Randomness as a Constraint

S. Prestwich, R. Rossi, S. A. Tarim [visiting!]

statistical constraints

first some background from our ECAI'14 paper *Statistical Constraints*...

a statistical constraint *exploits statistical inference to determine what assignments satisfy a given statistical property at a prescribed significance level*

we introduced the first two examples of statistical constraints embedding two well-known statistical tests: the t-test & the Kolmogorov-Smirnov test

we considered an inspection scheduling problem:

> *10 units to be inspected 25 times each over a planning horizon of 365 days, an inspection lasts 1 day & requires 1 inspector, 5 inspectors who can inspect at any given day, average inspection rate λ is 1 inspection every 5 days*

statistical constraint *inter-arrival times between inspections of the same unit should be approximately exponentially distributed*

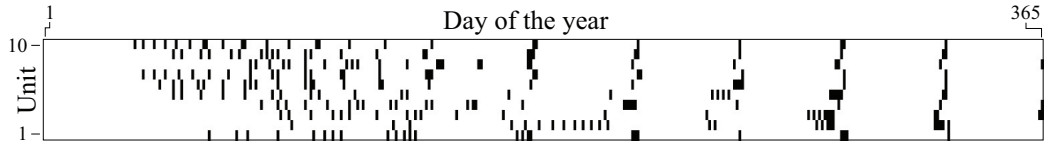
this mimics a "memoryless" inspection plan: the probability of being inspected at any time is independent of # past inspections

parametric Kolmogorov-Smirnov constraint: if the null hypothesis

> *intervals between inspections follows an exponential distribution with rate* λ

is rejected at significance level $\alpha = 0.1$ then the plan is infeasible

here's the inspection plan we found:



passes the test but has a visible pattern that could be used to forecast inspections!

it should "look more random"

this is the topic of the new work (submitted to CPAIOR'15)

random-looking solutions

for some applications we require a list of numbers, or some other data structure, that is (or looks) *random* while satisfying constraints

eg design of randomised experiments to avoid statistical bias, generation of random phylogenetic trees, quasirandom sequences for numerical integration, randomised lists without repetition for use in experimental psychology, random programs for compiler verification, random scheduling of inspections for unpredictability...

an obvious approach: use a randomised search strategy, eg SLS, GA, backtrack search with randomised value ordering

this can work, eg it can generate a random permutation of a list

but in general it has several drawbacks...

(1) if we want only random-looking solutions then the constraint model is not correct (the correctness of a constraint model should be independent of the search strategy used to solve it)

(2) randomised search can't prove that a random-looking solution doesn't exist, or prove that a solution is as random-looking as possible

(3) unless the randomised search is carefully designed it is likely to make a biased choice of solution

(4) even if we sample solutions in an unbiased way, there's no guarantee (even a probabilistic one) that such a solution will *look* random

- randomly-sampled *unconstrained* sequences are almost certain to appear random, but a *constrained* problem might have mostly regular-looking solutions

- optimising an objective function might lead to regular-looking solutions

(I'll show examples of both phenomena)

instead it would be useful to have available a constraint `israndom`$(\vec{v})$ that forces a vector $\vec{v}$ of variables to be random, which could simply be added to a constraint model

first 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 (its *Shannon entropy* is computed from the symbol probabilities)

but we need to quantify the randomness of a sequence

in *algorithmic* information theory, randomness can be viewed as a property of a specific data sequence

its *Kolmogorov complexity/algorithmic entropy* is defined as the length of the smallest algorithm that can describe it

eg 111111 may have the same probability of occurring as 101101 but it has lower entropy because it can be described more simply

algorithmic entropy formally captures the intuitive notion of whether a list of numbers "looks random"

this makes algorithmic entropy useful for our purposes!

(I'll refer to it simply as *entropy* by a slight abuse of language)

now we'd like a constraint of the form $\texttt{entropy}(\vec{v}, e)$ to ensure that entropy$(\vec{v}) \geq e$

unfortunately this is impossible because (algorithmic) entropy is uncomputable

instead we take a pragmatic approach: define constraints that eliminate patterns exploited by well-known data compression algorithms

a close relationship between entropy & compressibility: applying a compression algorithm to a sequence of numbers, & measuring the length of the compressed sequence gives an upper bound on the entropy of the original sequence

so by excluding readily-compressible solutions we hope to exclude low-entropy (non-random) solutions

entropy constraints

we need constraints to exclude low-entropy solutions, which we call *entropy constraints*

but what types of pattern should/can be excluded by constraints, how can the constraints be implemented, what filtering algorithms are available?

we'll address these problems...

