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)

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/

February 8, 2002

Abstract

We introduce an adaptive density-based heuristics for SAT solvers. This heuristics has a solver-independent part, estimating probabilities for being satisfiable by means of thresholds, and a solver-dependent part for estimating running times on satisfiable and unsatisfiable instances. Both parts rely on a (large) database for random formulas, generated by a reliable, reproducible and strong (pseudo-)random formula generator "OKgenerator", which is based on AES (Advanced Encryption Standard, the successor of DES)..

1 Introduction

Let us consider a boolean formula F and the problem of finding a good (simple) branching for a given SAT solver. There are $2 \cdot n(F)$ many choices, where $n(F)$ is the number of variables, namely for each variable $v \in \mathrm{var}(F)$ the branchings $v^{(0)} =$ "first $\langle v \to 0\rangle$, then $\langle v \to 1\rangle$" and $v^{(1)} =$ "first $\langle v \to 1\rangle$, then $\langle v \to 0\rangle$". A reasonable measurement for the complexity when choosing branching $v^{(\varepsilon)}$ is

$$\begin{aligned}&\mathrm{prob\_SAT}_{\varepsilon} \cdot \mathrm{av\_steps\_SAT}_{\varepsilon} + \\ &\mathrm{prob\_USAT}_{\varepsilon} \cdot (\mathrm{prob\_SAT}_{\overline{\varepsilon}} \cdot \mathrm{av\_steps\_SAT}_{\overline{\varepsilon}} + \\ &\qquad\qquad \mathrm{prob\_USAT}_{\overline{\varepsilon}} \cdot \mathrm{av\_steps\_USAT}_{\overline{\varepsilon}})\end{aligned}$$

where

- $\mathrm{prob\_(U)SAT}_{\varepsilon}$ is (an estimation of) the *probability* that a formula "like" $\langle v \to \varepsilon\rangle * F$ is (un-)satisfiable (where $\langle v \to \varepsilon\rangle * F$ is the result of application of partial assignment $v \to \varepsilon$ to F);

- av_steps_(U)SAT$_\varepsilon$ is (an estimation of) the *average number of steps* it takes to process a(n) (un-)satisfiable formula "like" $\langle v \to \varepsilon \rangle * F$.

The basic problem 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 $\Delta(F)$ of F is the function with domain the different clause-sizes p occurring in F mapping $p \mapsto \Delta(F)(p) = \frac{c^{(p)}(F)}{n(F)}$, using $c^{(p)}(F)$ for the number of clauses in F of size p. We assume a "strengthened threshold conjecture", which asserts that for n going to infinity the probability of being satisfiable for clause-sets with fixed mixed density Δ 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 $\gamma(\Delta)$ for being unsatisfiable[1)] thus we have to determine the thresholds $\Gamma(\Delta, p)$ for arbitrary mixed densities Δ and clause-sizes $p \notin \mathrm{dom}(\Gamma)$, where we have $\gamma(\Delta^-_{p,\varepsilon}) = 0$ and $\gamma(\Delta^+_{p,\varepsilon}) = 1$ for $\varepsilon > 0$, where $\Delta^\pm_{p,\varepsilon}$ is obtained from Δ by mapping additionally p to $\Gamma(\Delta, p) \pm \varepsilon$.

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. Given the parameters, the generated formula must be *reproducible* on any system having an ISO/IEC 14882 compliant C++ compiler (C++ since this is the only programming language where we have a precise standard *and* good compilers on many platforms).[2)]
2. The range of parameters must be *known*, *reasonably large*, and the behaviour of the generator must be *stable* over this whole range.[3)]
3. In case we finally realise (after *many* hours of computation time) that the formulas produced by our (pseudo)-random generator are easier for some 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

1) rather than for being satisfiable, since then γ is non-decreasing, with values 0 for "small" densities and values 1 for "large" densities

2) Changing the environment may change the behaviour for all formula generators I have seen yet.

3) 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!

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!

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. By $\mathbb{N} = \{1, 2, 3, \ldots\}$ we denote the set of positive integers.

