Randomness as a Constraint

S. D. Prestwich[1], R. Rossi[2], and S. A. Tarim[3]
[1] *Insight Centre for Data Analytics, University College Cork, Ireland*
[2] *University of Edinburgh Business School, Edinburgh, UK*
[3] *Department of Management, Cankaya University, Ankara, Turkey*

Abstract. Some optimisation problems require a random-looking solution with no apparent patterns, for reasons of fairness, anonymity, undetectability or unpredictability. Randomised search is not a good general approach because problem constraints and objective functions may lead to solutions that are far from random. We propose a constraint-based approach to finding pseudo-random solutions, inspired by the Kolmogorov complexity definition of randomness and by data compression methods. Our "entropy constraints" can be implemented in constraint programming systems using well-known global constraints. We apply them to a problem from experimental psychology and to a factory inspection problem.

1 Introduction

For some applications we require a list of numbers, or some other data structure, that is (or appears to be) *random*, while also satisfying certain constraints. Examples include the design of randomised experiments to avoid statistical bias [13], the generation of random phylogenetic trees [14], quasirandom (low discrepancy) sequences for efficient numerical integration and global optimisation [24], randomised lists without repetition for use in experimental psychology [10], random programs for compiler verification [6], and the random scheduling of inspections for the sake of unpredictability [30].

An obvious approach to obtaining a random-looking solution is simply to use a randomised search strategy, such as stochastic local search or backtrack search with a randomised value ordering. In some cases this works, for example it can generate a random permutation of a list, but in general there are several drawbacks with the randomised search approach:

- If only random-looking solutions are acceptable then the constraint model is not correct, as it permits solutions that are unacceptable. The correctness of a constraint model should be independent of the search strategy used to solve it.
- Randomised search can not prove that a random-looking solution does not exist, or prove that a solution is as random-looking as possible.
- Unless the randomised search is carefully designed (see [9] for example) it is likely to make a biased choice of solution.

– Even if we sample solutions in an unbiased way, there is no guarantee that such a solution will look random. Although randomly sampled unconstrained sequences are almost certain to appear random (almost all long sequences have high *algorithmic entropy* [2]) a constrained problem might have mostly regular-looking solutions. Similarly, optimising an objective function might lead to regular-looking solutions. In Section 3.2 we give examples of both phenomena.

Instead it would be useful to have available a constraint `israndom`($\boldsymbol{v}$) that forces a vector $\boldsymbol{v}$ of variables to be random, which could simply be added to a constraint model. First, however, we must define what we mean by *random*.

In information theory, randomness is a property of the data source used to generate a data sequence, not of a single sequence. The *Shannon entropy* of the source can be computed from its symbol probabilities using Shannon's well-known formula [32]. But in algorithmic information theory, randomness can be viewed as a property of a specific data sequence, and its *Kolmogorov complexity*, or *algorithmic entropy*, is defined as the length of the smallest algorithm that can describe it. For example the sequence 1111111111 may have the same probability of occurring as 1010110001 but it has lower algorithmic entropy because it can be described more simply (write 1 ten times). Algorithmic entropy formally captures the intuitive notion of whether a list of numbers "looks random", making it useful for our purposes. We shall refer to algorithmic entropy simply as *entropy* by a slight abuse of language.

Having chosen (algorithmic) entropy as our measure of randomness, we would like to have available a constraint of the form `entropy`($\boldsymbol{v}, e$) to ensure that the entropy of $\boldsymbol{v}$ is at least e. Unfortunately, defining such a constraint is impossible because algorithmic entropy is uncomputable [5]. Instead we take a pragmatic approach by defining constraints that eliminate patterns exploited by well-known data compression algorithms, which can be combined as needed for specific applications. There is a close relationship between algorithmic entropy and compressibility: applying a compression algorithm to a sequence of numbers, and measuring the length of the compressed sequence, gives an upper bound on the algorithmic entropy of the original sequence. Thus by excluding readily-compressible solutions we hope to exclude low-entropy (non-random) solutions.

The paper is organised as follows. Section 2 presents constraint-based approaches to limiting the search to high-entropy solutions. Section 3 applies these ideas to two problems. Section 4 discusses related work. Section 5 concludes the paper and discusses future work.

2 Entropy constraints

We require constraints to exclude low-entropy solutions, which we shall call *entropy constraints*. This raises several practical questions: what types of pattern can be excluded by constraints, how the constraints can be implemented, and what filtering algorithms are available? We address these problems below. In our experiments we use the Eclipse constraint logic programming system [1].

2.1 Non-uniform distributions

Data in which some symbols occur more often than others have non-uniform probability distributions. Huffman coding [15] and arithmetic coding [29] are compression methods that exploit this feature by encoding symbols with a variable number of bits (a *prefix code* with many bits for rare symbols and few bits for common symbols).

To eliminate this type of pattern from our solutions we define some simple entropy constraints. We assume that all variables have the same domain $\{0, \ldots, m-1\}$ but it is easy to generalise to different domains. Given lower bounds $\boldsymbol{\ell} = \langle \ell_0, \ldots, \ell_{m-1} \rangle$ and upper bounds $\boldsymbol{u} = \langle u_0, \ldots, u_{m-1} \rangle$ on the frequencies of symbols $0, \ldots, m-1$ we define a *frequency entropy constraint* $\texttt{freq}(\boldsymbol{v}, m, \boldsymbol{\ell}, \boldsymbol{u})$. It can be directly implemented by a global cardinality constraint [27] $\texttt{GCC}(\boldsymbol{v}, \langle 0, \ldots, m-1 \rangle, \boldsymbol{\ell}, \boldsymbol{u})$.