(i) non-uniform distributions

data in which some symbols occur more often than others have non-uniform probability distributions

Huffman coding & arithmetic coding 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 & few bits for common symbols)

to eliminate this type of pattern from our solutions we define some simple entropy constraints

assume that all variables have the same domain $\{0, \ldots, m-1\}$ (easy to generalise to different domains)

given lower bounds $\vec{\ell} = \langle \ell_0, \ldots, \ell_{m-1} \rangle$ & upper bounds $\vec{u} = \langle u_0, \ldots, u_{m-1} \rangle$ on the frequencies of symbols $0, \ldots, m-1$ define a *frequency entropy constraint* $\texttt{freq}(\vec{v}, m, \vec{\ell}, \vec{u})$

can be directly implemented by a global cardinality constraint [Régin] $\texttt{GCC}(\vec{v}, \langle 0, \ldots, m-1 \rangle, \vec{\ell}, \vec{u})$

let's see the effect of these constraints

consider a CSP with 100 finite domain variables $v_0, \ldots, v_{99} \in \{0, 1, \ldots, 9\}$ & no problem constraints, only entropy constraints

I'll use the default backtrack search heuristic to find the *lex-least solution* which I take as a proxy for the lowest-entropy solution

it's the solution that's least under the obvious lexicographical ordering (order by increasing value of the first variable, breaking ties using the second variable etc, under the static variable ordering $v_1, v_2, \ldots$)

lex-least solution is 000 . . . but if we add $\texttt{freq}(\vec{v}, 10, \vec{0}, 1\vec{4})$ we get

```
00000000000000111111111111122222222222233333333
33333344444444444445555555555555566666666666677
```

we estimate its entropy by gzip compression: gzip doesn't necessarily give an accurate entropy estimate, but it's been successfully used for this

denoting the entropy of solutions by ϵ & measuring it in bytes, the lex-least solution has $\epsilon = 43$

000... has $\epsilon = 26$ so it's an improvement

but a random sequence of digits in the range 0–9 typically has ϵ values of 80–83 so it's far from random (as can be seen)

(ii) 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 `compress`, V.24 bis, GIF, PKZip, Zip, LHarc, PNG, gzip & ARJ, and use algorithms by Ziv & Lempel

they detect repeated k-grams & replace them by pointers to dictionary entries

eg 011101011100 contains two occurrences of the 5-gram 01110, which can be stored in a dictionary & 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: given an integer $k \geq 2$ & an upper bound t on the number of occurrences of all k-grams over symbols $\{0, \ldots, m-1\}$ in a vector $\vec{v}$ of variables, define a constraint $\texttt{dict}(\vec{v}, m, k, t)$

we call this a *dictionary entropy constraint*: it prevents more than t occurrences of **any** k-gram in $\vec{v}$ given m symbols

it can be implemented by the `Multi-Inter-Distance`$(\vec{x}, t, p)$ global constraint [Ouellet & Quimper] with $p = n-k+1$ (the number of $\vec{x}$ variables), $\vec{x} = \langle x_0, \ldots, x_{n-k} \rangle$

the $x_i = \sum_{j=0}^{k-1} m^j v_{i+j}$ are auxiliary variables with domains $\{0, \ldots, m^k - 1\}$ representing k-grams

this global constraint enforces an upper bound t on the number of occurrences of each value within any p consecutive $\vec{x}$ variables

in the special case $t = 1$ we can instead use `alldifferent`$(\vec{x})$

to test this idea on our artificial problem we add $\texttt{dict}(\vec{v}, 10, k, 1)$ for various k (but not `freq`):

k	lex-least solution	ϵ
50	00000000000000000000000000000000000000000000001	
	00000000000000000000000000000000000000000000000	29
25	00000000000000000000001000000000000000000000000	
	20000000000000000000003000000000000000000000000	35
12	00000000001000000000020000000003000000000040	
	00000000050000000000600000000070000000008000	48
6	00000100000200000300000400000500000600000700000080	
	00090000110000120000130000140000150000160000170000	60
3	00010020030040050060070080090110120130140150160170	
	18019021022023024025026027028029031032033034035036	71
2	00102030405060708091121314151617181922324252627282	
	93343536373839445464748495565758596676869778798890	80

the results show that as we reduce k the solution contains fewer obvious patterns

