On an application of an run length encoding sub-scheme to data transmission recovery and error correction

I.N. Galidakis[1]
Agricultural University of Athens
Iera Odos 75 av.,
Athens 11855, Greece.

Abstract

We utilize the fixed points of a lossy compression run length encoding sub-scheme by embedding arbitrary messages in the dead blocks of these fixed points. The latter allows partial or total recovery after transmission of arbitrary data runs and pre/post-processing error correction[2][3].

1 Introduction

A popular lossless compression method is the RLE[4] scheme. Briefly, applied to base 10 for an n run of identical data bits[5], it stores repeated runs of the same bit as a bit count followed by the data bit that occurs in the repeating run. A variant of the RLE scheme can be programmed which ignores the consistency of the repeated sub-run bits and stores only the bit counts. This would make for a lossy compression scheme (Section 3), until we stumble at the mechanics of its fixed points (Section 4). If we embed any original message inside the missing (dead) bits of the repeating run, by encoding the communicated message as a fixed point data run stream we can offer partial or even total backwards recovery of the message at the receiving end and pre/post-processing to be used as a self(double)-check error corrector.

2 Bit Information Bottlenecks

We are receiving a total message n as a data run of bits one bit k at a time. What is intuitively a reliable measure of the *size* of the incoming $n \in \mathbb{N}$? The number of its bits, because when k bits complete, intuition gives a rough estimate for the size of n: When we see a typed decimal n for example, we can roughly guess its size, by looking at its total string length. For reading simplicity, we work with base 10 streams of bits/decimal digits. A *digit information bottleneck* will occur then, when:

$$n \sim k \cdot 10^{k-1} \tag{2.1}$$

Expression (2.1) for $k \in \{0, 1, 2, 3, ..., 9\}$ takes the following forms:

[1]Corresponding author: jgal@aua.gr
[2]2020 Mathematics Subject Classification: 68P20, 68P30, 68P01, 94A08.
[3]Keywords: data, recovery, error correction, post-processing, redundancy.
[4]Abbreviation for 'Run Length Encoding'.
[5]'bit' hencefrom is meant as a synonym for 'base-b digit'.

- $0 \cdot 10^{-1} = 0$
- $1 \cdot 10^{0} = 1$
- $2 \cdot 10^{1} \sim 2a$
- $3 \cdot 10^{2} \sim 3ab$
- $4 \cdot 10^{3} \sim 4abc$
- $5 \cdot 10^{4} \sim 5abcd$
- $6 \cdot 10^{5} \sim 6abcde$
- $7 \cdot 10^{6} \sim 7abcdef$
- $8 \cdot 10^{7} \sim 8abcdefg$
- $9 \cdot 10^{8} \sim 9abcdefgh$

where $a, b, c, d, e, f, g \in \{0, 1, 2, 3, ..., 9\}$. A *bottleneck digit* then, is a digit which counts exactly this many digits following, including itself in an incoming sequence of data bits. A *bottleneck block* is an entire block of decimal digits which starts with a bottleneck digit and includes all the digits that the bottleneck digit counts including itself. Note that 0 and 1 are bottleneck digits and bottleneck blocks simultaneously.

3 A Lossy Compression RLE Sub-scheme

The two definitions above of bottleneck digit and bottleneck block give rise to the lossy compression scheme LC for $n \in \mathbb{N}$: Compress n by picking only its bottleneck digits and discard the rest of the bottleneck block as dead bits. For example, if: $n = 837162538491827361524351526351552430192837162635 15$, the compressed number will be: $\mathrm{LC}(n) = 88653501935$. We write n in a convenient format below, so the scheme is easy to recognize:

$$n = \underline{8}3716253\ \underline{8}4918273\ \underline{6}15243\ \underline{5}1526\ \underline{3}51\ \underline{5}5243\ \underline{0}\ \underline{1}\ \underline{9}28371626\ \underline{3}51\ \underline{5} \quad (3.1)$$

The bottleneck digits are indicated by an underline and are the digits picked up. The bottleneck blocks are the digit blocks between the bottleneck digits. Sample Code 3 of the Appendix (using Maple [5]) gives an algorithm (LC) for this compression scheme, with input an arbitrary length list L consisting of decimal digits $k \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. For example:

```
L:=[8,4,1,3,2,4,5,8,6,7,5,4,3,2,5,6,7,8,
9,1,0,9,4,7,5,4,2,8,7,6,9,1];
LC(L);
[8,6,5,1,0,9,9]
```

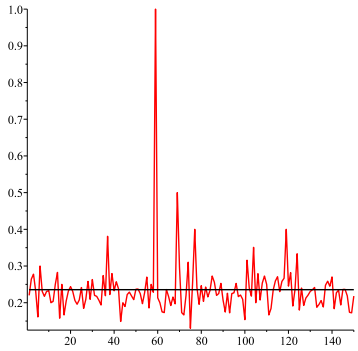

Figure 1: Compression ratio/efficiency of LC with list size 1-150 and trial size 150 (red) against measured average efficiency ~ 0.22342 (black), close to theoretical average efficiency $\sim 10/46 \sim 0.21739$.