To see the effect of these constraints, consider a CSP with 100 finite domain variables $v_0, \ldots, v_{99} \in \{0, 1, \ldots, 9\}$ and no problem constraints. We use the default backtrack search heuristic to find the *lex-least solution* which we take as a proxy for the lowest-entropy solution. The lex-least solution is the solution that is least under the obvious lexicographical ordering (order by increasing value of the first variable, breaking ties using the second variable and so on, under the static variable ordering $v_1, v_2, \ldots$). If even the lex-least solution has high entropy then we believe that other solutions will too. To test this we applied randomised search to a problem with entropy constraints, and never obtained a solution that could be compressed further than the lex-least solution.

If we add $\texttt{freq}(\boldsymbol{v}, 10, \mathbf{0}, \mathbf{14})$[1] to this problem we obtain the lex-least solution

00000000000000111111111111112222222222222222333333333
33333344444444444444555555555555556666666666666677

We shall estimate the entropy of this lex-least solution by compressing it using the well-known gzip[2] compression algorithm. Though gzip does not necessarily give an accurate entropy estimate it has been successfully used for this purpose (see for example [4]). Denoting the entropy of solutions by ϵ and measuring it in bytes, the lex-least solution has $\epsilon = 43$. For a random sequence of digits in the range 0–9 we typically obtain ϵ values of 80–83 so the lex-least solution is far from random, as can be observed. With no entropy constraints the lex-least solution is simply 0 repeated 100 times, which gzip compresses from 100 bytes to 26 bytes. There is no paradox in compressing these *random* numbers: integers in the range 0–9 can be represented by fewer than 8 bits (1 byte). Note that gzip compresses an empty text file to 22 bytes, so most of the 26 bytes is decompression metadata.

2.2 Repeated strings

For the compression of discrete data probably the best-known methods are based on *adaptive dictionaries*. These underlie compression algorithms such as Linux

[1] We use $\boldsymbol{i}$ with integer i to denote a vector $\langle i, \ldots, i \rangle$ of appropriate length.

[2] http://www.gzip.org/

`compress`, V.24 bis, GIF, PKZip, Zip, LHarc, PNG, gzip and ARJ [31] and use algorithms described in [34–36]. These methods detect repeated k-grams (blocks of k adjacent symbols) and replace them by pointers to dictionary entries. For example the string 011101011100 contains two occurrences of the 5-gram 01110, which can be stored in a dictionary and both its occurrences replaced by a pointer to that entry.

We could generalise the `freq` constraint to limit the number of occurrences of every possible k-gram, but as there are m^k of them this is impractical unless k is small. A more scalable approach is as follows. Given an integer $k \geq 2$ and an upper bound t on the number of occurrences of all k-grams over symbols $\{0, \ldots, m-1\}$ in a vector $\boldsymbol{v}$ of variables, we define a constraint $\texttt{dict}(\boldsymbol{v}, m, k, t)$. We shall call this a *dictionary entropy constraint*. It can be implemented by the $\texttt{Multi-Inter-Distance}(\boldsymbol{x}, t, p)$ global constraint [20] with $p = n - k + 1$ (the number of $\boldsymbol{x}$ variables), $\boldsymbol{x} = \langle x_0, \ldots, x_{n-k} \rangle$, and the $x_i = \sum_{j=0}^{k-1} m^j v_{i+j}$ are auxiliary integer variables representing k-grams, where n is the sequence length. This global constraint enforces an upper bound t on the number of occurrences of each value within any p consecutive $\boldsymbol{x}$ variables. In the special case $t = 1$ we can instead use $\texttt{alldifferent}(\boldsymbol{x})$ [28].

To test this idea on the artificial problem of Section 2.1 we add $\texttt{dict}(\boldsymbol{v}, 10, k, 1)$ dictionary constraints for various k-values (but not the frequency constraints of Section 2.1). The results in Table 1 show that as we reduce k the solution contains fewer obvious patterns, and $k = 2$ gives a solution that (to gzip) is indistinguishable from a purely random sequence, as 80 bytes is within the range of ϵ for random sequences of this form.

k	lex-least solution	ϵ
50	00000000000000000000000000000000000000000000000001	
	00000000000000000000000000000000000000000000000000	29
25	00000000000000000000000001000000000000000000000000	
	20000000000000000000000003000000000000000000000000	35
12	00000000000010000000000020000000000030000000000040	
	00000000005000000000006000000000007000000000008000	48
6	00000010000020000030000040000050000060000070000080	
	00009000011000012000013000014000015000016000017000	60
3	00010020030040050060070080090110120130140150160170	
	18019021022023024025026027028029031032033034035036	71
2	00102030405060708091121314151617181922324252627282	
	93343536373839445464748495565758596676869778798890	80

Table 1. Lex-least solutions with dictionary entropy constraints

In this example there are sufficient digrams (2-grams) to avoid any repetition, but for smaller m this will not be true, and in the general case we might need to use larger t or larger k. Note that we must choose $t_k \geq \lceil n/m^k \rceil$ otherwise the problem will be unsatisfiable.

