, \varrho_m \in \mathbb{R}_{\geq 0}$  the limit

$$\lim_{n \rightarrow \infty} P_0(n, (p_1, \dots, p_m), (r(\varrho_1 \cdot n), \dots, r(\varrho_m \cdot n)))$$

exists.

This part of the conjecture expresses the assumption, that satisfiability or unsatisfiability of a (mixed) random clause-set is determined (“with high probability”) for  $n$  large enough already by just knowing the (relative) densities.

Let  $\mathcal{D}$  be the set of all *mixed densities*, which we define as maps  $\varrho : P \rightarrow \mathbb{R}_{\geq 0}$  for finite  $P \subseteq \mathbb{N}$ . We define  $\gamma : \mathcal{D} \rightarrow [0, 1]$  by  $\gamma(\emptyset) := 0$ , while for  $\varrho : P \rightarrow \mathbb{R}_{\geq 0}$ ,  $P \neq \emptyset$ ,  $P = \{p_1, \dots, p_m\}$ ,  $p_1 < \dots < p_m$  let

$$\gamma(\varrho) := \lim_{n \rightarrow \infty} P_0(n, (p_1, \dots, p_m), (r(\varrho(1) \cdot n), \dots, r(\varrho(m) \cdot n))).$$

On  $\mathcal{D}$  we define a natural partial order  $\leq$  by

$$\varrho \leq \varrho' :\iff \text{dom}(\varrho) \subseteq \text{dom}(\varrho') \wedge \forall p \in \text{dom}(\varrho) : \varrho(p) \leq \varrho'(p),$$

while  $\varrho < \varrho'$  holds if  $\varrho \leq \varrho'$  and  $\varrho \neq \varrho'$ . For  $p \in \mathbb{N}$  and  $q \in \mathbb{R}_{\geq 0}$  we denote by  $\langle p \rightarrow q \rangle$  the map with domain  $\{p\}$  and  $p \mapsto q$ . The following properties hold:

1.  $\varrho \leq \varrho' \Rightarrow \gamma(\varrho) \leq \gamma(\varrho')$
2.  $\forall p \in \mathbb{N} \exists B \in \mathbb{R}_{>0} : \gamma(\langle p \rightarrow B \rangle) = 1$
3.  $\forall \varrho \in \mathcal{D} \exists \varepsilon \in \mathbb{R}_{>0} : \gamma(\varepsilon \cdot \varrho) = 0.$

Considering  $\gamma$  only as a partial function, these three properties in fact can be proven without relying on any assumption.

## 2.2 Example: The $(2 + \lambda)$ -model

In [8, 9] the following case of mixed random clause-sets has been studied (by means of statistical physics). For fixed (but arbitrary)  $\lambda \in [0, 1]$  the authors argued that the function

$$q \in \mathbb{R}_{\geq 0} \mapsto f_\lambda(q) := q \cdot \langle 2 \rightarrow 1 - \lambda, 3 \rightarrow \lambda \rangle \in \mathcal{D},$$

shows a phase transition w.r.t. satisfiability, that is in our terminology, the function  $q \mapsto \gamma(f_\lambda(q))$  jumps at a certain threshold value  $q_\lambda$  from being constant 0 to being constant 1. We know (using the threshold for 2-SAT), that for  $\lambda \neq 1$  and  $q > \frac{1}{1-\lambda}$  we have  $\gamma(f_\lambda(q)) = 1$ . The above authors now claimed furthermore, that for  $\lambda < \lambda_c = 0.413\dots$  in fact we have  $q_\lambda = \frac{1}{1-\lambda}$ . This assertion has been proven for  $\lambda \leq \lambda'_c = \frac{2}{5}$  in [2] (and it has been argued, that this value for  $\lambda'_c$  is in fact the right one). It follows, that for all  $\varepsilon > 0$  and  $\mu \leq \frac{\lambda'_c}{1-\lambda'_c} = \frac{2}{3}$  we have

$$\gamma(\langle 2 \rightarrow 1 - \varepsilon, 3 \rightarrow \mu \rangle) = 0.$$

This shows, that under certain circumstances there is no fixed relation of the form (for example) “six ternary clauses are as good as one binary clause”, but we may need arbitrarily many ternary clauses to replace a binary one.

## 2.3 Guaranteeing the existence of thresholds in a general setting

By the following addition to Part A of our conjecture we obtain the existence of thresholds under very general conditions.

**Strong Threshold Conjecture Part B** For all  $\varrho, \varrho' \in \mathcal{D}$  with  $\varrho < \varrho'$  and  $0 < \gamma(\varrho) < 1$  we have  $\gamma(\varrho') = 1$ .

An equivalent formulation is that for all  $\varrho, \varrho' \in \mathcal{D}$  with  $\varrho' < \varrho$  and  $0 < \gamma(\varrho) < 1$  we have  $\gamma(\varrho') = 0$ , and we see that that this part of the conjecture generalises Corollary 1 in [2]. This part of our conjecture in fact seems to be provable to me by standard means. Some sort of underlying intuition is, that at a density  $\varrho$  where the limit-probability of being satisfiable is neither 0 or 1, any addition of a fraction of clauses (of any fixed size) must destroy this very unstable situation.

It follows the existence of “threshold functions” in the following sense. Consider  $\varrho \in \mathcal{D}$  and  $p \in \mathbb{N} \setminus \text{dom}(\varrho)$ . Then there is exactly one  $\Gamma(\varrho, p) \in \mathbb{R}_{\geq 0}$ , such that for all  $x \in \mathbb{R}_{\geq 0}$

$$\begin{aligned} x < \Gamma(\varrho, p) &\Rightarrow \gamma(\varrho \cup \langle p \rightarrow x \rangle) = 0 \\ x > \Gamma(\varrho, p) &\Rightarrow \gamma(\varrho \cup \langle p \rightarrow x \rangle) = 1 \end{aligned}$$

holds. We have the following general properties:

1.  $\varrho \leq \varrho' \Rightarrow \Gamma(\varrho, p) \geq \Gamma(\varrho', p)$
2.  $p \leq p' \Rightarrow \Gamma(\varrho, p) \leq \Gamma(\varrho, p')$ .