Theorem 3.1 *The average compression efficiency of* LC *is:* $\sim 10/46$.

Proof: [6] Let L_i denote the number of times i is a leading bottleneck digit, i.e., one of the retained digits. For any given sequence of length N, $N = L_0 + L_1 + 2L_2 + 3L_3 + \cdots + 9L_9$. Suppose each digit i is equally likely to be a leader, i.e., that the L_i are approximately equal to some L, hence $N \sim 46L$. The number of retained digits is $L_0 + L_1 + \cdots + L_9 \sim 10L$, so the average compression efficiency must be around $10/46 \sim 0.21739$. $\square$

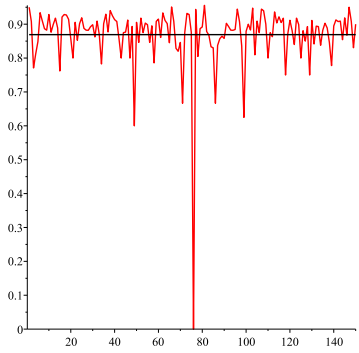

Figure 2: Compression/decompression error of LC with list size 1-150 and trial size 150 (red) against measured average error ~ 0.87 (black), corresponding to theoretical error $1 - 1/k$ for an average block size $k \sim 7.6$.

[6] Credit: James Waldby (sci.math forum, 2002).

Unfortunately LC is very lossy. Using a simple probabilistic distribution model ([1]), for a long data run of n bits with average block size k, it's clear that on average $\sim n/k$ bits will be retained (the bottleneck block headers) and therefore $\sim n - n/k$ bits will be lost (the bottleneck block contents sans their first bit). This gives an average error $1 - 1/k$.

Note that there *are* examples where the loss is minimal or very close to zero. Some obvious examples are the data streams: $n = \overbrace{d \dots d}^{k}$, consisting of k identical bits $d \in \{0, 1, \dots, 9\}$ or $n = d_1 \overbrace{0 \dots 0}^{d_1 - 1} d_2 \overbrace{0 \dots 0}^{d_2 - 1} \dots d_m \overbrace{0 \dots 0}^{d_m - 1}$, with $d_i \in \{0, 1, \dots, 9\}$, $i \in \mathbb{N}$, etc.

4 The Fixed Points of LC

Excepting the trivial cases shown in the previous Section, we can find many other numbers n such that $|0.n - 0.\,\mathrm{LC}(n)| < \epsilon$ for a given ϵ. A number n like this would be a non-trivial *fixed point* of this compression scheme[7], in a similar sense that 22 is a fixed point of RLE.

Using Sample Code 4 we generate some such fixed point data streams:

```
L:=[3,5,1,4,2];M:=[];
GFPLCM(L,32,M);
"Bad seed list: Cannot generate matching fixed point sequence"
```

Indeed! Note that $\mathrm{LC}(n) = [3, 4, \dots]$ and the second digit does not match. This initial digit seed sequence L therefore, cannot generate a fixed point stream. Try instead:

```
L:=[3,5,1,5,0];M:=[];
GFPLCM(L,32,M);
[3,5,1,5,0,x,x,x,1,5,x,x,x,x,0]
```

Verify compression:

```
LC(L); #compress
[3,5,1,5,0]
```

and that works. We can extend the generation to an arbitrary length fixed point stream by varying the parameter 'blocks' in GFPLCM (32, above). We call such numbers 2d (1d for data, 1d for redundancy) numbers. A 2d number broken down to its bottleneck blocks can be read in two ways, left to right and top to bottom using the bottleneck digits as anchors. This means: Read the following number from top to bottom and from left to right. Reading thus, the

[7] In an extended sense: In standard Analysis, for an appropriate function $f : \mathbb{R} \to \mathbb{R}$, a number $x \in \mathbb{R}$ is called a fixed point of f, iff it satisfies: $f(x) = x$. In this case we take $x = n$ to be a fixed point of f to allow for $\lim_{n \to \infty} |0.n - 0.\,\mathrm{LC}(n)| < \epsilon$, for given ϵ instead, to accommodate arbitrarily long data runs and their shorter compressions.

digits match the digits if you only read from top to bottom and only the first digit of the number in every row:

- $\underline{1}$
- $\underline{2}3$
- $\underline{3}09$
- $\underline{3}47$
- $\underline{0}$
- $\underline{9}12553173$
- $\underline{3}76$
- $\underline{4}705$
- $\underline{7}092646$
- $\underline{0}$
- $\underline{9}21148687$
- ...