2 Mixed random formulas

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

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

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

$$\boldsymbol{\mathcal{MCLS}(n, (p_1, \ldots, p_m), (c_1, \ldots, c_m))} := \bigcup_{i=1}^{m} \mathcal{MCLS}(n, p_i, c_i)$$

be the set of all multi-clause-sets with m different clause-lengths. Considering all elements of $\mathcal{MCLS}(n, (p_1, \ldots, p_m), (c_1, \ldots, c_m))$ as having the same probability, we define the probability of the event of unsatisfiability in this (finite) probability space by

$$\boldsymbol{P_0(n, (p_1, \ldots, p_m), (c_1, \ldots, c_m))} := \frac{|\mathcal{MCLS}(n, (p_1, \ldots, p_m), (c_1, \ldots, c_m)) \setminus \mathcal{SAT}|}{|\mathcal{MCLS}(n, (p_1, \ldots, p_m), (c_1, \ldots, c_m))|}.$$

Let $r : \mathbb{R}_{\geq 0} \to \mathbb{N}_0$, $r(x) := \lfloor x \rfloor$ 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 $m \in \mathbb{N}$, $p_1, \ldots, p_m \in \mathbb{N}$ with $p_1 < \cdots < p_m$ and for all $\Delta_1, \ldots, \Delta_m \in \mathbb{R}_{\geq 0}$ the limit*

$$\lim_{n \to \infty} P_0(n, (p_1, \ldots, p_m), (r(\Delta_1 \cdot n), \ldots, r(\Delta_m \cdot n)))$$

exists.

Let $\mathcal{D}$ be the set of all maps $\Delta : P \to \mathbb{R}_{>0}$ for finite $P \subseteq \mathbb{N}$. We define $\boldsymbol{\gamma} : \mathcal{D} \to [0, 1]$ by $\gamma(\emptyset) := 0$, while for $\Delta : P \to \mathbb{R}_{>0}$, $P \neq \emptyset$, $P = \{p_1, \ldots, p_m\}$, $p_1 < \cdots < p_m$ let

$$\gamma(\Delta) := \lim_{n \to \infty} P_0(n, (p_1, \ldots, p_m), (r(\Delta_1 \cdot n), \ldots, r(\Delta_m \cdot n))).$$

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

$$\Delta \leq \Delta' :\Longleftrightarrow \mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Delta') \wedge \forall\, p \in \mathrm{dom}(\Delta) : \Delta(p) \leq \Delta'(p),$$

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

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

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

An equivalent formulation is that for all $\Delta, \Delta' \in \mathcal{D}$ with $\Delta' < \Delta$ and $0 < \gamma(\Delta) < 1$ we have $\gamma(\Delta') = 0$, and we see that that this part of the conjecture generalises Corollary 1 in [2].

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

$$x < \Gamma(\Delta, p) \Rightarrow \gamma(\Delta \cup \langle p \to x \rangle) = 0$$
$$x > \Gamma(\Delta, p) \Rightarrow \gamma(\Delta \cup \langle p \to x \rangle) = 1$$

holds. We have the following general properties:

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

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 [12] is 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 $\frac{2}{3} = \frac{\frac{2}{5}}{1 - \frac{2}{5}}$), 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 $\Gamma(\langle 3 \to \frac{2}{3} \rangle, 2) = 1$. It follows that for all $0 \leq \alpha \leq \frac{2}{3}$ we have $\Gamma(\langle 3 \to \alpha \rangle, 2) = 1$.

With $\Gamma : \mathcal{D} \times \mathbb{N} \to \mathbb{R}_{\geq 0}$ 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 Γ), but it does not capture for example the assertion in [8], that for all $\lambda \in [0, 1]$ the functions $x \in \mathbb{R}_{\geq 0} \mapsto \gamma(\langle 2 \mapsto (1 - \lambda) \cdot x, 3 \mapsto \lambda \cdot x \rangle)$ 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 $f : \mathbb{R}_{\geq 0} \to \mathcal{D}$ 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 $\Delta(F) \in \mathcal{D}$ be the map with domain $\{|C| : C \in F\}$, which maps $p \mapsto \frac{c^{(p)}(F)}{n(F)}$. Assume an "approximation" $\tilde{\gamma} : \mathcal{D} \to [0,1]$ of $\gamma : \mathcal{D} \to [0,1]$ is given.