2.3 Correlated sources

Though gzip and related compression algorithms often do a very good job, they are not designed to detect all patterns. A solution with no repeated k-grams might nevertheless have low entropy and noticeable patterns. For example the following sequence of 100 integers in the range 0–9 compresses to 80 bytes, and is therefore indistinguishable from a random sequence to gzip:

> 01234567890246813579036914725804815926370516273849
> 94837261507362951840852741963097531864209876543210

Yet it was written by hand following a simple pattern and is certainly not a random sequence, as becomes apparent if we examine the differences between adjacent symbols:

> 1 1 1 1 1 1 1 1 1 -9 2 2 2 2 -7 2 2 2 2 -9 3 3 3 -8 3 3 -5 3 3 -8 4 4 -7 4 4 -7 4
> -3 4 -7 5 -4 5 -4 5 -4 5 -4 5 0 -5 4 -5 4 -5 4 -5 4 -5 7 -4 3 -4 7 -4 -4 7 -4 -4 8
> -3 -3 5 -3 -3 8 -3 -3 -3 9 -2 -2 -2 -2 7 -2 -2 -2 -2 9 -1 -1 -1 -1 -1 -1 -1 -1 -1

The same differences often occur together but gzip is not designed to detect this type of pattern. As another example, the high-entropy ($k = 2$) solution found in Section 2.2 has differences

> jjkilhmgnfoepdqcrbsbjjkilhmgnfoepdqcrcjjkilhmgnfoe
> pdqdjjkilhmgnfoepejjkilhmgnfofjjkilhmgngjjkilhmhj

where differences $-9 \ldots +9$ are represented by symbols a–s. The differences also look quite random: gzip compresses this list of symbols to 92 bytes which is typical of a random sequence of 99 symbols from a–s. Yet they have a non-uniform distribution: a does not occur at all while j occurs 15 times.

In data compression an example of a *correlated source* of data is one in which each symbol depends probabilistically on its predecessor. This pattern is exploited in speech compression methods such as DPCM [7] and its variants. Another application is in lossless image compression, where it is likely that some regions of an image contain similar pixel values. This is exploited in the JPEG lossless compression standard [33], which predicts the value of a pixel by considering the values of its neighbours. Greater compression can sometimes be achieved by compressing the differences between adjacent samples instead of the samples themselves. This is called *differential encoding*.

We can confound differential compressors by defining new variables $v_i^{(1)} = v_i - v_{i+1} + m - 1$ with domains $\{0, \ldots, 2(m-1)\}$ to represent the differences between adjacent solution variables (shifted to obtain non-negative values), and applying entropy constraints from Sections 2.1 and 2.2 to the $v_i^{(1)}$. We shall call these *differential* [*frequency, dictionary*] *entropy constraints*. We use the notation $\boldsymbol{v}^{(1)}$ because later on we shall consider differences of differences $\boldsymbol{v}^{(2)}$ and so on.

Adding a differential frequency constraint $\texttt{freq}(\boldsymbol{v}^{(1)}, 18, \mathbf{0}, \mathbf{10})$ and a differential dictionary constraint $\texttt{dict}(\boldsymbol{v}^{(1)}, 18, 3, 1)$ to the earlier constraints we get differences

jkilhmgnfoepdqcrbsbjkjilikhmhlgngmfofnepeodqdpcrcq
djkkhliimgoenfpejlhnfmhjklglgofmiiingjmhkhkjljijk

which has $\epsilon = 90$, and lex-least solution

0010203040506070809112213231424152516261727182819 2
933453543638373944648475568574958765966978677998 89

which has $\epsilon = 80$: both ϵ values indicate totally random sequences of the respective symbol sets. However, in some ways this solution still does not look very random: its initial values are 0, and roughly the first third of the sequence has a rather regular pattern. This is caused by our taking the lex-least solution, and by there being no problem constraints to complicate matters. In such cases we could use a SPREAD-style constraint [22] to prevent too many small values from occurring at the start of the sequence, or perhaps resort to randomised search. Note that on this artificial example randomised search usually finds high-entropy solutions even without entropy constraints, but it is not *guaranteed* to do so.

2.4 Using entropy constraints

By expressing the randomness condition as constraints we ensure that *all* solutions are incompressible by construction. Therefore the search method used to find the sequences does not matter and we can use any convenient and efficient search algorithm, such as backtrack search (pruned by constraint programming or mathematical programming methods) or metaheuristics (such as tabu search, simulated annealing or a genetic algorithm). As long as the method is able to find a solution it does not matter if the search is biased, unless we require several evenly distributed solutions. But in the latter case we could define a new problem $\mathcal{P}'$ whose solution is a set of solutions to the original problem $\mathcal{P}$, with constraints ensuring that the $\mathcal{P}$-solutions are sufficiently distinct.