For all such numbers, we have:

Corollary 4.1 *If $b_n = \{d_{n,1}, d_{n,2}, d_{n,3}, \ldots, d_{n,k}\}$ is a bottleneck block of size $|b_n| = k$, in a data run list $L = \{d_i\}_{i\in\mathbb{N}}$ of digits $d_i \in \{0, 1, 2, \ldots, 9\}$, then, for all $n \geq 1$:*

$$\begin{aligned} d_{n,k} &= d_{\left(\sum_{j=1}^{n-1} |b_j|+k\right)} \\ &= d_{|b_1|+|b_2|+\ldots+|b_{n-1}|+k} \\ &= d_{d_{1,1}+d_{2,1}+\ldots+d_{n-1,1}+k} \\ &= d_{d_1+d_2+\ldots+d_{n-1}+k} \end{aligned} \tag{4.1}$$

Proof: By construction, from the definition of fixed point of LC. □

5 Properties of Fixed Points of LC

A fixed point sequence data run has an fairly interesting property: It can recover itself backwards completely if we have enough digits of the sequence starting at a bottleneck digit of the original sequence provided the structural condition from Corollary (4.1) is met. Suppose we are receiving the sequence:

```
L=[3,5,1,5,0,0,7,6,1,5,2,6,9,9,0,0,7,1,6,
2,3,3,7,6,3,6,4,1,9,1,5,8,2,7,4]
```

one bit at a time and we have lost all bits prior to those from position 10 backwards and suppose that we also know that the bit at position 10 is a bottleneck. So we have the sub-run:

```
LT:=[5,2,6,9,9,0,0,7,1,6,2,3,3,7,6,3,6,4,1,9,1,5,8,2,7,4];
```

Compress the above using LC:

```
LT:=LC(LT);#compress LT
[5,0,0,7,6,1,5]
```

The digits left of 5 are digits leading the truncated sub-run, LT. So we add them to the list, up to suitable successive bottleneck digits, to the left, to form a new recovered sub-run:

```
LT:=[5,0,0,7,6,1,5,2,6,9,9,0,0,7,1,6,2,3,3,7,6,3,6,4,1,
9,1,5,8,2,7,4];
LT:=LC(LT);#compress again
[5,1,5,0,0,7,6,1,5]
```

The first two digits are the leading digits before our new truncated sub-run LT. We add those to the list, but we have no bottleneck digit, so the next bottleneck digit to the left must necessarily be a 3. Hence:

```
LT:=[3,5,1,5,0,0,7,6,1,5,2,6,9,9,0,0,7,1,6,2,
3,3,7,6,3,6,4,1,9,1,5,8,2,7,4]
```

If we compress again:

```
LT:=LC(LT);#compress
[3,5,1,5,0,0,7,6,1,5]
```

which means LT is a fixed point, so we have recovered the entire data run. It is now clear that the entire recovery is essentially a consequence of Corollary (4.1). A more involved example: Suppose an original data run is:

```
[9,5,1,5,0,9,8,6,2,5,2,6,9,5,|,1,5,2,8,1,0,0,9,0,
7,8,6,5,9,7,3,8,6,1,1,6,6,8,7,6,7,8,6,1,5,2,2,5,2,
7,6,0,2,7,6,4,2,5,3,9,9,7,4,6,7,0,9,4,2,5,7,1,4,4,
1,5,8,8,9,1,2,6,8,5,3,1,5,5,5,0,1,0,0]
```

Suppose that we have only the following sub-run[8]:

```
LT=[1,5,2,8,1,0,0,9,0,7,8,6,5,9,7,3,8,6,1,1,6,6,8,
7,6,7,8,6,1,5,2,2,5,2,7,6,0,2,7,6,4,2,5,3,9,9,7,4,6,7,0,9,
4,2,5,7,1,4,4,1,5,8,8,9,1,2,6,8,5,3,1,5,5,5,0,1,0,0]
```

First, compress via LC:

[8]Sub-runs always right of |. Recovered left of |.

```
LC(LT);
[1,5,0,9,8,6,2,5,2,6,9,5,|,1,5,2,8,1,0,0].
```

We pick the next block to the left of | and add it to the list:

```
LT:=[5,2,6,9,5,1,5,2,8,1,0,0,9,0,7,8,6,5,9,7,3,8,6,1,1,6,6,
8,7,6,7,8,6,1,5,2,2,5,2,7,6,0,2,7,6,4,2,5,3,9,9,7,4,6,7,
0,9,4,2,5,7,1,4,4,1,5,8,8,9,1,2,6,8,5,3,1,5,5,5,0,1,0,0];
LC(LT);#compress again
[5,1,5,0,9,8,6,2,|,5,2,6,9,5,1,5,2,8,1,0,0]
```

We can add a bottleneck digit 9 at the start, so then

```
LT:=[9,5,1,5,0,9,8,6,2,5,2,6,9,5,1,5,2,8,1,0,0,9,0,7,8,6,5,
9,7,3,8,6,1,1,6,6,8,7,6,7,8,6,1,5,2,2,5,2,7,6,0,2,7,6,4,2,
5,3,9,9,7,4,6,7,0,9,4,2,5,7,1,4,4,1,5,8,8,9,1,2,6,8,5,3,1,
5,5,5,0,1,0,0];
```

Check compression:

```
LC(LT);#compress
[9,5,1,5,0,9,8,6,2,5,2,6,9,5,1,5,2,8,1,0,0]
```

and $L = LT$, which means we have recovered the entire run successfully, so we are done.

