K-Zeros Run Length Limited Encoding

Lior Cohen **Hadas Abraham**

Problem Definition:

Several problems are linked to data that is coded in such a way that long runs of the same binary values are written consecutively.

For example, with a long string of zeros, there's no way for the disk drive's controller to know the exact position of the read head, and thus no way to know exactly how many zeros there are.

To prevent this problem, there are algorithms that allow us to code such data by limiting the size of zero-windows; this makes it possible for the drive controller to stay synchronized.

Our goal is to implement such coding.

Classic Zeros Run Length Limited Encoding:

Given a word $w \in \{0,1\}^n$ we would like to output an encoded word with zero runs no longer than $k = \lceil \log n \rceil + 1$.

We will first initialize a word $y = w1$ (1 is appended to the input word – we'll call it the *One Flag*). Then we iterate over the indices of y that correspond to the indices of the input word w. If we encounter an index i in which a k zero run starts, we cut the run and add $bin(i)0$ at the end of the word, where $bin(i)$ is the binary representation of the index we encountered (where its size is exactly k - 1).

*The encoded word has one redundancy bit.

Decoding an encoded word $w \in \{0,1\}^{n+1}$, would be cutting its k rightmost bits up until we encounter the *One Flag*. Each iteration, we will append k zero run at the index found at the block that we cut, where the index would be inferred by converting $bin(i)$ to its decimal value. Once we encounter the *One Flag* we remove it and return the word.

Correctness: A proof was given in the lectures.

Complexity: $O(n)$ as mentioned in [1].

K-Zeros Run Length Limited Encoding:

An extension of the classic algorithm would be encoding a word such that there would be no zero runs longer (or equal) than k, where k is a an integer of the user's choice.

Encoding:

Given a word $w \in \{0,1\}^n$ and an integer k, we would want to split w to blocks with length of n' where n' is defined such that $k = \log n' + 1$. Our main goal is encoding a word with the minimal redundancy bits possible. Our algorithm will add one redundancy bit for each block that is being encoded (every B_i plus the block of the first n' bits).

Note that n' = $\max\{i \in N \mid k = \lceil \log i \rceil + 1\} = 2^{k-1}$, thus we guarantee that both minimal number of blocks are encoded, and the last block will have minimal size.

We will first initialize an empty word z. Then, we will start by encoding a block of the first n' bits. Following that, we will split the rest of the word to $n' - k$ sized blocks (note that the last block might be of size lower than $n' - k$). let B_i be the $i - th$ block of this split. Then, every iteration on i, we will remove the k last bits of z and append it to the block B_i, then encode it and append the encoded word to z.

Correctness:

The correctness of the classic algorithm ensures that words of size x will have zero-runs no longer than $\log x$. In our algorithm, each word (last k bits of z appended to some block B_i) we use the classic algorithm at is of size n' (except, maybe, the last one which is of a smaller size), thus each encoded word has no k-or-longer zero-runs. We did not append the encoded words as they are, because k-or-longer zero-runs could appear at some window that contains two consecutive words. By appending the last k bits of z to every word before encoding it, we make sure such window cannot appear.

Redundancy:

The number of redundancy bits of the encoded word would be $\left\lceil\frac{n-n'}{n'-k}\right\rceil+1$, where $\left\lceil\frac{n-n'}{n'-k}\right\rceil$ is the number of blocks we split the input word into. Note that n' can be interpreted as 2^{k-1}, and then the number of redundancy bits is $\left\lceil\frac{n-2^{k-1}}{2^{k-1}-k}\right\rceil+1$

Now, we wish to compare the number of redundancy bits in the described algorithm, to the naïve algorithm.

In the naïve algorithm we divide the input word into n' sized blocks, run the classic RLL algorithm on each one, and append them by adding another redundancy bit ('1') between two consecutive encoded words. This algorithm would make redundancy of $2\cdot\frac{n}{2^{k-1}}-1=2\cdot(\frac{n}{2^{k-1}}-\frac{1}{2})$.