All our entropy constraints are of only two types: `freq` and `dict`. Both can be implemented via well-known Constraint Programming global constraints, or in integer linear programs via reified binary variables, or in SAT via suitable clauses. (In our experiments of Section 3 we use the second method because our constraint solver does not provide the necessary global constraints.) We can relate them by a few properties (proofs omitted):

$$\begin{aligned}
\texttt{dict}(\boldsymbol{v}, m, k, t) &\Rightarrow \texttt{dict}(\boldsymbol{v}, m, k+1, t) \\
\texttt{dict}(\boldsymbol{v}^{(i)}, m, k, t) &\Rightarrow \texttt{dict}(\boldsymbol{v}^{(i-1)}, m, k+1, t) \\
\texttt{freq}(\boldsymbol{v}^{(i)}, m, \mathbf{0}, \boldsymbol{t}) &\Rightarrow \texttt{dict}(\boldsymbol{v}^{(i-1)}, m, 2, t)
\end{aligned}$$

From these we can deduce

$$\texttt{freq}(\boldsymbol{v}^{(i)}, m, \mathbf{0}, \boldsymbol{t}) \Rightarrow \texttt{dict}(\boldsymbol{v}, m, i+1, t)$$

which provides an alternative way of limiting k-gram occurrences. But higher-order differential constraints should be used with caution. For example we could use $\texttt{freq}(\boldsymbol{v}^{(2)}, m, \mathbf{0}, \mathbf{1})$ instead of $\texttt{dict}(\boldsymbol{v}, m, 3, 1)$ as both prevent the trigram

125 from occurring more than once. But the former is stronger as it also prevents trigrams 125 and 668 from both occurring: the trigrams have the order-1 differences (1,3) and (0,2) respectively, hence the same order-2 difference (2). If we do not consider 125 and 668 to be similar in any relevant way for our application then using the `freq` constraint is unnecessarily restrictive.

3 Applications

We consider two applications. Section 3.1 describes a known problem from experimental psychology, and Section 3.2 describes an artificial factory inspection problem where the inspection schedule must be unpredictable.

3.1 Experimental psychology

Experimental psychologists often need to generate randomised lists under constraints [10]. An example of such an application is word segmentation studies with a continuous speech stream. The problem discussed in [10] has a multiset W of 45 As, 45 Bs, 90 Cs and 90 Ds, and no two adjacent symbols can be identical. There is an additional constraint: that the number of CD digrams must be equal to the number of As. From this multiset must be generated a randomised list.

Generating such a list was long thought to be a simple task and a standard list randomisation algorithm was used: randomly draw an item from W and add it to the list, unless it is identical to the previous list item in which case replace it and randomly draw another item; halt when W is empty. But it was shown in [10] that this algorithm can lead to a large bias when the frequencies of the items are different, as in the problem considered. The bias is that the less-frequent items A and B appear too often early in the list, and not often enough later in the list. The bias effect is particularly bad for short lists generated by such randomisation-without-replacement methods. The bias can ruin the results of an experiment by confounding frequency effects with primacy or recency effects.

The solution proposed by [10] is to create *transition frequency* and *transition probability* tables, and to use these tables to guide sequence generation (we do not give details here). This is therefore a solved problem, but the authors state that *the correct ... table corresponding to the constraints of a given problem can be notoriously hard to construct*, and it would be harder to extend their method to problems with more complex constraints.

Generating such lists is quite easy using our method. For the above example we create a CSP with 270 variables v_i each with domain $\{0, 1, 2, 3\}$ representing A, B, C and D respectively. To ensure the correct ratio of items we use frequency constraints $\texttt{freq}(\boldsymbol{v}, 4, \langle 45, 45, 90, 90\rangle, \langle 45, 45, 90, 90\rangle)$. We add a constraint to ensure that there are 45 CD digrams. To ensure a reasonably even spread of values we add constraints $\texttt{freq}(\boldsymbol{v}_i, 4, \mathbf{0}, \langle 11, 11, 22, 22\rangle)$ to each fifth $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_5$ of $\boldsymbol{v}$. Finally we add constraints $\texttt{dict}(\boldsymbol{v}_i, 4, 5, 1)$ to each fifth.

Backtrack search turned out to be very slow so we use a local search algorithm, implemented in the constraint solver to take advantage of its filtering algorithms. The algorithm is not intended to be particularly competitive or interesting, but we describe it here for the sake of reproducibility. In an initial iteration it attempts to assign a value to each variable in turn using the ordering $v_1, v_2, \ldots$, and halts when domain wipeout occurs. We refer to this initial choice of value for each variable the *default*. In the next iteration it makes a few random assignment choices, but in most cases it choose the default. If any choice leads to domain wipeout then all alternatives are tried for that variable (but backtracking to other variables does not occur). If the new set of choices assign no fewer variables than the defaults then they become the new defaults. If all variables are assigned in an iteration then the algorithm has found a feasible solution and it halts.

The solution we found after a few tens of seconds is shown in Figure 1, with $\epsilon = 118$. In further tests local search achieved ϵ in the range 114–124, whereas without entropy constraints it achieved ϵ of 100–115.

```
CBDCDCADCBDBCBCABCDCBDCDBCABADCBCDCABDCBCADCDCDCACADAD
CDADCBDABDCDCBCDCDCDABCABDCDBCDBADCABDBDCACDCACDBCACDC
ADCDCACDCBDACDCADCDCBDCABCDBDBDACACDCDCDBDCACDBCDCADBD
CDCDADCACDACDCDCDCBDCDBDBDBDCDCACDBDBCDBCADCADADCDCADC
BCDBCDCDACACBCDCBCDBDCDCDCBCADCDCADCBDADCADACABDADBDAD
```

Fig. 1. A solution to the experimental psychology problem

[10] noted a systematic bias in the A:C ratio between fifths of the sequence: approximately 0.59, 0.58, 0.53, 0.49, 0.32. In ours we have 0.47, 0.50, 0.47, 0.44 and 0.59. Our approach has much in common with the Mix system used by psychologists but also important differences (see the discussion in Section 4).