We set  $\Gamma(p) := \Gamma(\emptyset, p)$ , obtaining the “usual” threshold for random  $p$ -SAT. The following values are known:

1.  $\Gamma(2) = 1$  ([7]); according to [14] it is not known, whether  $\gamma(\langle 2 \mapsto 1 \rangle)$  in fact exists, but computational experiments might suggest  $\gamma(\langle 2 \mapsto 1 \rangle) \approx 0.1$ . By the conjecture part (B) for any  $p > 2$  we have  $\Gamma(\langle 2 \mapsto 1 \rangle, p) = 0$ .
2. By [2], Theorem 3 for all  $0 < \varepsilon < 1$  we get  $\gamma(\langle 2 \mapsto 1 - \varepsilon, 3 \mapsto \frac{2}{3} \rangle) < 1$  (with  $\frac{2}{3} = \frac{\frac{2}{3}}{1 - \frac{2}{3}}$ ), while Theorem 4 yields  $\gamma(\langle 2 \mapsto 1 - \varepsilon, 3 \mapsto 2.28 \rangle) = 1$ , and thus from our conjecture  $\Gamma(\langle 2 \mapsto 1 - \varepsilon \rangle, 3) \in [\frac{2}{3}, 2.28]$  follows.
3. By no. 2 we have  $\Gamma(\langle 3 \mapsto \frac{2}{3} \rangle, 2) = 1$ . It follows that for all  $0 \leq \alpha \leq \frac{2}{3}$  we have  $\Gamma(\langle 3 \mapsto \alpha \rangle, 2) = 1$ .

With  $\Gamma : \mathcal{D} \times \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$  we have fixed a notation for thresholds in the common situation, where all densities are fixed except for one clause-length (the second argument of  $\Gamma$ ), but it does not capture for example the assertion in [8, 9] (see the previous subsection), that for all real numbers  $\lambda \in [0, 1]$  the functions  $q \in \mathbb{R}_{\geq 0} \mapsto \gamma(\langle 2 \mapsto (1 - \lambda) \cdot q, 3 \mapsto \lambda \cdot q \rangle)$  show a threshold behaviour (in [2], Theorem 2, the existence of a “weak threshold” has been shown, depending on the number of variables). Now it is in fact easily seen, that from our conjecture (parts A and B) the existence of thresholds follow under very general conditions:

**Lemma 2.1** Consider any map  $f : \mathbb{R}_{\geq 0} \rightarrow \mathcal{D}$  which is strictly increasing (that is, for all  $x, x' \in \mathbb{R}_{\geq 0}$  with  $x < x'$  we have  $f(x) < f(x')$ ), such that  $\gamma(f(0)) = 0$  and there is  $x \in \mathbb{R}_{> 0}$  with  $\gamma(f(x)) > 0$ . Then there is exactly one “threshold”  $t \in \mathbb{R}_{\geq 0}$ , such that for all  $x \in \mathbb{R}_{\geq 0}$  the following holds:

$$x < t \Rightarrow \gamma(f(x)) = 0, \quad x > t \Rightarrow \gamma(f(x)) = 1.$$

### 3 A generic branching rule

Let  $\mathcal{MCLS}'$  be the set of  $F \in \mathcal{MCLS}$  with  $\perp \notin F$ .

For  $F \in \mathcal{MCLS}'$  let  $\rho(F) \in \mathcal{D}$  be the map with domain  $\{|C| : C \in F\}$ , which maps  $p \mapsto \rho_p(F) = \frac{c_p(F)}{n(F)}$ . Assume an “approximation”  $\tilde{\gamma} : \mathcal{D} \rightarrow [0, 1]$  of  $\gamma : \mathcal{D} \rightarrow [0, 1]$  is given.

Let the “specified density”  $\hat{\rho}(F)$  be the pair  $(n(F), \rho(F)) \in \mathcal{S}$ , where  $\mathcal{S}$  is the set of all pairs  $(n, \varrho) \in \mathbb{N}_0 \times \mathcal{D}$  such that for all  $p \in \text{dom}(\varrho)$  we have  $p \leq n$ . Consider a SAT solver  $\mathcal{A} : \mathcal{MCLS}' \rightarrow \{0, 1\}$ . We assume functions

$$\mu_0, \mu_1 : \mathcal{S} \rightarrow \mathbb{R}_{\geq 1},$$

where  $\mu_\varepsilon(n, \varrho)$  for  $\varrho : \{p_1, \dots, p_m\} \rightarrow \mathbb{R}_{\geq 0}$  ( $m \in \mathbb{N}_0$ ) approximates the average number of “leaves” for a run of  $\mathcal{A}$  on unsatisfiable ( $\varepsilon = 0$ ) resp. satisfiable ( $\varepsilon = 1$ ) clause-set  $F \in \mathcal{MCLS}(n, (p_1, \dots, p_m), (r(\varrho(p_1)) \cdot n), \dots, (r(\varrho(p_m)) \cdot n))$ .

We split the expected “mathematical running time” of the solver  $\mathcal{A}$  into functions  $\mu_0, \mu_1$  handling unsatisfiability and satisfiability *separately*, since experiments suggest that the behaviour of (DPLL)-SAT solvers is quite different on unsatisfiable and on satisfiable instances, and in general treating satisfiability resp. unsatisfiability requires different algorithmic “resources”. Lower bounds for resolution refutations of random formulas are available, typically independent of the densities, although the densities are mentioned in order to guarantee, that there are enough unsatisfiable examples; see for example [1, 4].

Consider  $F \in \mathcal{MCLS}$ . We want to choose a branching out of the  $2 \cdot n(F)$  possible elementary branchings  $v^{(0)}, v^{(1)}$ , where  $v^{(0)}$  stands for the branching “first  $\langle v \rightarrow 0 \rangle$ , then  $\langle v \rightarrow 1 \rangle$ ”, and  $v^{(1)}$  stands for the branching “first  $\langle v \rightarrow 1 \rangle$ , then  $\langle v \rightarrow 0 \rangle$ ”. We define the “heuristic complexity” of  $v^{(\varepsilon)}$  as (using  $F_\varepsilon := \langle v \rightarrow \varepsilon \rangle * F$ )

$$\begin{aligned} \mathbf{h}(v^{(\varepsilon)}) := & (1 - \tilde{\gamma}(\rho(F_\varepsilon))) \cdot \mu_1(\hat{\rho}(F_\varepsilon)) + \\ & \tilde{\gamma}(\rho(F_\varepsilon)) \cdot (\mu_0(\hat{\rho}(F_\varepsilon)) + \\ & (1 - \tilde{\gamma}(\rho(F_{1-\varepsilon}))) \cdot \mu_1(\hat{\rho}(F_{1-\varepsilon})) + \tilde{\gamma}(\rho(F_{1-\varepsilon})) \cdot \mu_0(\hat{\rho}(F_{1-\varepsilon}))). \end{aligned}$$

The motivation for this heuristics should be rather obvious:

1. With approximate probability  $1 - \tilde{\gamma}(\rho(F_\varepsilon))$  the first branch  $v \rightarrow \varepsilon$  yields a satisfiable clause-set  $F_\varepsilon$ , and we will need  $\mu_1(\hat{\rho}(F_\varepsilon))$  steps for this to find out.
2. With approximate probability  $\tilde{\gamma}(\rho(F_\varepsilon))$  the first branch  $v \rightarrow \varepsilon$  yields an unsatisfiable clause-set, and then we have to evaluate  $F_{1-\varepsilon}$ .

