Knuth Algorithm for Balanced Codes

Noa Marelly* Ohad Goudsmid †

February 25, 2020

1 Problem Definition

In many cases, we want that encoding that we use do not have many appearances of the same symbol, for example when dealing with charge leakage. In this case, charge is depleting over time, and we want to be able to read the information encoded in the memory. Using codes with predefined ratio between *high* and *low* number of entries, allows to set the threshold dynamically, resulting in ability to access the data even when charge is lost.

When the number of appearances of each symbol is the same (for binary alphabet), we have the property that no word is "contained" in another, a property which is highly important in laser disk storage.

We want to be able to encode vectors of length n into vectors of length m ($m \geq n$) such that the number of occurrences of each symbol are equal to one another.

Definition 1.1. A vector v over alphabet Σ of length m (where m is divisible by $|\Sigma|$) is called **balanced** if the number of occurrences of each symbol $\sigma \in \Sigma$ is exactly $\frac{m}{|\Sigma|}$.

Definition 1.2. A encoding scheme $(\mathcal{E}, \mathcal{D})$ is called balaced if $\forall v \in \Sigma^n$ $\mathcal{E}(v)$ is a balanced codeword.

Our goal is to implement such coding scheme.

*314617705, noa.marelly@campus.technion.ac.il
†207838657, goudsmidohad@campus.technion.ac.il

2 Known Solution

We based our solution on the algorithm suggested in [1] for binary alphabet, and extended it to larger alphabets.

The algorithm suggested in [1] goes as follows, given $v \in \{0,1\}^n$:

1. initialize $i = 0$
2. Calculate the number of 1s and 0s in v, denoted $\#_0(v), \#_1(v)$ correspondingly.
   If $\#_0(v) = \#_1(v)$ halt and return $u = v \cdot i \cdot \bar{i}$.
3. Increase $i \leftarrow i + 1$ and flip $v_i \leftarrow \bar{v_i}$
4. Return to 2.

We saw the proof for this algorithm in the lectures. Notice that the output vector has bigger length than v, which is described by the formula $|u| = |v| + 2\log_2(|v|)$. By recursively using this algorithm on the index (without adding its negation) we can (asymptotically) decrease u's length.
We used this recursive solution in our implementation.

3 Generalisation for Larger Alphabets

For supporting larger alphabets some modifications need to be made. First, an important observation is that if we want to change symbols in (predefined) couples some vectors cannot be balanced in a similar manner as before. But, instead when balancing the appearances of σ, we need to choose a symbol σ' such that either $\#_\sigma(x) < \frac{|v|}{|\Sigma|} < \#_{\sigma'}(v)$ or
$\#_{\sigma'}(x) < \frac{|v|}{|\Sigma|} < \#_\sigma(v)$. Doing so requires us to encode the replacing symbol as well as the index from before, adding a single symbol for the encoding in each stage. This change allows the previous proof for the correctness of the algorithm to be valid for larger alphabets.

Therefore, by repeating the process of balancing a symbol to the goal of $\frac{|v|}{|\Sigma|}$ appearances $|\Sigma|-1$ times, we will obtain a balanced vector from v. Now we need to balance the additional symbols and indices we added.

The additional information can be balanced by duplicating it $|\Sigma|$ times, each time replacing each letter (from the previous copy) by its successive (using some arbitrary order on the letters).

This algorithm is with complexity linear both in $|v|$ and $|\Sigma|$.

4 Implementation

We implemented our solution[1] in Python3, based on the previously suggested algorithm. In our implementation we support alphabets of size up to 10, and use $\Sigma = \{0, 1, \ldots, |\Sigma| - 1\}$ as the alphabet of size $|\Sigma|$.

We have several important functions, defined in **knuth.py**:

1. ENCODE_KNUTH: An encoding function, which receives a vector $v \in \Sigma^n$ and the size of Σ, and returns a balanced word u. Note that $n_{(mod|\Sigma|)}$ must be 0.
2. DECODE_KNUTH: A decoding function, which receives a vector $u \in \Sigma^m$, the size of Σ and the original vector length l. It returns the original vector.
3. CALCULATE_OUTPUT_LENGTH: This function calculates the encoded length of a vector, given its original length and the size of the alphabet.

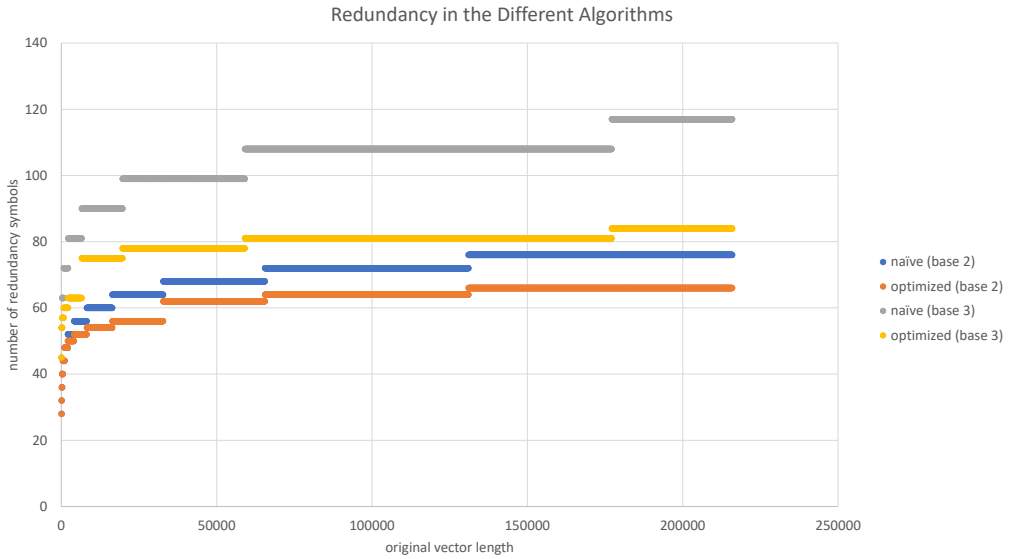
We also implemented some tests which appear in **tests.py**.

5 Optimisation for Reducing the Length of the Encoded Vector

In order to achieve a better redundancy, we apply the algorithm recursively on the additional information. We found out that for short vectors, duplicating is better. Thus, in our solution we apply the algorithm recursively until duplicating is shorter. This way, we achieve the optimal length that can be achieved by this method.

The following graph describes the redundancy of the different implementations of the algorithm.

[1] Code can be found in this Git repository



This graph compares the redundancy between our optimised implementation and the non-recursive solution from [1] and the lectures. As shown in the graph, the redundancy of the non-recursive (naïve) algorithm is larger than the redundancy in the optimised algorithm. Furthermore, we can see that the difference between the implementation is greater when the length of the vector is greater. Thus, when using longer vectors it is much better to use our implementation compared to the original non-recursive implementation.

References

[1] Donald E. Knuth. Efficient balanced codes. *IEEE Transactions on Information Theory*, 32(1):51–53, 1986.