6 Limitations

The above scheme has two problems: First, one needs to have a sufficiently long sub-run to generate the immediate left neighbors up to the first sub-run block. For example, if the sub-run in the previous example was:

```
LT:=[9,0,7,8,6,5,9,7,3,8,6,1,1,6,6,8,7,6,7,8,6,1,5,2,2,
5,2,7,6,0,2,7,6,4,2,5,3,9,9,7,4,6,7,0,9,4,2,5,7,1,4,4,
1,5,8,8,9,1,2,6,8,5,3,1,5,5,5,0,1,0,0];
```

Then:

```
LT:=LC(L);
[9,8,6,2,5,2,6,9,5,1,5,2,8,1,0,0]
```

which is barely a recovered sub-run left of starting block:

$[9, 0, 7, 8, 6, 5, 9, 7, 3, \ldots]$. Any other chosen sub-run *right* of $[9, 0, 7, \ldots]$ and the recovery would be incomplete. For example, if we only had the sub-run:

```
LT=[8,6,1,1,6,6,8,7,6,7,8,6,1,5,2,2,5,2,7,6,
0,2,7,6,4,2,5,3,9,9,7,4,6,7,0,9,4,2,5,7,1,4,
4,1,5,8,8,9,1,2,6,8,5,3,1,5,5,5,0,1,0,0]
```

then we get:

```
LT:=LC(L);
[8,6,2,5,2,6,9,5,1,5,2,8,1,0,0]
```

Note that this sub-run indeed consists of 'recovered' digits of the original run L, but it misses the in-between block: $[\ldots, 9, 0, 7, 8, 6, 5, 9, 7, 3, \ldots]$. We can estimate roughly an approximate bound for the least number of digits of a sub-run needed to recover the first block left of the sub-run we've got. Suppose the full run is $L = [[b_1], [b_2], \ldots, [b_n], [b_{n+1}], \ldots, [b_m]]$, consisting of m blocks $b_i, i \in \{1, 2, \ldots, m\}$. We assume we've only got the sub-run: $LT = [[b_n], [b_{n+1}], \ldots, [b_m]]$, consisting of $m - n + 1$ blocks $b_j, j \in \{n, n+1, \ldots, m\}$. The number of digits of the sub-run, is: $|LT| = S_R = \sum_{j=n}^{m} |b_j|$. The number of digits of the full-run, is: $|L| = S_T = \sum_{i=1}^{m} |b_i|$. So any sub-run gives a digit percentage: S_R/S_T of the full run. Now:

$$\frac{S_R}{S_T} = \frac{S_R}{\sum_{i=1}^{n-1} |b_i| + S_R} = \frac{1}{\left(\frac{\sum_{i=1}^{n-1} |b_i|}{S_R}\right) + 1} \tag{6.1}$$

Upon compression of LT, we get exactly the $m - n + 1$ digits sub-run: $\mathrm{LC}(LT) = LT' = [b_n, b_{n+1}, \ldots, b_m]$, therefore $|LT'|$ needs to be at least equal to the number of (recovered) digits left of LT. But the latter is: $\sum_{i=1}^{n-1} |b_i|$, hence equation (6.1) becomes:

$$\frac{S_R}{S_T} = \frac{1}{\left(\frac{m-n+1}{S_R}\right) + 1} \tag{6.2}$$

Assuming an average block length $|b_j| \sim 5$ for the sub-run (LT) we've got, we get:

$$\frac{S_R}{S_T} \sim \frac{1}{\left(\frac{m-n+1}{\sum_{j=n}^{m} 5}\right) + 1} = \frac{1}{\left(\frac{m-n+1}{5(m-n+1)}\right) + 1} = \frac{1}{\frac{1}{5} + 1} \sim 83.3\% \tag{6.3}$$

The above means that we shouldn't expect reliable recovery of the immediate left block of LT, unless we've got at least $\sim 83.3\%$ of the total run[9]. Unfortunately, although we can count $|LT|$ exactly[10], there's no way to know how much of the L we've got in a lossy transmission system. This necessitates that either the original message L contains an encoding of its total length, either

[9] For an average block length ~ 5 in the received data run or for a data run with block lengths approximately equi-distributed with average 5. For different average block length data runs in the receiver, this percentage will obviously vary from 2/3 to 9/10 as a function of average block length 2 to 9.

