Knuth Algorithm For Balanced Codes Using Gray Code

Itay Nakash, Yuval Goldberg

March 9, 2022

Abstract

Our goal is to use Gray code in the Knuth algorithm for balanced codes, and create balanced code word with only $\log(2n+1)+1 \approx \log(n)+2$ redundancy bits. Using Gray code for the Knuth algorithm will help us create an almost-continuous function for the code's Hamming weight, and by that, a code as needed.

1 Introduction

1.1 Problem Definition

We want to be able to encode information without using many appearances of the same symbol, to address engineering problems such as dealing with charge leakage, as we saw in the lectures.
In this example, charge is being depleting over time, and we want to be able to read the encoded information after the charge depletes. Using codes with predefined ratio between high entries and low entries will allow us to set the charge threshold dynamically and have access to the data even when charge has been lost.

Therefore, given a vector $v \in \{0,1\}^n$, we want to encode v into $u \in \{0,1\}^m$ such that $m > n$, and $\#_1(u) = \#_0(u)$.
We will define u as a balanced code word if $\#_1(u) = \#_0(u)$, and we will define encoder C as a balanced encoder if for each $v \in \{0,1\}^n$, $C(v) = u$ when u is a balanced code word.

1.2 Known Solution

We based our solution on Knuth's algorithm for balanced codes suggested in [1] for binary alphabet. The suggested algorithm's iteration goes as follows, given $v \in \{0,1\}^n$:

1. Initialize $i = 0$
2. Count the number of 1s in v, if $\#_1(v) = \#_0(v)$, go to 5

3. $i \leftarrow i+1$, and assign $\vec{v}_i = \overline{\vec{v}_i}$

4. Repeat 2

5. encode i in additional $log(n+1)$ bits in the end of $\vec{v}$ and return the new vector $\vec{v}$

We saw a proof to the correctness of this algorithm in the lectures.
The original algorithm is focusing about balancing the original word, without addressing the additional bits of metadata. The solution we saw in the lectures to this problem is to recursively use this algorithm on the index bits. This solution will add additional $log(|index|)$ for each iteration, and can add a significant number of redundancy bits.

1.3 Issues we addressed

The main issue we addressed was the need for fine-tuning the metadata bits (the bits using to decode the index of the last flipped bit).
In the current Knuth algorithm, balancing those bits can add additional redundancy to the code.
We decided to balance the word including its metadata (the index), using gray code, and by that to get a fully balanced word, using only $\log{(n+1)}+1$ redundancy bits.

In addition, after comparing analytical and numerical methods, we added a decode function the can decode a word without the length of the original encoded word.

2 Initial attempt

Initially we tried to implement Knuth algorithm using regular gray code, and balancing the entire word at once (including the additional $\log n + 1$ bits of metadata).
In each step of this algorithm, we flip a single bit in the original data, and because of Gray code's property of flipping only a single bit when increasing the value by 1, in each step the Hamming weight of the whole word would change by 0 or 2, which is almost continuous, and resembles the original Knuth algorithm.
In order to deal with the jumps by 2, we added an extra balancing bit to the end of the word, using to fine-tune the weight, in order to achieve a fully-balanced word in the end. By that, we hoped we will save the additional balancing of the word, and encode each word with only $\log n + 1 + 1$ redundancy bits. Although brute-force and stress tests will show that for almost all words this attempt will work, we identified that it can't promise a balanced encoded word for all possible data vectors.
Explicitly, our first algorithm attempt was:

1. Initialize $i = 0$. Denote u^i as the Gray code of i.

2. Count the number of 1s in vu^i, If $|\#_1(vu^i) - \#_0(vu^i)| \leq 2$, go to 5.

3. Increase $i \leftarrow i+1$, and assign $\vec{v}_i = \overline{\vec{v}_i}$.

4. Repeat 2.
5. Add a fine-tuning bit w to the end of vu^i in order to achieve $|\#_1(vu^iw) - \#_0(vu^iw)| \leq 1$.
6. Return vu^iw.