Now choose the branching  $v^{(\varepsilon)}$  with minimal  $h(v^{(\varepsilon)})$ . For the computation of  $\tilde{\gamma}$  (solver independent) and of  $\mu_0, \mu_1$  (solver dependent) at this time the most successful approach seems to be to exploit a (large) database for random formula and to extract procedures for computing  $\tilde{\gamma}$  and  $\mu_0, \mu_1$  by multi-dimensional curve fitting.

### **Needed: A large database for (mixed) random formulas**

I plan to build up a (rather large) database (using PostgreSQL) for mixed random formulas which shall contain information on (individual) “random” formulas from  $\mathcal{MCLS}(n, (p_1, \dots, p_m), (c_1, \dots, c_m))$  for a variety of choices for the parameters. For each formula it is stored whether the formula is satisfiable or not, and, if available, then also further information is supplied about the solver who solved it, its running time, the computer used, and so on. This database (let’s call it “OKdatabase”) shall be accessible via the Internet, and I also want to build up a structure so that trusted sources can add data to this database (investigating “grid” technology).

So approximations for  $\gamma(\varrho)$  will be available for some finite set of  $\varrho \in \mathcal{D}$ , and (hopefully) these approximations can be integrated into a reasonable way for computing  $\tilde{\gamma}(\varrho)$  over a large range of mixed densities  $\varrho$ , using the “compression” of the data set obtained by the strong threshold conjecture. And by supplying further information on running times and tree sizes, we will try to figure out how to compute  $\mu_0(\hat{\rho}), \mu_1(\hat{\rho})$  for  $\hat{\rho} \in \mathcal{S}$  (for some given solvers (at least for OKsolver)).

Storing a large number of “real” random formulas is out of scope, but instead a pseudo-random generator shall be used (so we have to store only the respective parameter values), which shall be a good source for (pseudo)-randomness over a large parameter space, which shall be precisely defined, and where an efficient implementation is available, implementing the mathematical definition in a completely platform-independent manner. Such a generator, “OKgenerator” (written in standard C++) is the subject of the final section.

## 4 The random formulas generator OKgenerator, based on AES

The subject of this section is the precise specification of the random formula generator “OKgenerator” ([11]). The input of this procedure is as follows:

1. A *seed* (or key)  $s \in \{0, 2^{64} - 1\}$ .
2. A *formula number*  $k \in \{0, 2^{64} - 1\}$ .
3. The *number of variables*  $n \in \{1, \dots, 2^{31} - 1\}$ .
4. The *number*  $m \in \{1, \dots, 2^{31} - 1\}$  of *different clause-sizes*.
5. A list  $p_1 < \dots < p_m \leq n$  of *clause-sizes* with  $p_i \in \{1, \dots, 2^{31} - 1\}$ .
6. A list  $c_1, \dots, c_m$  of *numbers of clauses of size*  $p_i$  with  $c_i \in \{1, \dots, 2^{32} - 1\}$ .

The output is an element of  $\mathcal{MCLS}(n, (p_1, \dots, p_m), (c_1, \dots, c_m))$ , which is *intended* to be a random element (of course a “pseudo-random” element, since OKgenerator is deterministic), where all elements have the same probability. (In fact the output is a *sequence* (of clauses), and thus to obtain multi-clause-sets one had to consider two outputs as equal if they only differ in the order of clauses.) The parameters shall play the following roles (over the *whole range* of possible values):

1.  $\text{OKgenerator}(s, k, n, (p_1, \dots, p_m), (c_1, \dots, c_m))$  is the concatenation of the constant clause-length formulas  $\text{OKgenerator}(s, k, n, p_i, c_i)$ ,  $i = 1, \dots, m$ .
2. The formulas  $\text{OKgenerator}(s, k, n, p, c)$  and  $\text{OKgenerator}(s', k', n', p', c')$  shall be “completely unrelated” as soon as  $(s, k, n, p) \neq (s', k', n', p')$ .
3.  $\text{OKgenerator}(s, k, n, p, c)$  is a prefix of  $\text{OKgenerator}(s, k, n, p, c')$  in case of  $c \leq c'$ .

The usage of OKgenerator may be guided as follows:

1. For systematic exploration of random clause-sets the seed is set to  $s := 0$  — only for occasions like competitions it is useful to randomly select the seed (while the other settings are predetermined) in order to get a reproducible set of random formulas with predetermined properties.
2. The formula numbers takes consecutive values  $k = 0, 1, \dots$  to obtain a (long) sequence of random formulas for fixed  $n$ ,  $m$ ,  $(p_1, \dots, p_m)$  and  $(c_1, \dots, c_m)$ .

3. Consider two sequences  $F_0, F_1, \dots$  and  $F'_0, F'_1, \dots$  of random formulas, the first with respect to  $s = 0$  and fixed  $n, m, (p_1, \dots, p_m), (c_1, \dots, c_m)$  where  $k = 0, 1, \dots$ , while the second sequence has parameters  $s = 0$  and fixed  $n', m', (p'_1, \dots, p'_m), (c'_1, \dots, c'_m)$  and again  $k = 0, 1, \dots$ . Now these sequence are completely unrelated except of the following:

For all  $p_u = p'_v$ , in each formula  $F_i$  the block of clauses of length  $p_u$  is a prefix of the corresponding block of  $F'_i$  in case of  $c_u \leq c'_v$  (and vice versa).