[10] That's obviously the length of the receiver's data.

at the beginning or at the end or that the receiver has received $|L|$ correctly separately. In either case, once a receiver knows $|L|$ and calculates $|LT|$, one can calculate the expected reliability of recovery from (6.3).

Second, the above scheme doesn't always guarantee a unique recovery: Consider the streams[11]:

- 7254321 27547314509391281217877339505424056777653234214416567 35
- 35254321 27547314509391281217877339505424056777653234214416567 35
- 44254321 27547314509391281217877339505424056777653234214416567 35

If the stream is truncated at the bottleneck point 2754..., we get:

```
L:=[2,7,5,4,7,3,1,4,5,0,9,3,9,1,2,8,1,2,1,7,8,
7,7,3,3,9,5,0,5,4,2,4,0,5,6,7,7,7,6,5,3,2,3,4,
2,1,4,4,1,6,5,6,7,3,5];
LC(L);
[2,5,4,3,2,1,|,2,7,5,4,7,3,1,4,5]
```

Now the above has problems. We can pick up just the digit (1), the two digits (2,1), the three digits (3,2,1), the four digits (4,3,2,1), the five digits (5,4,3,2,1) or the digits (7,2,5,4,3,2,1) and continue the algorithm to give us different recovers with an updated stream.

If we pick the bottleneck digit (7) along with the block (2,5,4,3,2,1), we get the first sequence above. If we pick the block (5,4,3,2,1), we get:

```
L:=[5,4,3,2,1,2,7,5,4,7,3,1,4,5,0,9,3,9,1,2,8,1,
2,1,7,8,7,7,3,3,9,5,0,5,4,2,4,0,5,6,7,7,7,6,5,3,
2,3,4,2,1,4,4,1,6,5,6,7,3,5];
LC(L);
[5,2,|,5,4,3,2,1,2,7,5,4,7,3,1,4,5]
```

Add the bottleneck digit (3) and we get the second sequence above. If we pick the block (4,3,2,1), we get:

```
L:=[4,3,2,1,2,7,5,4,7,3,1,4,5,0,9,3,9,1,2,8,1,
2,1,7,8,7,7,3,3,9,5,0,5,4,2,4,0,5,6,7,7,7,6,5,3,
2,3,4,2,1,4,4,1,6,5,6,7,3,5];
LC(L);
[4,2,5,|,4,3,2,1,2,7,5,4,7,3,1,4,5]
```

Add the bottleneck digit (4) and we get the third sequence above. Unfortunately such cases are pathological and cannot predict a unique sequence, since there's no way to choose the next (previous) block uniquely. Such fixed point data runs are called *ambiguous*.

Now verify that any fixed point data run which contains *nested* bottleneck blocks must necessarily be an ambiguous fixed point. A *nested bottleneck block*

[11] Credit: Timothy Little (sci.math forum, 2002)

is a block of length k in which counting from right to left and from 1 to k, at least two entries in the block equal the counting index. For example, the following are nested bottleneck blocks:

- 987697777
- 59821
- 4731
- 4399
- 329

Fixed points which contain no nested bottleneck blocks exist. We therefore adjust procedure GFPLCM to generate digits which are not equal to their counting index from right to left inside their initial bottleneck blocks[12].

Corollary 6.1 *A non-ambiguous fixed point of* LC *predicts a backwards unique sequence.*

Proof: Choose a sufficiently long sub-run with a starting bottleneck block in order to recover more digits towards the beginning of the stream. Because the fixed point contains no nested bottleneck blocks, the compression process cannot extract a stream of digits which contain nested bottlenecks, hence the recovery process is unique and so the recovered stream is backwards unique. □

7 Error Correction and Partial Recovery in LC

Corollary 6.1 allows partially or totally recovered transmission, provided nested bottlenecks do not occur. A message can be encoded in the $d-1$ digits inside the bottleneck blocks following the bottleneck digit $d > 1$. The block headers will then serve as redundancy bits and the block contents will serve as the data bits. The message is then transmitted. If the message is long enough and new bits keep coming in, if the receiver missed a small initial sub-run segment, by receiving enough bottleneck blocks towards the end of the message bit stream he/she can recover the original all the way to its beginning.

Using the routines in the Appendix one may choose to transmit simple messages - up to using the bit codes (0..9). For example, let the message be[13] 1331415489975:

```
M:=[1,3,3,1,4,1,5,4,8,9,9,7,5];#message to be transmitted
```

The transmitter chooses a seed for fixed point encoding. Say:

[12]This modification already exists in GFPLCM of the Appendix: Encoded messages M through fixed point sequences, force a 0 place-holder inside the data run stream if the message's bit happens to generate a nested bottleneck block.