3.2 Factory inspection

Next we consider an artificial factory inspection problem. Suppose an inspector must schedule 20 factory inspections over 200 days. We represent this by finite domain variables $v_1, \ldots, v_{20} \in \{1, \ldots, 200\}$. The inspection plan should appear as random as possible so that the owners can not predict when they will be inspected.

To make the problem more realistic we could add constraints preventing different factories from being inspected on certain days, or introduce an objective function. We do this below, but first we shall simply restrict all inspections to certain available days which are unevenly distributed through the year: 1–40, 70–100, 130–150 and 180–190. We order the v_i in strictly ascending order using a global constraint `ordered`($\boldsymbol{v}$).

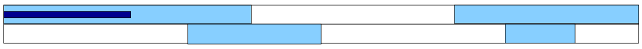
The lex-least solution is

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

with differences

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

All the inspections take place in the first 20 days, as is clearer if we visualise the schedule using two rows of 100 days each, the dark colour indicating scheduled inspections, the light colour indicating days in which an inspection could be scheduled, and white indicating days in which no inspection could be scheduled:



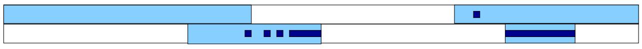
We could simply randomise the value ordering in each variable assignment during backtrack search, but this is unlikely to find the highest-entropy solutions. A typical solution found in this way is

73 139 142 144 146 147 148 149 150 180
181 182 183 184 185 186 187 188 189 190

with differences

66 3 2 2 1 1 1 1 30 1 1 1 1 1 1 1 1 1 1

and the schedule:

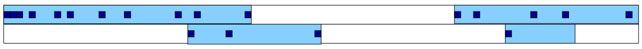


This solution also looks non-random. Because the variables are ordered, choosing each value randomly in turn causes clustering in the high values. We could randomise the search in a more clever way, for example by biasing earlier assignments to lower values, or branching under a different variable ordering. For such a simple problem this would not be hard to do, but the more complicated the constraint network is the harder this task becomes. Stochastic local search might typically find a high-entropy solution but, as pointed out in Section 1, randomised search alone is not enough so we shall apply entropy constraints.

As the v_i are all different we use difference variables. Applying entropy constraints $\texttt{freq}(\boldsymbol{v}^{(1)}, 198, \mathbf{0}, \mathbf{2})$ and $\texttt{freq}(\boldsymbol{v}^{(2)}, 395, \mathbf{0}, \mathbf{1})$ gives the lex-least solution

1 2 3 5 9 11 16 20 28 31 39 70 73 82 87 99 130 136 150 180

and the schedule



This is much more random-looking, indicating that an appropriate use of entropy constraints can yield higher-entropy solutions than random search, as mentioned in Section 1.

To confirm our intuition that this solution is sufficiently random we apply gzip. However, we do not compress the solutions in integer form as it contains spaces between numbers and uses multiple characters to represent integers: this helps to hide patterns from gzip so that its entropy estimate is poor. Instead we compress the binary schedule representations (as used above) of:

(i) the lex-least solution without entropy constraints

(ii) a randomised backtrack search solution
(iii) the lex-least solution with entropy constraints
(iv) mean results for random binary sequences of length 200 containing exactly 20 ones, without the restriction to certain ranges of days

We also estimate their *Approximate Entropy* (ApEn), a measure of the regularity and predictability of a sequence of numbers. ApEn was originally developed to analyse medical data [23] but has since found many applications. We use the definition given in [2] for a sequence n symbols from an alphabet of size m:

$$\mathrm{ApEn}(k) = \begin{cases} H(k) - H(k-1) & \text{if } k > 1 \\ H(1) & \text{if } k = 1 \end{cases}$$

where

$$H(k) = -\sum_{i=1}^{m^k} p_i \log_2 p_i$$

and p_i is the probability of k-gram i occurring in the sequence, estimated as its observed frequency f_i divided by $n - k + 1$. $H(1)$ is simply the Shannon entropy measured in bits, and ApEn(k) is a measure of the entropy of a block of k symbols conditional on knowing the preceding block of $k-1$ symbols. ApEn is useful for estimating the regularity of sequences, it can be applied to quite short sequences, and it often suffices to check up to ApEn(3) to detect regularity [2].

Results are shown in Table 2. The compression and ApEn results for (iii) are almost as good as those of (iv). This indicates not only that the inspections are unpredictable by the factory owners, but that the owners can not even detect in hindsight (by gzip and ApEn) the fact that the inspector had time constraints. Note that 0.47 is the theoretical ApEn(k) for all $k \geq 1$, for a binary source with probabilities 0.1 and 0.9 as in this example.