Note: Difference of 1 is only achievable when $Length(vu^iw)$ is odd.

2.1 Empirical testing

At first, in order to check the feasibility of the algorithm, we created a basic Python script that's working on small vectors. The script runs over all of the 20 bits vectors, and balancing each one of them using the algorithm. After balancing successfully all those vectors, we were optimistic and started working on the algorithm's correctness proof.

2.2 Proof attempt

Denote $k = \#_1 - \#_0$
As we saw in the regular Knuth proof from the lectures, in order to prove that the algorithm will find a balanced word during its encoding, we need to prove two lemmas:

1. The weight difference function is "almost-continuous" (continuous up to a difference of 2).
2. The weight difference, k, is starting when $k \leq \frac{n}{2}$ and finishing when $k' \geq \frac{n}{2}$ or vice versa.

We were able to show, as mentioned before, that the function is indeed continuous up to a difference of 2 ("almost-continuous"). However, we encountered some issues proving the second lemma.
After adding the index (using Gray code), we added additional $log(n+1)$ bits that change the weight balance of the encoded word.
We first noticed that when the encoded word v has $|\#_1(v) - \#_0(v)| \geq \log(n+1)+1$ it easy to prove that after flipping all of the bits in the original word and adding the Gray-encoded index, we will get a vector as wanted for the second condition. Let's assume $\#_1(v) - \#_0(v) \geq \log(n+1)+1$.
We will get $\#_0(\overline{v}) - \#_1(\overline{v}) \geq \log(n+1)+1$, so after adding an arbitrary $\log(n+1)+1$ bits word, that we denote as u:

$$\#_1(vu) - \#_0(vu) = (\#_1(v) - \#_0(v)) + (\#_1(u) - \#_0(u)) \geq (\log(n+1)+1) - (\log(n+1)+1) = 0$$
$$\#_0(\overline{v}u) - \#_1(\overline{v}u) = (\#_0(\overline{v}) - \#_1(\overline{v})) + (\#_0(u) - \#_1(u)) \geq (\log(n+1)+1) - (\log(n+1)+1) = 0$$

Therefore, the Hamming weight changes gradually from above half of the length of the word, to below it. So the second lemma applies in this special case, and we could prove the correctness of the whole algorithm for this case.

2.3 Edge cases

After a few disproof attempts, we encountered the following counterexample:

$$v = 111111110000000$$

Let's see all of the algorithm steps for this word, and their Hamming weight:

	Flipped word	*Gray*	
0.	111111110000000	0000	$weight = 8$
1.	011111110000000	0001	$weight = 8$
2.	001111110000000	0011	$weight = 8$
3.	000111110000000	0010	$weight = 6$
4.	000011110000000	0110	$weight = 6$
5.	000001110000000	0111	$weight = 6$
6.	000000110000000	0101	$weight = 4$
7.	000000010000000	0100	$weight = 2$
8.	000000000000000	1100	$weight = 2$
9.	000000001000000	1101	$weight = 4$
10.	000000001100000	1111	$weight = 6$
11.	000000001110000	1110	$weight = 6$
12.	000000001111000	1010	$weight = 6$
13.	000000001111100	1011	$weight = 8$
14.	000000001111110	1001	$weight = 8$
15.	000000001111111	1000	$weight = 8$

While the word length is 19, so a balanced code-word will have the weight of 9 or 10.
But even adding an additional fine-tuning bit, will extend the length of this word to 20, and it's not possible to balance it using this method.
In a similar way, using brute-force, we were able to find counterexamples for different kinds of unusual Gray codes (such as monotonic Gray code, that could promise an almost-increasing Gray code by Hamming weight) that disprove this algorithm for each Gray code.