[13]For the reader's convenience, procedures for converting between lists of bits and big natural numbers are provided in the Appendix, as L2N and N2L.

```
L:=[3,1,4,1,4,6,3];
```

The transmitter now generates a fixed point data stream using GFPLCM from Sample Code 5[14]

```
G:=GFPLCM(L,30,M);
[3,1,4,1,4,6,3,0,1,4,1,3,3,6,1,4,1,5,4,3,8,9,0,1,4,9,7,5,1]
```

The transmitter first checks that it's a fixed point stream for LC:

```
LC(G);#compress G
[3,1,4,1,4,6,3,0,1,4,1]
```

And it is. The transmitter then transmits G, along with its size ($|G| = 29$) as a separate message. Suppose now that because of an unpredictable transmission error the receiver receives only 29 along with the data run: 6141543890149751 of size 16. The average block size in the received message is: $(6+3+0+1+4+1)/6 \sim 2.5$. Equation (6.3) for a receiving data run of average block size 2.5 gives a minimum required received data run size of 71.4% of the total transmission. Since $16/29 \sim 55\% < 71.4\%$, he immediately concludes that recovery of at least one block left of G is unreliable. So he waits for another transmission that includes at least one more backwards block. If the next received transmission happens to be for example: 41336141543890149751 of size 20, $20/29 \sim 68\%$ and the average block size of the received message is: 3.4, with a required minimum percentage for reliable recovery 77.2%, so he now knows he has a better chance of at least one block's recovery so tries again. And this time it works. First, he compresses the received message:

```
LT := [4,1,3,3,6,1,4,1,5,4,3,8,9,0,1,4,9,7,5,1]
LC(LT);
[4,6,3,0,1,4,1]
```

The emergence of the two blocks 4630 and 1 are now obvious. Adds these as a header to LT and compresses again:

```
LT := [4,6,3,0,1,4,1,3,3,6,1,4,1,5,4,3,8,9,0,1,4,9,7,5,1]
LC(LT);
[4,1,4,6,3,0,1,4,1]
```

The emergence of the block 41 is now obvious and it is obviously incomplete. There's only one choice for its header: 3, so LT now becomes:

```
LT:=[3,4,1,4,6,3,0,1,4,1,3,3,6,1,4,1,5,4,3,8,9,0,1,4,9,7,5,1]
```

One more compression:

```
LT:=[3,4,1,4,6,3,0,1,4,1,3,3,6,1,4,1,5,4,3,8,9,0,1,4,9,7,5,1];
LC(LT);
[3,1,4,1,4,6,3,0,1,4,1]
```

[14]Note the 'bad' first bit '1' in M's first digit for this fixed point seed, which gives its place to the 0 placeholder.

and the message is fixed. The latter means that recovery is complete. Now he extracts the intended message:

```
G:=[3,4,1,4,6,3,0,1,4,1,3,3,6,1,4,1,5,4,3,8,9,0,1,4,9,7,5,1];
M := GFPLCME(G,29);
M:=[4,1,6,3,0,1,3,3,1,4,1,5,4,8,9,9,7,5]
```

And now the original message 1331415489975 is contained in M. Unfortunately, the recovered message M is a super-set of the actual message, since there's no way to know how far M starts within the encoded G[15].

8 Encoder Variants

An alternate variant[16] which uses *fixed length* bottleneck blocks can be constructed, which avoids some of the complexity of the scheme in the previous sections[17]. For illustrative purposes we describe the encoding using blocks of size 5. Suppose M is:

6358410218638717217992072132791852594419173322 64.

$|M| \equiv 0 \mod 4$, so the message can be encoded into exactly 12 blocks of length 5. Each block header (previously bottleneck digit) - except for the first block, is a duplicate of the digits from the start, in sequence. This gives:

- 6358
- 6 4102
- 3 1863
- 5 8717
- 8 2179
- 6 9207
- 4 2132
- 1 7918
- 0 5259
- 2 4419
- 3 1733
- 1 2264

[15] Unless the transmitter also transmits the seed used for encoding as a separate message.

[16] Credit: Timothy Little (sci.math forum, 2002).

[17] Implements simpler code since it avoids separate seeds for fixed-point message encoding, provides exact certainty measure for recoverability, avoids unnecessary calculations within indexes with Corollary (4.1) and extracts M exactly.

One verifies that any sub-run longer than the minimum required percentage starting at a block boundary recovers the entire sequence backwards completely and unambiguously. For example: Having only the sub-run: 31863 58717 82179 69207 42132 17918 05259 24419 31733 12264, the recovery process[18] will generate the sequence: 358 64102|31, so updating the sequence to include the new 5-block, we have: 64102 31863 58717 82179 69207 42132 17918 05259 24419 31733 12264 and the new recovery process will generate the sequence: 6358|64102 31 and the whole data stream is again completely recovered. The original message M is extracted through GFPFBLCME[19].