		ApEn(k)		
solution	ϵ	$k = 1$	$k = 2$	$k = 3$
(i)	29	0.47	0.03	0.03
(ii)	39	0.47	0.28	0.24
(iii)	54	0.47	0.46	0.42
(iv)	56	0.47	0.46	0.45

Table 2. Inspection schedule entropies

We apply a further statistical test of randomness to the binary representation: the *Wald-Wolfowitz runs test*. A *run* is an unbroken substring of 0s (or 1s) with preceding and subsequent 1s (or 0s) except at the start and end of the string. For example the string 01100010 contains five runs: 0, 11, 000, 1 and 0. A randomly chosen string is unlikely to have a very low or very high number of runs, and this can be used as a test of randomness. For a random binary sequence with

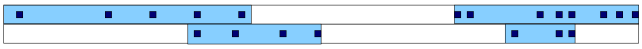
180 zeroes and 20 ones, we can be 95% confident that the sequence is random if there are between 33 and 41 runs. The lex-least solution has 36 runs so it passes the test. Thus the lex-least solution passes several tests of randomness. (One might ask why we did not also generate constraints to enforce ApEn and the runs test: we discuss this point in Section 5.)

This example is perhaps too simple as it contains only one constraint and no objective function. In fact if we replace the `ordered` constraint by an `alldifferent` constraint then local search without entropy constraints finds a high-entropy solution with $\epsilon = 55$, ApEn(1)=0.47, ApEn(2)=0.45 and ApEn(3)=0.45 — though backtrack search with randomised value ordering only achieves $\epsilon = 48$, ApEn(1)=0.47, ApEn(2)=0.46 and ApEn(3)=0.35. We should not have to manipulate the constraint model or choose the right solver to find high-entropy solutions, and there might be a good reason for using `ordered` instead of `alldifferent` (for example it breaks permutation symmetry), but to make the problem more interesting we now consider two alternative complications.

Firstly, suppose that we have some additional constraints:

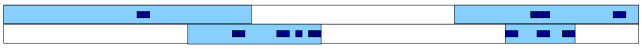
- no two inspections may occur on consecutive days: strengthen the `ordered` constraint to $v_{i+1} > v_i + 1$ ($i = 1 \ldots 19$)
- there must be at least 13 inspections in the first 100 days: $v_{13} \leq 100$
- there must be at least 2 inspections in the fourth time period: $v_{19} \geq 180$

Without entropy constraints local search now finds a lower-entropy solution with $\epsilon = 44$, ApEn(1)=0.47, ApEn(2)=0.46 and ApEn(3)=0.31, but with entropy constraints it finds this schedule:



with high entropy: $\epsilon = 57$, ApEn(1)=0.49, ApEn(2)=0.47 and ApEn(3)=0.47.

Secondly, instead of additional constraints suppose we have an objective function. Consider a scenario in which the factories are all in country A, the inspector lives in a distant country B, and flights between A and B are expensive. We might aim to minimise the number of flights while still preserving the illusion of randomness. To do this we could maximise the number of inspections on consecutive days. If we require an objective value of at least 10 then we are unable to find a solution under the above entropy constraints. We are forced to use less restrictive entropy constraints such as $\texttt{freq}(\boldsymbol{v}^{(1)}, 198, \mathbf{0}, \mathbf{10})$ and $\texttt{dict}(\boldsymbol{v}^{(1)}, m, 2, 1)$ yielding the following schedule by local search:



with 10 nights in a hotel, $\epsilon = 50$, ApEn(1)=0.47, ApEn(2)=0.38 and ApEn(3)=0.32. This illustrates the fact that imposing additional criteria might force us to accept lower-entropy solutions.

4 Related work

[30] proposed *statistical constraints* to enforce certain types of randomness on a solution: solutions should pass statistical tests such as the t-test or Kolmogorov-Smirnov. An inspection plan found in using the latter test is shown in Figure 2. If this is the only test that might be applied by an observer then we are done. However, the schedule exhibits a visible pattern that could be used to predict the next inspection with reasonable certainty. The pattern is caused by the deterministic search strategy used. It might be important to find an inspection schedule that is not predictable by visual inspection, or by a machine learning algorithm. In this case statistical tests are not enough and we must also enforce randomness.

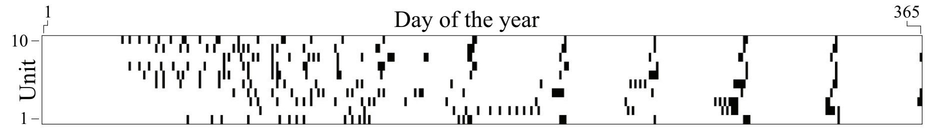


Fig. 2. A partially-random inspection plan that passes a statistical test

In the field of hardware verification SAT solvers have been induced to generate random stimuli: see for example [16] for a survey of methods such as adding randomly-chosen XOR constraints. Constraint solvers have also been used for the same purpose, a recent example being [19]. These approaches aim to generate an unbiased *set* of solutions, as do the methods of [6, 8, 9, 11, 12], whereas we aim to maximise the algorithmic entropy of a *single* solution. But (as pointed out in Section 2.4) we could obtain a similar result by defining a new problem $\mathcal{P}'$ whose solution is a set of solutions to the original problem $\mathcal{P}$, and add entropy constraints to $\mathcal{P}'$.

[3] describes a software system called Mix for generating constrained randomised number sequences. It implements a hand-coded local search algorithm with several types of constraint that are useful for psychologists, including constraints that are very similar to our `freq` and `dict` constraints (*maximum repetition* and *pattern* constraints respectively). However, no connection is made to Kolmogorov complexity or data compression, Mix does not use a generic constraint solver or metaheuristic, it does not use differential constraints (though it has other constraints we do not have), and it is designed for a special class of problem.