If this correlation is not wished, then one should use disjoint ranges for the formula number  $k$ .<sup>4)</sup>

For  $k \in \mathbb{N}$  let  $\mathcal{W}_k := \{0, \dots, 2^k - 1\}$  be the set of natural numbers from 0 to  $2^k - 1$ , corresponding to the set of unsigned  $k$ -bit integers. The “random source” for our generator is given by the function

$$\mathbf{aes} : \mathcal{W}_{128} \times \mathcal{W}_{128} \rightarrow \mathcal{W}_{128},$$

derived from the block cipher AES :  $\{0, 1\}^{128} \times \{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$  as defined in [5] (see also [13], Chapter 19) by considering sequences of bits as binary numbers (leading bits first)<sup>5)</sup>. See Appendix A for more details.

Based on  $\mathbf{aes}$ , we derive the literal generator

$$\mathbf{lg} : \mathcal{W}_{64} \times \mathcal{W}_{64} \times (\mathcal{W}_{31} \setminus \{0\}) \times \mathcal{W}_{31} \times \mathcal{W}_{64} \rightarrow (-\mathcal{W}_{31} \cup \mathcal{W}_{31}) \setminus \{0\}$$

in the following way, where “variables” are given by the elements of  $\mathcal{W}_{31} \setminus \{0\}$ , and the “literal”  $\mathbf{lg}(s, k, n, p, i)$  uses the arithmetical sign as polarity, where  $s$  is a 64-bit key (or seed),  $k$  is a 64-bit formula number,  $n$  is the number of variables (31 bit),  $p$  is the clause-length (31 bit) and  $i$  is a 64-bit literal number.

For  $a, b \in \mathbb{N}_0$ ,  $b \neq 0$  let  $a \bmod b$  be the unique number  $r$  with  $0 \leq r < b$  such that there is  $q \in \mathbb{N}_0$  with  $a = q \cdot b + r$ . We need the bijection  $\alpha_n : \{0, \dots, 2n - 1\} \rightarrow \{-n, \dots, -1\} \cup \{1, \dots, n\}$  for  $n \in \mathbb{N}$  defined by

$$\alpha_n(k) := \begin{cases} k + 1 & \text{if } k \leq n - 1 \\ -(k - (n - 1)) & \text{if } k \geq n \end{cases}.$$

Now

$$\mathbf{lg}(s, k, n, p, i) := \alpha_n(\mathbf{aes}(s \cdot 2^{64} + k, n \cdot 2^{96} + p \cdot 2^{64} + i) \bmod 2n).$$

<sup>4)</sup>Choosing a different seed would yield completely unrelated formulas without any dependencies w.r.t. the formula number, however in order to enable efficient use of the database of random formulas (generated by **OKgenerator**) only formulas with  $s = 0$  are stored, and I propose to restrict the use of seeds other than 0 to special occasions as discussed above.

<sup>5)</sup>the first argument for  $\mathbf{aes}$  and AES is the key, the second argument is the input block (the “plain text”), and the return value is the encrypted input block (the “cipher text”)

Based on the literal generator<sup>6)</sup>, we first consider the creation of random constant clause-length formulas. Consider  $s \in \mathcal{W}_{64}$ ,  $k \in \mathcal{W}_{64}$ ,  $n \in \mathcal{W}_{31} \setminus \{0\}$ ,  $p \in \mathcal{W}_{31} \setminus \{0\}$ ,  $p \leq n$  and  $c \in \mathcal{W}_{32} \setminus \{0\}$ , and let

$$\mathbf{cnfg}(s, k, n, p, c) := (C_1, \dots, C_c),$$

where the clauses  $C_i = (l_{i,1}, \dots, l_{i,p})$  (which are in fact (ordered) tuples) are defined as follows.

For  $A \subseteq \mathbb{N}$  and  $m \in \mathbb{N}$ ,  $m \leq |A|$  let  $(A)_m$  be the  $m$ -th element of  $A$  in the natural order. For  $1 \leq i \leq c$  and  $1 \leq j \leq p$  let

$$x_{i,j} := \text{lg}(s, k, n - j + 1, p, (i - 1) \cdot p + j - 1)$$

and let  $l_{i,j} \in (-\mathcal{W}_{31} \cup \mathcal{W}_{31}) \setminus \{0\}$  be determined by  $\text{sgn}(l_{i,j}) = \text{sgn}(x_{i,j})$  and

$$\begin{aligned} |l_{i,1}| &= |x_{i,1}|, \\ |l_{i,j}| &= (\{1, \dots, n\} \setminus \{|l_{i,1}|, \dots, |l_{i,j-1}|\})_{|x_{i,j}|} \text{ for } j > 1. \end{aligned}$$

It follows from this definition that to compute one single literal from a generated clause-set with constant clause-length  $p$  we need at most  $p$  many calls to the AES “oracle”.

In order to handle mixed clause-length, let “\*” denote concatenation of tuples, i.e.  $(a_1, \dots, a_m) * (b_1, \dots, b_n) = (a_1, \dots, a_m, b_1, \dots, b_n)$ . Now for  $s \in \mathcal{W}_{64}$ ,  $k \in \mathcal{W}_{64}$ ,  $n \in \mathcal{W}_{31} \setminus \{0\}$ ,  $m \geq 1$ , and  $p_i \in \mathcal{W}_{31} \setminus \{0\}$ ,  $c_i \in \mathcal{W}_{32} \setminus \{0\}$  for  $1 \leq i \leq m$ , where  $p_1 < \dots < p_m \leq n$ , let

$$\mathbf{mcnfg}(s, k, n; (p_1, \dots, p_m), (c_1, \dots, c_m)) := \bigstar_{i=1}^m \mathbf{cnfg}(s, k, n, p_i, c_i).$$

The formula generator `OKgenerator` is an implementation of the function `mcnfg`. For the natural generalisation of `mcnfg` to the creation of generalised clause-sets (which are “conjunctive normal forms” for *constraint satisfaction problems*) see Appendix B.

## Acknowledgments

I want to thank Jeremy Avigad and Charlie Reckhow for helpful discussions on the subject of random number generators and cryptology.