Finally:

$$\begin{aligned}
\#Redundancy\ bits\ of\ represented\ algorithm &= \left\lceil\frac{n-2^{k-1}}{2^{k-1}-k}\right\rceil+1 \\
&\leq \frac{n-2^{k-1}}{2^{k-1}-k}+2=\frac{n}{2^{k-1}-k}-\frac{2^{k-1}}{2^{k-1}-k}+2 \\
&=\frac{n}{2^{k-1}-k}-\frac{2^{k-1}+2\cdot(2^{k-1}-k)}{2^{k-1}-k} \\
&=\frac{n}{2^{k-1}-k}-\frac{3\cdot 2^{k-1}-2k}{2^{k-1}-k}<\frac{n}{2^{k-1}-k}-\frac{3\cdot 2^{k-1}-3k}{2^{k-1}-k} \\
&=\frac{n}{2^{k-1}-k}-3=\frac{n}{2^{k-1}-k}-3+\frac{n}{2^{k-1}}-\frac{n}{2^{k-1}} \\
&=\frac{n}{2^{k-1}}-\frac{1}{2}+\frac{n}{2^{k-1}-k}-\frac{n}{2^{k-1}}-\frac{5}{2} \\
&<\frac{n}{2^{k-1}}-\frac{1}{2}+\frac{n}{2^{k-1}-k}-\frac{n}{2^{k-1}} \\
&=\frac{n}{2^{k-1}}-\frac{1}{2}+\frac{nk}{(2^{k-1}-k)\cdot(2^{k-1})} \\
&=\frac{1}{2}\cdot\#Redundancy\ bits\ of\ naive\ algorithm+\epsilon(n,k)
\end{aligned}$$

Where $\epsilon(n,k)\to 1$ when both $n\to\infty$ and $k\to\log n$. More precisely, when $n=1024$ and $k=10$, $\epsilon(n,k)=0.04$.

$$\#Naive\ algorithm\ rate = \frac{n}{2\cdot\left(\frac{n}{2^{k-1}}-\frac{1}{2}\right)+n} \approx \frac{n}{\frac{n}{2^{k-2}}+n} = \frac{1}{\frac{1}{2^{k-2}}+1}$$
$$= \frac{1}{\frac{1+2^{k-2}}{2^{k-2}}} = \frac{2^{k-2}}{1+2^{k-2}} = \frac{1}{1+2^{-k+2}}$$

$$\#Represented\ algorithm\ rate = \frac{n}{n+\left\lceil\frac{n-2^{k-1}}{2^{k-1}-k}\right\rceil+1} \approx \frac{n}{n+\frac{n-2^{k-1}}{2^{k-1}-k}}$$
$$= \frac{n}{n+\frac{n}{2^{k-1}-k}-\frac{2^{k-1}}{2^{k-1}-k}} \approx^{k\geq 3\Rightarrow\frac{2^{k-1}}{2^{k-1}-k}\leq 4}$$
$$= \frac{n}{n+\frac{n}{2^{k-1}-k}} = \frac{1}{1+\frac{1}{2^{k-1}-k}} = \frac{1}{\frac{(2^{k-1}-k+1)}{2^{k-1}-k}}$$
$$= \frac{2^{k-1}-k}{2^{k-1}-k+1} = \frac{1}{1+\frac{1}{2^{k-1}-k}}$$

As described in [2], we denote the best capacity of such algorithms by $E_{0,k'}$ where $k' = k+1$.

the best rate of such algorithms is given by $\log_2$ of the largest real root of the polynomial:

$$X^{k'+1} - X^{k'} - X^{k'-1} - \cdots - X - 1$$

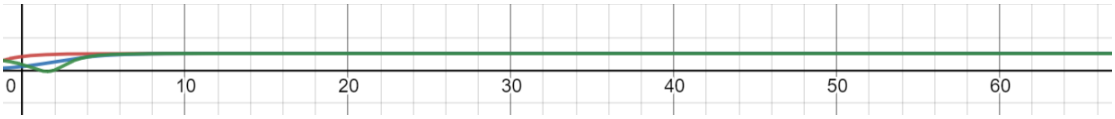
Alternatively for large enough k values:

$$1 - E_{0,k'} = \frac{\log_2 e}{2^{k'+2}} \Longrightarrow E_{0,k'} = 1 - \frac{\log_2 e}{2^{k'+2}} \Longrightarrow E_{0,k'} = \frac{2^{k'+2} - \log_2 e}{2^{k'+2}}$$

Algorithm Rates Comparison:

	$E_{0,k'}$	$Naive\ Algorithm$	$Represented\ Algorithm$
$k=3$	0.975	0.6667	0.5
$k=4$	0.988	0.8	0.8
$k=5$	0.994	0.889	0.9167
$k=6$	0.997	0.941	0.963

As of large k values, we get the following graph (with red being the capacity bound, blue being the Naïve algorithm and green being the represented algorithm):