The `SPREAD` constraint [22] has something in common with our frequency constraints but with a different motivation. It balances distributions of values, for example spreading the load between periods in a timetable. It has efficient filtering algorithms but it does not aim to pass compression-based randomness tests.

Markov Constraints [21] express the Markov condition as constraints, so that constraint solvers can generate Markovian sequences. They have been applied to the generation of musical chord sequences.

5 Discussion

We proposed several types of entropy constraint to eliminate different types of pattern in a solution, leading to high-entropy solutions as estimated by compression algorithms and the Approximate Entropy function. These are complementary to statistical tests of the kind explored in [30]. All our constraints are based on well-known global constraints and can also be implemented in MIP or SAT. Note that instead of specifying bounds on the occurrences of symbols and k-grams we could allow the user to specify bounds on the Approximate Entropy $\mathrm{ApEn}(k)$ for various k. However, we believe that the former approach is more intuitive.

Using constraints to represent randomness makes it easy to generate random-looking solutions with special properties: we simply post constraints for randomness and for the desired properties, then any solution is guaranteed to satisfy both. However, applying entropy constraints is something of an art involving a compromise between achieving high entropy, satisfying the problems constraints and possibly optimising an objective function. Even with few or no problem constraints we must take care not to exclude so many patterns that no solutions remain, as Ramsey theory [25] shows that any sufficiently large object must contain some structure. In fact adding entropy constraints does not necessarily preserve satisfiability. If a problem has no sufficiently random-looking solutions then entropy constraints might eliminate all solutions. However, an advantage of this is that (as mentioned in Section 1) we can prove that no such solutions exist: this cannot be done with the randomised search approach. Alternatively we could take an optimisation approach by treating entropy constraints as soft constraints, and searching for the most random-looking solution.

Of course our solutions are only pseudorandom, not truly random. They were generated by restricting repeated symbols and k-grams in order to be incompressible to a certain class of data compression algorithms. It could be objected that they might fail other tests of randomness that are important to applications. Our response to this argument is: we can turn these other tests into additional constraints. For example if our solution in Section 3.2 had failed the Wald-Wolfowitz runs test, we could have added a constraint to ensure that it passed the test, as follows. Suppose we have a sequence of n binary numbers, with n_0 zeroes and n_1 ones ($n_0 + n_1 = n$). Under a normal approximation (valid for $n_0, n_1 \geq 10$) the expected number of runs is

$$\mu = 1 + \frac{2n_0 n_1}{n}$$

and the variance of this number is

$$\sigma^2 = \frac{2n_0 n_1(2n_0 n_1 - n)}{n^2(n-1)} = \frac{(\mu - 1)(\mu - 2)}{n - 1}$$

To test for randomness with 95% confidence we require that the observed number of runs R is within $\mu \pm 1.96\sigma$. To implement this test as a constraint on binary variables $v_1, \ldots, v_n$ we define new binary variables $b_i = \text{reify}(v_i = v_{i+1})$ and post a constraint

$$\mu - 1.96\sigma \leq \left(1 + \sum_{i=0}^{n-2} b_i\right) \leq \mu + 1.96\sigma$$

If we do not know the values of n_0 and n_1 in advance, the constraint implementation can create auxiliary integer variables n_0, n_1 and real variables μ, σ, and post additional constraints:

$$\begin{array}{rl} \sum v_i = n_1 & \qquad n_0 + n_1 = n \\ \mu n = n + n_0 n_1 & \qquad \sigma^2(n-1) = (\mu - 1)(\mu - 2) \end{array}$$

There is another possible objection to our approach — in fact to the whole idea of eliminating patterns in solutions. It can be argued that a solution with a visible pattern is statistically no less likely to occur than a solution with no such pattern, and that patterns are merely psychological artefacts. For example if we generate random binary sequences of length 6 then 111111 is no less random than 010110 because both have the same probability of occurring. Considering the latter to be "more random" than the former is a form of Gambler's Fallacy, in which (for example) gamblers assume that numbers with obvious patterns are less likely to occur than random-looking numbers. But if we wish to convince humans (or automated software agents designed by humans) that a solution was randomly generated then we must reject patterns that appear non-random to humans. This intuition is made concrete by the ideas of algorithmic information theory [5]. We do not expect all readers to agree with our view: randomness is a notoriously slippery concept [2] whose nature is beyond the scope of this paper.

There are several interesting directions for future work. We hope to find new applications, possibly with other patterns to be eliminated. We could try to devise randomness constraints using ideas from other literatures. One approach is to take randomness tests applied to number sequences and turn them into constraints. For example we might implement *spectral tests* [17] which are considered to be powerful. But they are complex and we conjecture that they are unlikely to lead to efficient filtering (though this is far from certain and would be an interesting research direction). Moreover, they seem better suited to very long sequences of numbers: far longer than the size of typical solutions to optimisation problems. For evaluating pseudo-random number generators there is a well-known set of simpler tests: the *Die-Hard Tests*,[3] later extended to the *Die Harder Tests*[4] and distilled down to three tests in [18]. However, these tests are also aimed at very long sequences of numbers, and again it is not obvious how to derive constraints from them.