Let the "specified density" $\hat{\boldsymbol{\Delta}}\boldsymbol{(F)}$ be the pair $(n(F), \Delta(F)) \in \mathcal{S}$, where $\mathcal{S}$ is the set of all pairs $(n, \Delta) \in \mathbb{N}_0 \times \mathcal{D}$ such that for all $p \in \mathrm{dom}(\Delta)$ we have $p \leq n$. Consider a SAT solver $\mathcal{A} : \mathcal{MCLS}' \to \{0,1\}$. We assume functions $\mu_0, \mu_1 : \mathcal{S} \to \mathbb{R}_{\geq 1}$, where $\mu_\varepsilon(n, \Delta)$ for $\Delta : \{p_1, \dots, p_m\} \to \mathbb{R}_{\geq 0}$ ($m \in \mathbb{N}_0$) approximates the average number of "leaves" for a run of $\mathcal{A}$ on unsatisfiable ($\varepsilon = 0$) resp. satisfiable ($\varepsilon = 1$) clause-set $F \in \mathcal{MCLS}(n, (p_1, \dots, p_m), (r(\Delta(p_1) \cdot n), \dots, r(\Delta(p_m) \cdot n)))$.

We split the expected "mathematical running time" of the solver $\mathcal{A}$ into functions μ_0, μ_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 or 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 $\langle v \to 0\rangle$, then $\langle v \to 1\rangle$", and $v^{(1)}$ stands for the branching "first $\langle v \to 1\rangle$, then $\langle v \to 0\rangle$". We define the "complexity" of $v^{(\varepsilon)}$ as (using $F_\varepsilon := \langle v \to \varepsilon\rangle * F$)

$$\begin{aligned}\mathbf{comp}(\boldsymbol{v^{(\varepsilon)}}) := (1 - \tilde{\gamma}(\Delta(F_\varepsilon))) \cdot \mu_1(\hat{\Delta}(F_\varepsilon)) + \\ \tilde{\gamma}(\Delta(F_\varepsilon)) \cdot \big(\mu_0(\hat{\Delta}(F_\varepsilon)) + \\ (1 - \tilde{\gamma}(\Delta(F_{1-\varepsilon}))) \cdot \mu_1(\hat{\Delta}(F_{1-\varepsilon})) + \\ \tilde{\gamma}(\Delta(F_{1-\varepsilon})) \cdot \mu_0(\hat{\Delta}(F_{1-\varepsilon}))\big).\end{aligned}$$

Now choose the branching $v^{(\varepsilon)}$ with minimal $\mathrm{comp}(v^{(\varepsilon)})$. For the computation of $\tilde{\gamma}$ (solver independent) and of μ_0, μ_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 μ_0, μ_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 $\gamma(\Delta)$ will be available (for some finite set of $\Delta \in \mathcal{D}$), and (hopefully) these approximations can be integrated into a reasonable way for computing $\tilde{\gamma}(\Delta)$. And by supplying further information on running times and tree sizes, we will try to figure out how to compute $\mu_0(\hat{\Delta}), \mu_1(\hat{\Delta})$ for $\hat{\Delta} \in \mathcal{S}$ (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`" ([10]). 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 $p_1 < \cdots < p_m \leq n$ 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 *intended* output is a random element of $\mathcal{MCLS}(n, (p_1, \dots, p_m), (c_1, \dots, c_m))$ (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. `OKgenerator`$(s, k, n, (p_1, \dots, p_m), (c_1, \dots, c_m))$ is the concatenation of the constant clause-length formulas `OKgenerator`(s, k, n, p_i, c_i), $i = 1, \dots, m$.
2. The formulas `OKgenerator`(s, k, n, p, c) and `OKgenerator`(s', k', n', p', c') shall be "completely unrelated" as soon as $(s, k, n, p) \neq (s', k', n', p')$.
3. `OKgenerator`(s, k, n, p, c) is a prefix of `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 select randomly 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 $k = 0, 1, \ldots$ to obtain a (long) sequence of random formulas for fixed n, m, $(p_1, \ldots, p_m)$ and $(c_1, \ldots, c_m)$.

3. Consider two sequences $F_0, F_1, \ldots$ and $F'_0, F'_1, \ldots$ of random formulas, the first with respect to seed $s = 0$ and fixed parameters $n, m, (p_1, \ldots, p_m), (c_1, \ldots, c_m)$ where $k = 0, 1, \ldots$, while the second sequence has parameters $s = 0$ and fixed $n', m', (p'_1, \ldots, p'_m), (c'_1, \ldots, c'_m)$ and again $k = 0, 1, \ldots$. 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.[4)]

For $k \in \mathbb{N}$ let $\boldsymbol{\mathcal{W}_k} := \{0, \ldots, 2^k - 1\}$ 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

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

which is derived from the block cipher AES : $\{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$ (the first argument is the key, the second parameter the input block (the "plain text"), and the return value is the encrypted input block (the "cipher text")) as defined in [5] in the natural way, considering sequences of bits as binary numbers (leading bits first). See Appendix A for more details.