<sup>6)</sup>the reader might wonder, why I didn't choose the somewhat simpler generation of a literal by first calculating the variable from the aes-result modulo  $n$ , and then calculating the sign from the aes-result modulo 2: in this way the sign would depend only on the last bit of the cipher text, and although this should be safe, it simply seemed unnatural to me

## References

- [1] Dimitris Achlioptas, Paul Beame, and Michael Molloy. A sharp threshold in proof complexity. In *STOC'01*, Crete, Greece, July 2001.
- [2] Dimitris Achlioptas, Lefteris M. Kirousis, Evangelos Kranakis, and Danny Krizanc. Rigorous results for random  $(2 + p)$ -SAT. *Theoretical Computer Science*, 265:109–129, 2001.
- [3] Paul Beame, Richard Karp, Toniann Pitassi, and Michael Saks. On the complexity of unsatisfiability proofs for random  $k$ -CNF formulas. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 561–571, May 1998.
- [4] Paul Beame, Richard Karp, Toniann Pitassi, and Michael Saks. The efficiency of resolution and Davis-Putnam procedures. Submitted. Journal version of [3], May 1999.
- [5] Joan Daemen and Vincent Rijmen. *AES Proposal: Rijndael*, September 1999. Document Version 2.
- [6] Brian Gladman. Implementation of the advanced encryption standard (AES). Can be downloaded from the Internet site [http://fp.gladman.plus.com/cryptography\\_technology/rijndael/index.htm](http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm), February 2002.
- [7] Andreas Goerdt. A threshold for unsatisfiability. *Journal of Computer and System Sciences*, 53:469–486, 1996.
- [8] Scott Kirkpatrick, Rémi Monasson, Bart Selman, Lidror Troyansky, and Ricardo Zecchina. Phase transition and search cost in the  $(2 + p)$ -SAT model. In *4th Workshop in Physics and Computation*, Boston, MA, 1996.
- [9] Scott Kirkpatrick, Rémi Monasson, Bart Selman, Lidror Troyansky, and Ricardo Zecchina. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, July 1999.
- [10] Oliver Kullmann. Upper and lower bounds on the complexity of generalized resolution and constraint satisfaction problems. Submitted to *Annals of Mathematics and Artificial Intelligence*, September 2000.
- [11] Oliver Kullmann. The implementation OKgenerator. Available at <http://cs-svr1.swan.ac.uk/~csoliver/>, February 2002. Version 1.2, Release 1.0.
- [12] Oliver Kullmann. Towards an adaptive density based branching rule for SAT solvers, using a database for mixed random conjunctive normal forms built upon the advanced encryption standard (AES). Submitted to SAT'2002 (Cincinnati), February 2002.

- [13] Michael Welschenbach. *Cryptography in C and C++*. Apress (Springer-Verlag), 2001.
- [14] David Wilson, February 2002. Personal Communication.

## A AES — from bits to numbers

To avoid any vagueness, I want to make explicit how the function

$$\text{aes} : \mathcal{W}_{128} \times \mathcal{W}_{128} \rightarrow \mathcal{W}_{128}$$

is based on the block cipher AES as described in [5]. We consider encryption only, and the block length and the key length are both 128 bits. Thus we have given the block cipher  $\text{AES} : \{0, 1\}^{128} \times \{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$ , where the first argument is the key, the second argument the input block (the “plain text” in ECB mode), and the return value is the encrypted input block (the “cipher text” in ECB mode). We interpret these 128-bit arguments as natural numbers from 0 to  $2^{128} - 1$  in “Big Endian” notation, that is the first bit is the high order bit, or, explicitly, a block  $x \in \{0, 1\}^{128}$  corresponds to the number  $\sum_{i=1}^{128} x_i \cdot 2^{128-i}$ .

Typically, in implementations<sup>7)</sup> input, output and key are one-dimensional arrays of 8-bit bytes, each of dimension 16, where the array indices range from 0 . . . 15. Thus now a 128-bit block  $x$  corresponds to a vector  $x = (x_0, \dots, x_{15}) \in \{0, \dots, 255\}^{16}$ , and the corresponding number in  $\mathcal{W}_{128}$  is given by  $\sum_{i=0}^{15} x_i \cdot 256^{15-i}$ .

Here are two examples, both first expressed in block notation, denoting bytes by 2-digit hexadecimal numbers (each block consists of 16 bytes) and then using natural numbers (in decimal notation):

$$\begin{aligned} \text{AES}(00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00) = \\ 66\ e9\ 4b\ d4\ ef\ 8a\ 2c\ 3b\ 88\ 4c\ fa\ 59\ ca\ 34\ 2b\ 2e \end{aligned}$$

$$\text{aes}(0, 0) = 136792598789324718765670228683992083246$$

$$\begin{aligned} \text{AES}(00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 05\ 00\ 80\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00) = \\ 32\ 41\ e3\ d4\ 99\ c7\ 3d\ 2f\ 38\ 73\ 73\ 1b\ 16\ d9\ e0\ 60 \end{aligned}$$

$$\text{aes}(5 * 256^1, 128 * 256^{15}) = 66803520035993111070459895814771302496$$

---

<sup>7)</sup>I use the implementation [6] by Brian Gladman

## B Generalised clause-sets

Let  $d \in \mathbb{N}$  be the *domain size*. We consider generalised clause-sets as studied in [10] (with uniform domains  $D_v = \{0, \dots, d-1\}$ ), where literals are pairs  $(v, \varepsilon)$ ,  $v$  a variable,  $\varepsilon \in \{0, \dots, d-1\}$ , clauses are finite sets  $C$  of literals such that no literals  $(v, \varepsilon), (v, \varepsilon') \in C$  with  $\varepsilon \neq \varepsilon'$  exist, and clause-sets are finite sets of clauses.<sup>8)</sup>

Based on aes, we derive the literal generator

$$\mathbf{lg}^* : \mathcal{W}_{64} \times \mathcal{W}_{32} \times (\mathcal{W}_{32} \setminus \{0\}) \times (\mathcal{W}_{32} \setminus \{0, 1\}) \times (\mathcal{W}_{32} \setminus \{0\}) \times \mathcal{W}_{64} \rightarrow (\mathcal{W}_{32} \setminus \{0\}) \times \mathcal{W}_{32},$$

where the literal  $\mathbf{lg}^*(s, k, n, d, p, i)$  depends on the following parameters:

1.  $s$  is a 64-bit key (or seed);
2.  $k$  is a 32-bit formula number;
3.  $n \geq 1$  is the number of variables (32 bit);
4.  $d \geq 2$  is the domain size (32 bit);
5.  $p \geq 1$  is the clause-length (32 bit);
6.  $i$  is a 64-bit literal number.