[3] http://www.stat.fsu.edu/pub/diehard/

[4] http://www.phy.duke.edu/ rgb/General/dieharder.php

Acknowledgment

This publication has emanated from research supported in part by a research grant from Science Foundation Ireland (SFI) under Grant Number SFI/12/RC/2289. S. Armagan Tarim is supported by the Scientific and Technological Research Council of Turkey (TUBITAK).

References

1. K. R. Apt, M. Wallace. Constraint Logic Programming Using Eclipse. Cambridge University Press, 2007.
2. E. Beltrami. What Is Random? Chance and Order in Mathematics and Life. Copernicus, 1999.
3. M. van Casteren, M. H. Davis. Mix, a Program for Pseudorandomization. *Behaviour Research Methods* 38(4):584–589, 2006.
4. R. Cilibrasi, P. M. B. Vitányi. Clustering by Compression. *IEEE Transactions on Information Theory* 51(4):1523–1545, 2005.
5. G. J. Chaitin. Algorithmic Information Theory. Cambridge University Press, 1987.
6. K. Claessen, J. Duregård, M. H. Palka. Generating Constrained Random Data with Uniform Distribution. *12th International Symposium on Functional and Logic Programming, Lecture Notes in Computer Science* 8475:18–34, 2014.
7. C. C. Cutler. Differential Quantization for Television Signals. U. S. Patent 2,605,361, July 1952.
8. R. Dechter, K. Kask, E. Bin, R. Emek. Generating Random Solutions for Constraint Satisfaction Problems. *Proceedings of the 18th National Conference on Artificial Intelligence*, 2002, pp. 15–21.
9. S. Ermon, C. P. Gomes, B. Selman. Uniform Solution Sampling Using a Constraint Solver As an Oracle. *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, AUAI Press 2012, pp. 255–264.
10. R. M. French, P. Perruchet. Generating Constrained Randomized Sequences: Item Frequency Matters. *Behaviour Research Methods* 41(4):1233–1241, 2009.
11. E. Hebrard, B. Hnich, B. O'Sullivan, T. Walsh. Finding Diverse and Similar Solutions in Constraint Programming. *Proceedings of the 20th National Conference on Artificial Intelligence*, 2005.
12. P. van Hentenryck, C. Coffrin, B. Gutkovich. Constraint-Based Local Search for the Automatic Generation of Architectural Tests. *Proceedings of the 15th International Conference on the Principles and Practice of Constraint Programming* 2009, pp. 787–801.
13. K. Hinkelmann, O. Kempthorne. Design and Analysis of Experiments I and II. Wiley, 2008.
14. E. A. Housworth, E. P. Martins. Random Sampling of Constrained Phylogenies: Conducting Phylogenetic Analyses When the Philogeny is Partially Known. *Syst. Biol.* 50(5):628–639, 2001.
15. D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the IRE* 40:1098–1101, 1951.
16. N. Kitchen, A. Kuehlmann. Stimulus Generation for Constrained Random Simulation. *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, IEEE Press, 2007, pp. 258–265.

17. D. E. Knuth. The Art of Computer Programming volume 2: Seminumerical Algorithms (2nd ed.). Addison-Wesley 1981, pp. 89.
18. G. Marsaglia, W. W. Tsang. Some Difficult-to-pass Tests of Randomness. *Journal of Statistical Software* 7(3), 2002.
19. R. Naveh, A. Metodi. Beyond Feasibility: CP Usage in Constrained-Random Functional Hardware Verification. *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* 8124:823–831, 2013.
20. P. Ouellet, C.-G. Quimper. The Multi-Inter-Distance Constraint. *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 2011, pp. 629–634.
21. F. Pachet, P. Roy. Markov Constraints: Steerable Generation of Markov Sequences. *Constraints* 16(2):148–172, 2011.
22. G. Pesant, J.-C. Régin. SPREAD: A Balancing Constraint Based on Statistics. *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* 3709:460–474, 2005.
23. S. M. Pincus, I. M. Gladstone, R. A. Ehrenkranz. A Regularity Statistic for Medical Data Analysis. *Journal of Clinical Monitoring and Computing* 7(4):335–345, 1991.
24. W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling. Numerical Recipes in C. Cambridge University Press, UK, 2nd edition, 1992.
25. F. P. Ramsey. On a Problem of Formal Logic. *Proceedings London Mathematical Society* s2-30(1):264–286, 1930.
26. P. Refalo. Impact-Based Search Strategies for Constraint Programming. *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science Volume* 3258:557–571, 2004.
27. J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. *14th National Conference on Artificial Intelligence* pp. 209–215, 1996.
28. J.-C. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. *Proceedings of the 12th National Conference on Artificial Intelligence* Vol. 1, 1994, pp. 362–367.
29. J. J. Rissanen, G. G. Langdon. Arithmetic Coding. *IBM Journal of Research and Development* 23(2):149–162, 1979.
30. R. Rossi, S. Prestwich, S. A. Tarim. Statistical Constraints. *21st European Conference on Artificial Intelligence*, 2014.
31. K. Sayood. Introduction to Data Compression. Morgan Kaufmann, 2012.
32. C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal* 27(3):379–423, 1948.
33. G. K. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM* 34:31–44, 1991.
34. T. A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer* pp. 8–19, June 1984.
35. J. Ziv, A. Lempel. A Universal Algorithm for Data Compression. *IEEE Transactions on Information Theory* IT-23(3):337–343, 1977.
36. J. Ziv, A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory* IT-24(5):530–536, 1978.