$k = 2$ gives a solution that (to gzip) is indistinguishable from a purely random sequence (80 bytes is within the range of ϵ for random sequences of this form)

(in this example there are sufficient digrams to avoid any repetition, but for smaller m this will not be true, & in the general case we might need to use larger t or larger k)

(iii) correlated sources

though gzip & 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 still have low entropy & noticeable patterns, eg this sequence of 100 integers in the range 0–9

01234567890246813579036914725804815926370516273849
94837261507362951840852741963097531864209876543210

compresses to 80 bytes, so is indistinguishable from a random sequence to gzip

yet it was written by hand & is certainly not random, obvious if we look at differences:

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

another eg: the high-entropy ($k = 2$) solution found above has differences

jjkilhmgnfoepdqcrbsbjjkilhmgnfoepdqcrcjjkilhmgnfoe
pdqdjjkilhmgnfoepejjkilhmgnfofjjkilhmgngjjkilhmhj

(using a–s as base-19 digits): these also look quite random, & gzip compresses them to 92 bytes which is typical of a random sequence of 99 symbols from a–s

yet they have a non-uniform distribution: *a* doesn't 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 & its variants

another application is in lossless image compression, where it is likely that some regions of an image contain similar pixel values, exploited in the JPEG lossless compression standard, which predict 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: *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) & applying entropy constraints to the $v_i^{(1)}$

we call these *differential* [*frequency*, *dictionary*] *entropy constraints*

adding $\texttt{freq}(\vec{v}^{(1)}, 18, \vec{0}, \vec{10})$ & $\texttt{dict}(\vec{v}^{(1)}, 18, 3, 1)$ to the earlier constraints we get differences

jkilhmgnfoepdqcrbsbjkjilikhmhlgngmfofnepeodqdpcrcq
djkkhliimgoenfpejlhnfmhjklglgofmiiingjmhkhkjljijk

which has $\epsilon = 90$ (but still no a!) & lex-least solution

00102030405060708091122132314241525162617271828192
93345354363837394464847556857495876596697867799889

which has $\epsilon = 80$: pretty random

yet in some ways this solution still does not look very random: its initial values are 0, & roughly the first third of the sequence has a rather regular pattern

this is caused by our taking the lex-least solution, & by there being no problem constraints to complicate matters — but this seems unavoidable in lex-least solutions

(but we could use a `SPREAD`-style constraint [Pesant & Régin] to prevent too many small values from occurring at the start of the sequence)

discussion

if even the lex-least solution has high entropy, we believe that other solutions will too

to test this we applied randomised search to the model with entropy constraints, & always obtained solutions that compressed to the same number of bytes (80)

by constraining a problem via entropy constraints we can restrict the search space to such solutions, or prove that no such solution exists

by expressing the randomness condition as constraints we ensure that *all* solutions are *incompressible by construction*

so the search method used to find the sequences doesn't matter and we can use any convenient & efficient search algorithm, eg backtrack search (pruned by constraint programming or mathematical programming methods) or metaheuristics (eg 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* — then 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` & `dict`

both can be implemented via well-known Constraint Programming global constraints, or in integer linear programs via reified binary variables

we can relate them by a few properties:

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

from these we can deduce

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

which is an alternative way of limiting k-grams, but these should be used with caution

eg we could use $\texttt{freq}(\vec{v}^{(2)}, m, \vec{0}, \vec{1})$ instead of $\texttt{dict}(\vec{v}, m, 3, 1)$: both prevent the trigram 125 from occurring more than once, but the former is stronger as it also prevents trigrams 125 & 668 from both occurring: the trigrams have the order-1 differences (1,3) & (0,2) respectively, hence the same order-2 difference (2)

application: experimental psychology

experimental psychologists often need to generate randomised lists under constraints, eg word segmentation studies with a continuous speech stream

a problem discussed in [French & Perruchet, 2009] has a multiset W of 45 As, 45 Bs, 90 Cs & 90 Ds

from this multiset must be generated a randomised list

an additional constraint: (#CD)=(#A)

generating such a list was long thought to be easy & a standard list randomisation algorithm was used:

> randomly draw an item from W & add it to the list, unless it is identical to the previous list item in which case replace it & randomly draw another item; halt when W is empty

but French & Perruchet showed that this algorithm is biased: less-frequent items A & B appear too often early in the list, not often enough later in the list

the bias can ruin the results of experiments

they solved this problem via *transition frequency* & *transition probability* tables to guide sequence generation

but they write: *the correct ... table corresponding to the constraints of a given problem can be notoriously hard to construct* — it would be harder to extend their method to problems with more complex constraints

generating such lists is quite easy using our method

we create a CSP with 270 variables v_i each with domain $\{0, 1, 2, 3\}$ representing A, B, C & D

to ensure the correct ratio of items we use frequency constraints $\texttt{freq}(\vec{v}, 4, \langle 45, 45, 90, 90\rangle, \langle 45, 45, 90, 90\rangle)$

we add a constraint to ensure that there are 45 CD digrams

for an even spread of values we add constraints

$$\texttt{freq}(\vec{v}_i, 4, \vec{0}, \langle 11, 11, 22, 22 \rangle)$$

to each fifth $\vec{v}_1, \ldots, \vec{v}_5$ of $\vec{v}$

we add constraints $\texttt{dict}(\vec{v}_i, 4, 5, 1)$ to each fifth

backtrack search turned out to be very slow so we use a simple local search algorithm

the solution we found after a few tens of seconds is:

```
CBDCDCADCBDBCBCABCDCBDCDBCA
BADCBCDCABDCBCADCDCDCACADAD
CDADCBDABDCDCBCDCDCDABCABDC
DBCDBADCABDBDCACDCACDBCACDC
ADCDCACDCBDACDCADCDCBDCABCD
BDBDACACDCDCDBDCACDBCDCADBD
CDCDADCACDACDCDCDCBDCDBDBDB
DCDCACDBDBCDBCADCADADCDCADC
BCDBCDCDACACBCDCBCDBDCDCDCB
CADCDCADCBDADCADACABDADBDAD
```

with $\epsilon = 118$ (& no bias)

in further tests local search achieved ϵ in the range 114–124, whereas local search without entropy constraints achieved ϵ of 100–115

factory inspection

suppose an inspector must schedule 20 factory inspections over 200 days

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't predict when they'll 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

for simplicity we just restrict all inspections to certain available days which are unevenly distributed through the year: 1–40, 70–100, 130–150 & 180–190

we order the v_i in strictly ascending order using a global constraint `ordered`$(\vec{v})$

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:

111111111111111111100000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000

we could simply randomise the value ordering in each variable assignment during backtrack search, but 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

& schedule

00000000000000000000000000000000000000000000000000
00000000000000000000100000000000000000000000000000
00000000000000000000000000000000000100101011111
00000000000000000000000000001111111110000000000

this solution has a very non-uniform distribution of values & is not very 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 wouldn't 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 randomised search alone isn't enough

so we apply entropy constraints

as the v_i are all different we use difference variables

applying entropy constraints

$$\texttt{freq}(\vec{v}^{(1)}, 198, \vec{0}, \vec{2}) \quad \texttt{freq}(\vec{v}^{(2)}, 395, \vec{0}, \vec{1})$$

gives lex-least solution

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

& schedule

1110100010100001000100000001001000000010000000000
0000000000000000001001000000010000100000000000010
0000000000000000000000000001000001000000000000001
0000000000000000000000000001000000000000000000000

much more random-looking: entropy constraints can yield higher-entropy solutions than random search

to confirm our intuition that this solution is random we apply gzip to:

(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 & predictability of a sequence of numbers

ApEn was originally developed to analyse medical data but has since found many applications

we use the definition given in [Beltrami] for a sequence n symbols from an alphabet of size m:

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

where

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

& 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, & 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, & it often suffices to check up to ApEn(3) to detect regularity

results:

solution	ϵ	ApEn(k) $k=1$	$k=2$	$k=3$
(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

the compression & ApEn results for (iii) are almost as good as those of (iv)

so the inspections are unpredictable by the factory owners, & the owners *can not even detect in hindsight* (by gzip & ApEn) the fact that the inspector had time constraints

(0.47 is the theoretical ApEn(k) for all $k \geq 1$, for a binary source with probabilities 0.1 & 0.9)

we apply a further statistical test of randomness to the binary representation: the *Wald-Wolfowitz runs test* (eg 01100010 contains five runs: 0, 11, 000, 1 & 0)

a randomly chosen string is unlikely to have a very low or very high number of runs, & this can be used as a test of randomness

for a binary sequence with 180 zeroes & 20 ones, we can be 95% confident that it's random if it has 33–41 runs

the lex-least solution has 36 runs so it passes this test too

finally, suppose that the factories are all in country A, the inspector lives in a distant country B, & flights between A & 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're forced to use less restrictive entropy constraints such as

$$\texttt{freq}(\vec{v}^{(1)}, 198, \vec{0}, 1\vec{0}) \quad \texttt{dict}(\vec{v}^{(1)}, m, 2, 1)$$

yielding the following schedule by local search:

000000000000000000001100000000000000000000
000000000000000000000000000011100000000001100
000000000000000000000000000000011000011010 11
000000000000000000000000001100011001100000000000

with 10 nights in a hotel, $\epsilon = 50$, ApEn(1)=0.47, ApEn(2)=0.38 & ApEn(3)=0.32

optimising an objective function, or adding a constraint, can lead to lower-entropy solutions

related work

apart from our statistical constraints paper, not too much (*any pointers?*)

the `SPREAD` constraint has something in common with our frequency constraints but with a different motivation: it balances distributions of values, eg spreading the load between periods in a timetable; for our purposes its feature of excluding outliers is undesirable

Markov Constraints [Pachet & Roy] express the Markov condition as constraints, so that constraint solvers can generate Markovian sequences; applied to generation of musical chord sequences

[Casteren & Davis] describe a software system called Mix for generating constrained randomised number sequences

- implements a hand-coded local search algorithm with several types of constraint useful for psychologists, some similar to our `freq` & `dict`

- but no connection made to Kolmogorov complexity or compression, doesn't use a generic constraint solver or metaheuristic, it doesn't use differential constraints, & designed for a special problem class

in hardware verification SAT solvers have been induced to generate random stimuli, eg [Kitchen & Kuehlmann] survey methods

CP solvers also used, eg [Naveh & Metodi]

but these generate an unbiased set of solutions, whereas we aim to maximise the algorithmic entropy of a single solution

(as mentioned above, we could find several solutions simultaneously by defining a new problem & applying entropy constraints)

conclusion

we proposed entropy constraints to eliminate patterns in a CP solution, giving high-entropy solutions as estimated by gzip & ApEn

our constraints are based on well-known global constraints but can also be implemented in MIP

using constraints to represent randomness makes it easy to generate random sequences with special properties: just post constraints for randomness & for the desired properties, then any solution satisfies both

but applying entropy constraints is something of an art involving a compromise between achieving high entropy, satisfying the problems constraints & 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

(Ramsey theory shows that any sufficiently large object must contain some structure)

possible objection (i)

our (pseudo-)random solutions might fail other tests of randomness

our response: turn those tests into entropy constraints too!

eg if our solution inspection schedule had failed the Wald-Wolfowitz runs test, we could have added a constraint to ensure that it passed the test...

suppose we have a sequence of n binary numbers, with n_0 zeroes & 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

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

& the variance of this number is

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

for randomness with 95% confidence the observed #runs must be within $\mu \pm 1.96\sigma$

define new binary variables $b_i = \text{reify}(v_i = v_{i+1})$ & post a constraint

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

(there's also a version for the case where we don't know the values of n_0 & n_1 in advance)

possible objection (ii)

(in fact to the whole idea of eliminating patterns in solutions)

a solution with a visible pattern is statistically no less likely to occur than a solution with no such pattern

eg 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?!

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

THE END