For  $a, b \in \mathbb{N}_0$ ,  $b \neq 0$  there is exactly one  $(q, n) \in \mathbb{N}_0^2$  with  $0 \leq r < b$  and  $a = q \cdot b + r$ , and we set  $a \bmod b := r$ ,  $a \operatorname{div} b := q$ . Now

$$\begin{aligned} \mathbf{lg}^*(s, k, n, d, p, i) &:= (v, \varepsilon) \\ v &:= ((A \bmod (n \cdot d)) \bmod n) + 1 = (A \bmod n) + 1 \\ \varepsilon &:= (A \bmod (n \cdot d)) \operatorname{div} n \\ A &:= \text{aes}(s \cdot 2^{64} + (d-2) \cdot 2^{32} + k, n \cdot 2^{96} + p \cdot 2^{64} + i). \end{aligned}$$

Consider  $s \in \mathcal{W}_{64}$ ,  $k \in \mathcal{W}_{32}$ ,  $n \in \mathcal{W}_{32} \setminus \{0\}$ ,  $d \in \mathcal{W}_{32} \setminus \{0, 1\}$ ,  $p \in \mathcal{W}_{32} \setminus \{0\}$ ,  $p \leq n$  and  $c \in \mathcal{W}_{32} \setminus \{0\}$ , and let

$$\mathbf{gcfnfg}(s, k, n, d, p, c) := (C_1, \dots, C_c),$$

where the clauses  $C_i = (l_{i,1}, \dots, l_{i,p})$  (which are in fact (ordered) tuples) are defined as follows. For  $1 \leq i \leq c$  and  $1 \leq j \leq p$  let

$$(x_{i,j}, \varepsilon_{i,j}) := \mathbf{lg}^*(s, k, n-j+1, d, p, (i-1) \cdot p + j - 1).$$

---

<sup>8)</sup>The meaning of literal  $(v, \varepsilon)$  is that  $v$  shall *not* get value  $\varepsilon$ .

Furthermore let  $v_{i,j}$  be defined by

$$v_{i,1} = x_{i,1},$$

$$v_{i,j} = (\{1, \dots, n\} \setminus \{v_{i,1}, \dots, v_{i,j-1}\})_{x_j} \text{ for } j > 1.$$

and set  $l_{i,j} := (v_{i,j}, \varepsilon_{i,j})$ .

Finally for  $s \in \mathcal{W}_{64}$ ,  $k \in \mathcal{W}_{32}$ ,  $n \in \mathcal{W}_{32} \setminus \{0, 1\}$ ,  $d \in \mathcal{W}_{32} \setminus \{0, 1\}$ ,  $m \geq 1$ , and  $p_i \in \mathcal{W}_{32} \setminus \{0\}$ ,  $c_i \in \mathcal{W}_{32} \setminus \{0\}$  for  $1 \leq i \leq m$ , where  $p_1 < \dots < p_m \leq n$ , let

$$\mathbf{mgcnfg}(s, k, n, d; (p_1, \dots, p_m), (c_1, \dots, c_m)) := \prod_{i=1}^m \mathbf{gcnfg}(s, k, n, p_i, c_i).$$

For  $n \leq 2^{31} - 1$  we have

$$\mathbf{mgcnfg}(s, k, n, 2; (p_1, \dots, p_m), (c_1, \dots, c_m)) =$$

$$\mathbf{mcnfg}(s, k, n; (p_1, \dots, p_m), (c_1, \dots, c_m)),$$

when identifying literals  $(v, 0)$  with  $v$  and  $(v, 1)$  with  $\bar{v}$ . `OKgenerator` implements also `mgcnfg`.

## C The implementation

The C++ program `OKgenerator` is designed as a UNIX tool, reading parameters from the command line, and printing the output to standard output. It uses the C++ package “LINT” for precise integer arithmetic from [13] and the AES implementation [6] by Brian Gladman.

The list of argument for `OKgenerator` is processed from left to right, and each argument causes some action. Help is available via

```
OKgenerator -h
```

Typical examples are:

```
OKgenerator n=300 l=3 cp=1275 -o
OKgenerator n=300 l=3 cp=300 l=2 cp=250 -o
OKgenerator n=300 l=3 dp=2/5 l=2 dp=8/10 -o
OKgenerator n=300 l=3 dp=0.4 l=2 dp=0.8 ds=4 -g -o
```

The first command produces a clause-set with 300 variables, clause-length 3, and 1275 clauses (“-o” means “output”). In the second example we have 300 clauses of size 3 and additionally 250 clauses of length 2 (the meaning of “cp” is “set clause-number and push current clause-size and clause-number on the stack”). In the third example one sees that the number of clauses can be given also by a density (there are  $\frac{2}{5} \cdot 300$  clauses of length 3 and  $\frac{8}{10} \cdot 300$  clauses of size 2; all computations are exact, and results are (correctly) rounded to the nearest integer). Finally with the last example we set the domain size to 4 (while “-g” switches to generalised clause-sets).

## C.1 Standardised densities

Due to rounding, the same formula may be produced by many different densities, which would obscure the use of the database. Therefore we define the *standardised density*  $\bar{\rho}_p(F)$  as follows (for  $n(F) \neq 0$ ):

Let  $a := \lfloor \rho_p(F) \rfloor$ ,  $b := \rho_p(F) - a$ , and consider the decimal expansion  $(d_i)_{i \in \mathbb{N}}$  of  $b$  (that is,  $d_i \in \{0, \dots, 9\}$  and  $\sum_{i=1}^{\infty} d_i \cdot 10^{-i} = b$ ) such that there is no  $n \in \mathbb{N}$  with  $d_i = 9$  for all  $i \geq n$  (so that the decimal expansion is unique). Let  $k \in \mathbb{N}_0$  be minimal with the property that considering only the first  $k$  digits in the decimal expansion of  $b$  is sufficient to reproduce the given number of clauses, i.e.

$$r\left(\left(a + \sum_{i=1}^k d_i \cdot 10^{-i}\right) \cdot n(F)\right) = c_p(F).$$

Now  $\bar{\rho}_p(F) := a + \sum_{i=1}^k d_i \cdot 10^{-i}$ .

A random clause-set generated by `OKgenerator` is stored in the database via formula number, number of variables, the list of different clause-sizes, and for each clause-size the standardised density. To facilitate the use of standardised densities, `OKgenerator` outputs in the comment part of the generated formula besides the basic parameters also the standardised densities, for example (using the Dimacs format, which is activated by the option “-D”):

```
> OKgenerator n=15 l=3 dp=0.82 l=4 dp=1/4 -D -o
c OKRandGen( (0, 0), (0, 0), 15; 3, 12; 4, 4; )
c Standardised density: 3 -> .8, 4 -> .26
p cnf 15 16
-13 -14 7 0
7 -10 2 0
5 -2 13 0
-11 4 -1 0
9 -5 -14 0
5 -13 -12 0
-10 -4 2 0
10 -8 -14 0
-3 -5 -1 0
12 1 -8 0
14 7 10 0
5 -7 -11 0
15 -12 10 7 0
9 -12 -7 -2 0
-5 -14 1 8 0
-15 12 8 11 0
```

The first parameter is the 64-bit key as a pair of 32-bit numbers, the second parameter is the 64-bit formula number in the same presentation (with high-order bits first), while the third parameter is the number of variables; it follows the list of pairs of clause-sizes and clause-numbers.