Based on aes, we derive the literal generator

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

in the following way, where "variables" are given by the elements of $\mathcal{W}_{31} \setminus \{0\}$, and the "literal" $\mathrm{lg}(s, k, n, p, i)$ 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 $\alpha_n : \{0, \ldots, 2n-1\} \to \{-n, \cdots -1\} \cup \{1, \ldots, n\}$ 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

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

Based on the literal generator, 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

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

[4)] 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.

where the clauses $C_i = (l_{i,1}, \ldots, l_{i,p})$ (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

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

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

$$|l_{i,1}| = |x_{i,1}|,$$
$$|l_{i,j}| = \big(\{1,\ldots,n\} \setminus \{|l_{i,1}|,\ldots,|l_{i,j-1}|\}\big)_{|x_j|} \text{ for } j > 1$$

In order to handle mixed clause-length, let "$*$" denote concatenation of tuples, i.e. $(a_1,\ldots,a_m)*(b_1,\ldots,b_n) = (a_1,\ldots,a_m,b_1,\ldots,b_n)$. 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 $p_1 < \cdots < p_m \leq n$, let

$$\mathbf{mcnfg}(s,\ k,\ n;\ (p_1,\ldots,p_m),(c_1,\ldots,c_m)) := \mathop{*}_{i=1}^{m} \mathrm{cnfg}(s,k,n,p_i,c_i).$$

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.

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, 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. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, July 1999.

[9] 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.

[10] Oliver Kullmann. The implementation OKgenerator. Available at http://cs-svr1.swan.ac.uk/~csoliver/, February 2002. Version 1.2, Release 1.0.

[11] Michael Welschenbach. *Cryptography in C and C++*. Apress (Springer-Verlag), 2001.

[12] David Wilson, February 2002. Personal Communication.

A AES — from bits to numbers

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

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

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 AES : $\{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$, where the first argument is the key, the second argument the input block (the "plain text" in EBC mode), and the return value is the encrypted input block (the "cipher text" in EBC mode). We interprete 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, explicitely, a block $x \in \{0,1\}^{128}$ corresponds to the number $\sum_{i=1}^{128} x_i \cdot 2^{128-i}$.

Typically, in implementations[5)] input, output and key are one-dimensional arrays of 8-bit bytes, each of dimension 16, where the array indices range from $0 \ldots 15$. Thus now a 128-bit block x corresponds to a vector $x = (x_0, \ldots, x_{15}) \in \{0, \ldots, 255\}^{16}$, 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 hexadecimal notation (each block consists of 16 bytes) and then using numbers:

AES(0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0,
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0) =
66 e9 4b d4 ef 8a 2c 3b 88 4c fa 59 ca 34 2b 2e

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

AES(0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0,
80 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0) =
32 41 e3 d4 99 c7 3d 2f 38 73 73 1b 16 d9 e0 60

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

5) 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 [9] (with uniform domains $D_v = \{1, \dots, d\}$), where literals are pairs (v, ε), v a variable, $\varepsilon \in \{1, \dots, d\}$, 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.

Based on aes, we derive the literal generator

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

where the literal $\mathrm{lg}^*(s, k, n, d, p, i)$ 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 let

$$A := \mathrm{aes}(s \cdot 2^{64} + (d-2) \cdot 2^{32} + k, \; n \cdot 2^{96} + p \cdot 2^{64} + i).$$

and set

$$\mathrm{lg}^*(s, k, n, d, p, i) := (v, \varepsilon)$$

where

$$v := ((A \bmod (n \cdot d)) \bmod n) + 1 = (A \bmod n) + 1$$

$$\varepsilon := ((A \bmod (n \cdot d)) \operatorname{div} n) + 1$$

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

$$\boldsymbol{\mathrm{gcnfg}(s, k, n, d, p, c)} := (C_1, \dots, C_c),$$

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

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

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

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

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

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

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

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

when identifying literals $(v, 1)$ with v and $(v, 2)$ with $\overline{v}$. `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 [11] 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=2/5 l=2 dp=8/10 d=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).