Note that in this case Corollary 4.1 implies that the data bits satisfy:

$$d_n = d_{\left(1+4+\sum_{i=1}^{n-1} 5\right)} = d_{1+4+5(n-1)} = d_{5n} \tag{8.1}$$

From Equations (6.2) and (6.3), it is clear that the ratio of received over total bits for this kind of encoding with block size b needs to be at least:

$$p_{need} = \frac{1}{\frac{1}{b}+1} \tag{8.2}$$

while a message M with length $|M| = n$ bits, increases in size by n/b through the encoding, so the ratio of increase over message length n is $(n + (n/b))/n = 1/p_{need}$. Figure 3 shows the two ratios in a combined graph.

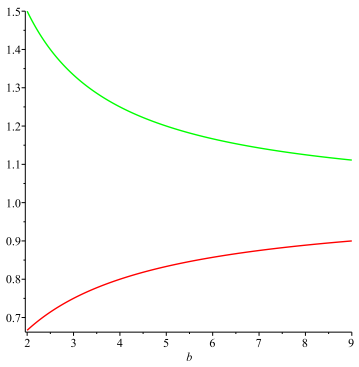

Figure 3: Needed received ratio for maximum likelihood of reliable backwards recovery p_{need} (red) and message length increase $1/p_{need}$ (green) due to redundancy bits as a function of block size b.

[18] through FBLC$(L, 5)$ in the Appendix

[19] Message extraction from a recovered run needs a null bit to patch the length of the first block of the recovered data run, so the data run will be: $LT := [x, 6, 3, 5, 8, 6, 4, 1, 0, 2, 3, 1, \ldots]$ and then the message extraction call is: $M :=$ GFPFBLCME$(L, 5)$.

9 More General Messages

All the above, for fairly simple messages with data bits ranging in 0..9. However LC or FBLC are not dependent upon bit-size, so one may well use bits with wider ranges. This would require that the transmission/reception gate has a way to differentiate between successive bits, which is usually done via an intervening characteristic sentinel bit, such as 0 or any pre-excluded encoder bit. If the previous does not apply and one's limited to only $(0..b)$ range bits, first encode M into its decimal equivalent M^* using base $b+1$ encoding and then transmit that[20].

Unfortunately, using greater range bits results in larger bottleneck block sizes, hence also in increased p_{need} and that reduces the likelihood for reliable backwards recovery, a problem which cannot be addressed easily, unless a very high percentage of the transmitted run is received reliably. A more workable compromise can be reached if the block size is fixed, as in Section 8.

10 Results and Discussion

Contrary to RLE which is lossless and highly efficient when the repeated block count is large ([2]), it is clear that LC as a compression scheme is fairly useless since the average error is very high and not easily controlled. As far as it being a lossy transmission recovery method, as already said, its recovery reliability is obviously at least p_{need}, which depends on block size d, which in turn is maximized for $b = 2$. In other words, for greater b, it tends to become useless, as the ratio of received to total message length tends to 1, Note however, that even in such extreme cases[21] it can still serve as an error correction double-check post-processor for the transmitted message: Any bit errors that may have crept in the received data run due to faults in transmission, and the received data run will fail the fixed-point property, since LC performs its own parity check both backwards and forward through the block headers which serve as redundancy bits[22]. As such, it can be used in conjunction with other error-correction methods which have higher redundancy ([4], [3]) to improve error-correction or validate otherwise correctly extracted encrypted data messages ([6]).

[20] For example, if only letters from the English alphabet are allowed (base 26) and the message is: $M =$'helloworld', then $M^* = \sum_{i=0}^{25} d_i \cdot 26^i$, with $d_i \in \{\mathrm{Ord}(h), \mathrm{Ord}(e), \mathrm{Ord}(l), \ldots\}$. After message recovery decode M^* to base 26 to recover the full-bit message M, etc.

[21] I.e., when for example the receiver has received a data run of the same length as that sent by the transmitter.

[22] It's clear for example that if the received data run does not satisfy Corollary 4.1 or equations (8.1) depending on case, the message has been corrupted one way or another.

11 Acknowledgements

The author is grateful to Tim Little and James Waldby for their respective contributions. Also to the University of Crete's Mathematics for 2000 Program Initiative, which allowed the author access to the Maple package, without which implementation wouldn't be possible.