To create the next random formula in the sequence (with formula number 1, while the above formula has formula number 0) set the low-order part of the formula number via the action “nr1=”:

```
> OKgenerator n=15 l=3 dp=0.82 l=4 dp=1/4 nr1=1 -o
% OKgenerator( (0, 0), (0, 1), 15; 3, 12; 4, 4; )
% Standardised density: 3 -> .8, 4 -> .26
(11,-12,-15)
(5,-10,-3)
(-6,-11,8)
(3,9,-7)
(11,-8,3)
(5,12,6)
(13,7,-5)
(-11,6,12)
(-4,6,-15)
(8,2,-5)
(-12,7,-5)
(-9,-10,8)
(-9,13,14,-15)
(-13,1,-12,15)
(-10,12,9,-13)
(7,9,-14,-12)
```

## C.2 Other uses of the generator

In the previous example we didn’t use the Dimacs format for the output, but the default format, which can be altered in many ways to facilitate the use of `OKgenerator` for other purposes than for creating random clause-sets. For example with

```
> OKgenerator n=1 l=1 cp=20 cb="" ce=" " -nc -o -nl
-1 -1 -1 -1 -1 1 1 -1 1 -1 1 1 -1 1 -1 1 1 -1 -1 -1
```

a list of 20 random bits has been produced (“cb” sets the symbol for the begin of a clause, “ce” for the end of a clause, “-nc” switches off the output of the comment part, and with “-nl” we produced a final end-of-line). With

```
> OKgenerator n=5 l=5 cp=8 cb="" ce="\n" sep="\t" -nc -o
```

```

-3      -4      1      -5      2
-2      4       1      -3      5
2       4       5       3      -1
4       3      -5      -1      2
2       1       3       4       5
5       -3      -2      4       1
-3      -4      -1      -5      2
2       -5      -1      -4      -3

```

a list of 8 random permutations from  $S_5$  pointwise multiplied with random vectors from  $\{-1, +1\}^5$  has been created (the entries separated by tabulator characters, each new permutation on a new line).

## D An example calculation

```

> OKgenerator n=100 l=3 cp=2 -o
% OKgenerator( (0, 0), (0, 0), 100; 3, 2; )
% Standardised density: 3 -> 0.02
(30,-71,-75)
(18,-9,-100)

```

**First clause:**  $(30, -71, -75)$

**First literal:** 30

$$\begin{aligned} \lg(0, 0, 100, 3, 0) &= \alpha_{100}(\text{aes}(0 * 2^{64} + 0, 100 \cdot 2^{96} + 3 \cdot 2^{64} + 0) \bmod 2 \cdot 100) \\ &= \alpha_{100}(\text{aes}(0, 7922816251481773991575523688448) \bmod 200) \end{aligned}$$

where

$$\begin{aligned} \text{aes}(0, 7922816251481773991575523688448) &= \\ &266358227669704183816922654441660243029 \end{aligned}$$

and thus

$$\begin{aligned} \text{aes}(0, 7922816251481773991575523688448) \bmod 200 &= 29 \\ \alpha_{100}(29) &= 29 + 1 = 30 \end{aligned}$$

**Second literal:**  $-71$

$$\begin{aligned}\lg(0, 0, 100 - 1, 3, 1) &= \alpha_{99}(\text{aes}(0, 99 \cdot 2^{96} + 3 \cdot 2^{64} + 1) \bmod 2 \cdot 99) \\ &= \alpha_{99}(\text{aes}(0, 7843588088967509653981979738113) \bmod 198) \\ &= \alpha_{99}(108985992402053770742915617566895285398 \bmod 198) \\ &= \alpha_{99}(168) = -(168 - (99 - 1)) = -70\end{aligned}$$

Thus the sign of the second literal is negative, while the variable is

$$(\{1, \dots, 100\} \setminus \{30\})_{70} = 70 + 1 = 71.$$

**Third literal:**  $-75$

$$\begin{aligned}\lg(0, 0, 100 - 2, 3, 2) &= \alpha_{98}(\text{aes}(0, 98 \cdot 2^{96} + 3 \cdot 2^{64} + 2) \bmod 2 \cdot 98) \\ &= \alpha_{98}(\text{aes}(0, 7764359926453245316388435787778) \bmod 196) \\ &= \alpha_{98}(97924266970237634107371866759759676590 \bmod 196) \\ &= \alpha_{98}(170) = -(170 - (98 - 1)) = -73\end{aligned}$$

$$(\{1, \dots, 100\} \setminus \{30, 71\})_{73} = 73 + 2 = 75.$$

**Second clause:**  $(18, -9, -100)$

**First literal:**  $18$

$$\begin{aligned}\lg(0, 0, 100, 3, 3) &= \alpha_{100}(\text{aes}(0, 100 \cdot 2^{96} + 3 \cdot 2^{64} + 3) \bmod 2 \cdot 100) \\ &= \alpha_{100}(\text{aes}(0, 7922816251481773991575523688451) \bmod 200) \\ &= \alpha_{100}(75904446460713774676643846235182933817 \bmod 200) \\ &= \alpha_{98}(17) = 17 + 1 = 18\end{aligned}$$

**Second literal:**  $-9$

$$\begin{aligned}\lg(0, 0, 100 - 1, 3, 4) &= \alpha_{99}(\text{aes}(0, 99 \cdot 2^{96} + 3 \cdot 2^{64} + 4) \bmod 2 \cdot 99) \\ &= \alpha_{99}(\text{aes}(0, 7843588088967509653981979738116) \bmod 198) \\ &= \alpha_{99}(124525820386698792474883652729908414127 \bmod 198) \\ &= \alpha_{99}(107) = -(107 - (99 - 1)) = -9\end{aligned}$$

$$(\{1, \dots, 100\} \setminus \{18\})_9 = 9.$$

**Third literal:**  $-100$

$$\begin{aligned} \lg(0, 0, 100 - 2, 3, 5) &= \alpha_{98}(\text{aes}(0, 98 \cdot 2^{96} + 3 \cdot 2^{64} + 5) \bmod 2 \cdot 98) \\ &= \alpha_{98}(\text{aes}(0, 7764359926453245316388435787781) \bmod 196) \\ &= \alpha_{98}(108144913130167446732707613185532996963 \bmod 196) \\ &= \alpha_{98}(195) = -(195 - (98 - 1)) = -98 \\ (\{1, \dots, 100\} \setminus \{18, 9\})_{98} &= 98 + 2 = 100. \end{aligned}$$