We would like to add a proof such that for every $k \geq 4$, the represented algorithm rate is an upper bound of the Naïve one:

$$\frac{Naive\ Rate}{Represeted\ Rate} = \frac{\frac{1}{1+2^{-k+2}}}{\frac{1}{1+\frac{1}{2^{k-1}-k}}} \leq 1 \Leftrightarrow \frac{1}{1+2^{-k+2}} \leq \frac{1}{1+\frac{1}{2^{k-1}-k}}$$

$$\Leftrightarrow$$

$$1+\frac{1}{2^{k-1}-k} \leq 1+2^{-k+2} \Leftrightarrow 1+\frac{1}{2^{k-1}-k} \leq 1+\frac{1}{2^{k-2}} \Leftrightarrow$$

$$\frac{1}{2^{k-1}-k} \leq \frac{1}{2^{k-2}} \Leftrightarrow 2^{k-2} \leq 2^{k-1}-k \Leftrightarrow k \leq 2^{k-1}-2^{k-2} \Leftrightarrow$$

$$k \leq 2^{k-2} \Leftrightarrow \log k \leq k-2 \Leftrightarrow k \geq 4$$

Decoding:

Decoding an encoded word $w \in \{0,1\}^m$ and an integer k, we would start from the end of the word to the start, decoding one block by another.

Then calculate the length of the first word to be decoded (the last word that has been encoded), and decode it by using the classic decoding algorithm (as well as cutting it from w). Then, we initialize an output z by assigning to it the decoded word without its first k bits, and appending them back to w. From now on, every iteration starts by cutting the last n' bits from w and decoding them, add the last $n' - k$ bits to z (from the left), and append the left k bits back to w.

Correctness:

Each iteration we add $n' - k$ bits to z, which (from the correctness of the algorithm) match $n' - k$ bits of the original word. By peeling the encoded blocks one by one, iterating from the right to the left, we make sure that each call to the decoding algorithm is independent of blocks from the right, thus ensuring the output is the original word.

Complexity:

$$first\ block + B_i\ blocks = O(n') + \sum_{i=1}^{\left\lceil \frac{n-n'}{n'-k} \right\rceil} O(|B_i|) =^{every\ B_i\ is\ of\ size \leq n'}$$

$$O(n') + \left\lceil \frac{n-n'}{n'-k} \right\rceil \cdot O(n') \leq^{(*)}$$

$$O(n) + \left\lceil \frac{n-n'}{n'-k} \right\rceil \cdot O(n) \leq O(n) + n \cdot O(n) = O(n^2)$$

(*) – Note that in case $n' \geq n$ we can just encode the original word by using the classic algorithm, because $k = \log n' \geq \log n$, and therefore by ensuring no zero-runs longer than $\log n$ are in the encoded word, we also ensure no zero-runs longer or equal to k.

K-Zeros Run Length Limited Encoding – File Support:

Finally, we wanted to add file support, such that a user could encode an entire file with the guarantee that no encoded line would have k or longer zero runs.

We split the given file into lines, and encode each line using our extended encoding algorithm as explained above. The output of this algorithm would be an encoded file. In order to decode the encoded file, again, we would iterate each line using the extended decoding algorithm as explained above.

K-Zeros Run Length Limited Encoding – Implementation:

Source: https://github.com/hadasabraham/KLimitRLL

Our solution is implemented in *Python*, based on the classical algorithm described in [1]. We support any k starting from 3.

In our implementation, we provide the following API:

- EncodeFile(str *filename,* int *k*)

Used to encode a file. The output is an encoded file that has no k-or-longer zero runs.

- DecodeFile(str *filename,* int *k*)

A decoding function that returns the original file.

- EncodeAnyRun(str *word,* int *k*)

Used to encode a word. The output is an encoded word that has no k-or-longer zero runs.

- DecodeAnyRun(str *word,* int *k*)

A decoding function that returns the original word.

- kLimitRLLTest(int *iters*, int *min_k*, int *max_k*, int *min_length*, int *max_length*)

A test function that runs for iters iterations, each time generating a binary word of size between min_length and max_length and a *k* of size between *min_k* and *max_k*. Checking whether the encoded word does not have zero-runs longer or equal to k, and that the decoded-encoded word is equal to the original one.

<u>Notes:</u>

1. In each of these functions, if no k was provided, the functions will assume $k = \lceil \log n \rceil + 1$, where n is the line/word length (as defined in the classical RLL algorithm).

2. Our functions provide support for words over the binary alphabet $\Sigma = \{0,1\}$.

References:

[1] - *M. Levy and E. Yaakobi, "Mutually uncorrelated code for DNA storage," IEEE Transactions on Information Theory, vol. 65, no. 6, pp. 3671–3691, Jun. 2019.*

[2] - *A. Kato and Kenneth Zeger, "A comment regarding: "On the Capacity of Two-Dimensional Run Length Constrained Channels", Jun. 2005.*