12 Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Appendix

Sample Code 1: Convert number to a list of digits

```
N2L:=proc(n)
local d,L,RL,Llen,m;
m:=n;L:=[];RL:=[];
while m>0 do
 d:=10*frac(m/10);
 RL:=[op(RL),d];
 m:=floor(m/10);
od;
Llen:=nops(RL); #reverse list
for m from 1 to nops(RL) do
 L:=[op(L),RL[Llen-m+1]];
od;
L;
end:
```

Sample Code 2: Convert list to number

```
L2N:=proc(L)
local k;
sum(L[k]*10^(nops(L)-k),k=1..nops(L));
end:
```

Sample Code 3: Compress list L:

```
LC:=proc(L)
local i,nL,c;
nL:=[];c:=1;
while c <= nops(L) do
 nL:=[op(nL),L[c]];
 if L[c]<>0 then
  c:=c+L[c];
```

```
 else
  c:=c+1;
 fi;
od;
nL;
end:
```

Sample Code 4: Fixed point encoder for message M

```
GFPLCM:=proc(sL,n,M)
local sG,cL,c,m,match,pickd,pickmd,bL;
if evalb((0 in sL) or (1 in sL)) then
 print("Bad seed list: Contains 0s or 1s");
fi;
if sL[1]>nops(sL) then
 print("Bad seed list size:
  List size needs to be at least:",sL[1]);
 return;
fi;
cL:=LC(sL);
match:=true;
for m from 1 to nops(cL) do
match:=match and (cL[m]=sL[m]);
if not match then print("Bad seed list:
 Cannot generate matching fixed point sequence");
 return;
fi;
od;
sG:=sL;
if nops(sL)<n then
 c:=1;pickd:=1;pickmd:=1;
 while c<= nops(sL) do
  if sG[c]<>0 then
   c:=c+sG[c];
  else
   c:=c+sG[c]+1;
  fi;
  pickd:=pickd+1;
 od;print("finished parsing sL");
 for m from nops(sL)+1 to c-1 do
  if nops(M)>0 and pickmd<=nops(M) then
   if M[pickmd]-1<>c-m then
    sG:=[op(sG),M[pickmd]];
    pickmd:=pickmd+1;
   else
    sG:=[op(sG),0];
   fi;
```

```
 else
  sG:=[op(sG),x];
 fi;
od;
while c<n and type(sG[pickd],nonnegint)=true do
 if c>nops(sG) then #add header digit
  sG:=[op(sG),sG[pickd]];
  for m from c+1 to c+sG[c]-1 do#add trailing(data) for block
  if m<=n then
   if nops(M)>0 and pickmd<=nops(M) then
    if M[pickmd]-1<>c-m then
     sG:=[op(sG),M[pickmd]];
     pickmd:=pickmd+1;
    else
     sG:=[op(sG),0];
    fi;
   else
    sG:=[op(sG),x];
   fi;
  fi;
 od;
 fi;
 if sG[c]<>0 then
  c:=c+sG[c];
 else
  c:=c+sG[c]+1;
 fi;
 pickd:=pickd+1;
od;
fi;
sG;
end:
```

Sample Code 5: Fixed point encoding message extractor

```
GFPLCME:=proc(sL,n)
local c,m,pickd,M;
M:=[];
c:=1;pickd:=1;
while c<n do
 for m from c+1 to c+sL[c]-1 do#pick message from block
  if m<=n then
   M:=[op(M),sL[m]];
  fi;
 od;
 if sL[c]<>0 then
  c:=c+sL[c];
```

```
 else
  c:=c+sL[c]+1;
 fi;
 pickd:=pickd+1;
od;
M;
end:
```

Sample Code 6: Fixed block length LC compression

```
FBLC:=proc(L,blockLen)
local nL,c;
nL:=[];c:=1;
while c<=nops(L) do
nL:=[op(nL),L[c]];
c:=c+blockLen;
od;
nL;
end:
```

Sample Code 7: Fixed block length message encoder

```
GFPFBLCM:=proc(L,blockLen)
local sL,c,pickd;
sL:=[];pickd:=0;
for c to nops(L) do
 if c+pickd mod blockLen = 0 then
  pickd := pickd+1;
  sL:=[op(sL), sL[pickd]]
 fi;
 sL:=[op(sL), L[c]];
od;
sL;
end:
```

Sample Code 8: Fixed block encoding message extractor

```
GFPFBLCME:=proc(L,blockLen)
local sM,c;
sM:=[];
for c to nops(L) do
 if c-1 mod blockLen <> 0 then
  sM:=[op(sM),L[c]]
 fi;
od;
end:
```

References

[1] J. Capon, *A probabilistic model for run-length coding of pictures*, IRE Transactions on Information Theory **5(4)** (Dec. 1959), 157–163.

[2] S. Golomb, *Run-length encodings (corresp.)*, IEEE Transactions on Information Theory **12(3)** (Jul. 1966), 399–401.

[3] D. Huffman, *A method for the construction of minimum redundancy codes*, Proc. IRE **40** (Sep. 1952), 1098–1101.

[4] V. Pless, *Introduction to the Theory of Error-Correcting Codes*, John Wiley and Sons, New York, Chichester, Brisbane, Toronto, 1989.

[5] D. Redfern, *The Maple Handbook*, Springer-Verlag, New York, 1996.

[6] C.A.Henk van Tilborg, *An Introduction to Cryptology*, Kluwer Academic Publishers, 1988.