## E A tool for simplified creation of large experiments

The software package “OKRandGen” contains besides `OKsolver` (obtained by applying the command `make`) also a (first) tool for automated creation of experiments, called `interpret_descriptor`, which is obtained by using

```
make interpret_descriptor.
```

It’s basic function is to read from standard input a description of an experiment, and to translate this into a sequence of calls of `OKgenerator`, which is written to standard output. For example, assume that in file “description1” you have the following text:

```
# Oliver Kullmann, 13.3.2002 (Swansea)
# test example

N = 10; # number of formulas
n = 200 to 300 step 20; # number of variables
l = 3; # clause length
dp = 1, 2 to 3 step 0.2, 3.1 to 4 step 0.1, 4.05 to 4.4 step 0.05,
    4.5 to 5 step 0.1, 5.2 to 6 step .2, 8, 12, 20;
# densities belonging to clause-length 3
l = 2;
dp = 0 to 1.2 step 0.2;
```

then by

```
cat description1 | interpret_descriptor > translation1
```

you create a file “translation1” with

$$5 * (6 * (1 + 6 + 10 + 8 + 6 + 5 + 3) * 7) = 8190$$

lines according to the three loops in the above kind of “program”, starting with

```

OKgenerator s0=0 s1=0 n=200 l=3 cp=200 -D nr0=0 nr1=0 -o
OKgenerator s0=0 s1=0 n=200 l=3 cp=200 -D nr0=0 nr1=1 -o
OKgenerator s0=0 s1=0 n=200 l=3 cp=200 -D nr0=0 nr1=2 -o
OKgenerator s0=0 s1=0 n=200 l=3 cp=200 -D nr0=0 nr1=3 -o
OKgenerator s0=0 s1=0 n=200 l=3 cp=200 -D nr0=0 nr1=4 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=0 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=1 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=2 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=3 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=4 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=80 l=3 cp=200 -D nr0=0 nr1=0 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=80 l=3 cp=200 -D nr0=0 nr1=1 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=80 l=3 cp=200 -D nr0=0 nr1=2 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=80 l=3 cp=200 -D nr0=0 nr1=3 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=80 l=3 cp=200 -D nr0=0 nr1=4 -o
OKgenerator s0=0 s1=0 n=200 l=2 cp=120 l=3 cp=200 -D nr0=0 nr1=0 -o

```

One may specify the seed and the initial formula number as different from the default values (zero) by

```

# Oliver Kullmann, 13.3.2002 (Swansea)
# test example 2

s = 6171819;
nr = 140;
N = 5; # number of formulas
n = 200 to 300 step 20; # number of variables
l = 3; # clause length
dp = 1, 2 to 3 step 0.2, 3.1 to 4 step 0.1, 4.05 to 4.4 step 0.05,
    4.5 to 5 step 0.1, 5.2 to 6 step .2, 8, 12, 20;
# densities belonging to clause-length 3
l = 2;
dp = 0 to 1.2 step 0.2;

```

which (again) yields 8190 lines of calling sequences for `OKgenerator` (using `cat description2 | interprete_descriptor > translation2`), beginning with

```

OKgenerator s0=0 s1=6171819 n=200 l=3 cp=200 -D nr0=0 nr1=140 -o
OKgenerator s0=0 s1=6171819 n=200 l=3 cp=200 -D nr0=0 nr1=141 -o
OKgenerator s0=0 s1=6171819 n=200 l=3 cp=200 -D nr0=0 nr1=142 -o
OKgenerator s0=0 s1=6171819 n=200 l=3 cp=200 -D nr0=0 nr1=143 -o
OKgenerator s0=0 s1=6171819 n=200 l=3 cp=200 -D nr0=0 nr1=144 -o
OKgenerator s0=0 s1=6171819 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=140 -o
OKgenerator s0=0 s1=6171819 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=141 -o

```

```

OKgenerator s0=0 s1=6171819 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=142 -o
OKgenerator s0=0 s1=6171819 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=143 -o
OKgenerator s0=0 s1=6171819 n=200 l=2 cp=40 l=3 cp=200 -D nr0=0 nr1=144 -o
OKgenerator s0=0 s1=6171819 n=200 l=2 cp=80 l=3 cp=200 -D nr0=0 nr1=140 -o

```

The general syntax of an “experiment descriptor” is as follows:

- Comments, starting with “#” and continuing until the end of the line, are ignored.
- Except of the use of the end-of-line symbol to end a comment, space characters are also ignored (so that the number “1600819315” may be written as “1 600 819 315”).

Now an “experiment descriptor” is a sequence of “assignments”, where an assignment has the form

$$V = \text{list of intervalls};$$

where the variable  $V$  is one of

**s** seed

**nr** formula number

**N** number of experiments

**sat** sat-status (see below)

**l** clause-length

**cp** number of clauses of the current clause-length

**dp** relative density for the current clause-length

**ds** domain size (for generalised clause-sets).

The “sat-status” is one of “0” (unsatisfiable), “1” (satisfiable) or “2” (both satisfiable and unsatisfiable cases). The creation of (only) satisfiable or (only) unsatisfiable instances requires access to a database, and is not implemented yet. The default value for variable **sat** is 2, while for **s** and **nr** the default value is 0. The items of the “list of intervalls” is a comma-separated list of “intervalls”, where an intervall is of the form

$$\begin{aligned}
 & a \\
 & a \text{ to } b \\
 & a \text{ to } b \text{ step } c
 \end{aligned}$$

for “numbers”  $a, b, c$  (the default step value for the second form is 1). Finally a number is of the form

$$\begin{aligned} & n \\ & n/m \\ & n.m \end{aligned}$$

for sequences  $n, m$  of decimal digits.

All calculations are precise. The order of the output corresponds to the order of assignments. If there are several assignments regarding the same clause-length, then the corresponding numbers of clause-occurrences are summed up.