3 Improved algorithm

After the failure, we encountered a similar solution, that is a special case of an algorithm shown in [2]. In contrast to our first assumption, and as shown above, to implement this algorithm we will need $\log(2n+1) + 1 \approx \log(n) + 2$ redundancy bits. The mentioned article proves that after finishing to flip increasing prefixes of the original word, we start to flip decreasing suffixes of the original word, and encode the index of the flip using Gray code, it will guarantee that a balanced word would be found

along the algorithm.
The improved algorithm goes a follows:

1. Initialize $i = 0$. Denote u^i as the Gray code of i.
2. Count the number of 1s in vu^i, If $|\#_1(vu^i) - \#_0(vu^i)| \leq 2$, go to 6.
3. If $i > 0$, flip $\vec{v}_{(i-1)(mod\, n)}$
4. Increase $i \leftarrow i + 1$.
5. Repeat 2.
6. Add a fine-tuning bit w to the end of vu^i in order to achieve $|\#_1(vu^i w) - \#_0(vu^i w)| \leq 1$.
7. Return $vu^i w$.

3.1 Finding the word length

We decided to add additional feature to our project, and to able users to decode a word without knowing its initial size (the user will use only the encoded full vector, without knowing how many redundancy bits it actually has). Since we know the encoding algorithm, we can conclude this equation for the original word size:

$$m = n + \log(2n + 1) + 1$$

Initially we tried solving this equation analytically, and used Lambert W function to calculate the original size:

$$n = \frac{1}{2}(2W(\frac{1}{2}e^{m-\frac{1}{2}}) - 1)$$

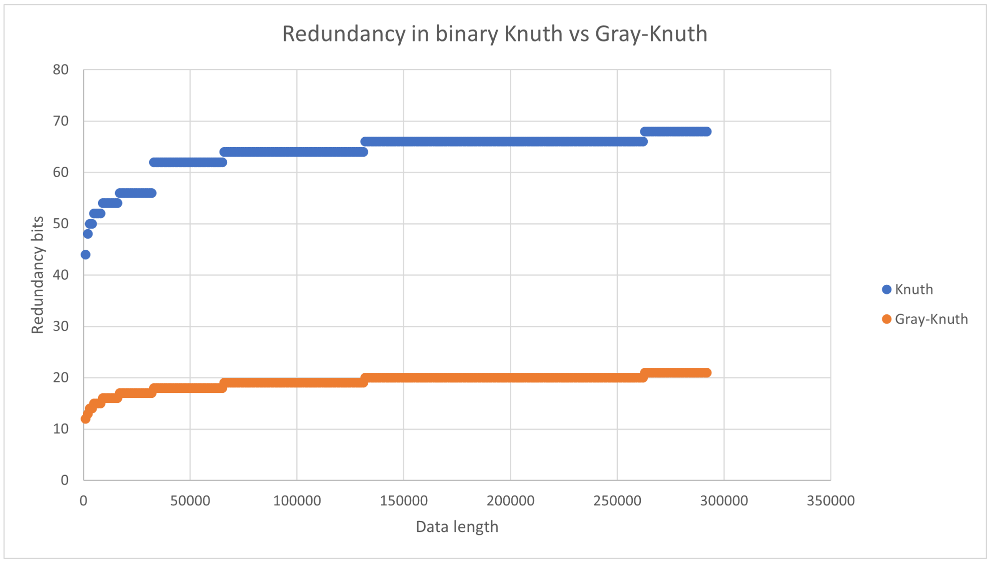
However, due to numerical errors, we saw that this calculation isn't accurate enough, so we had to check a few lengths in the same area of the computed length in order to find the correct one. In additions, this calculation doesn't even work for large words (with lengths of 1000 and above).
Therefore, we decided to solve this using binary search to find the exact value that will solve the equation, relying on the fact the the length of the encoded word is increasing as a function of n.

4 Implementation

We used git as our version-control framework during the development phase, and we uploaded our code as open source on GitHub: GitHub repository
In the GitHub repository you can find all the implementation under "*GrayKnuth*" package, with detailed README file. We used Python, and sympy package that implements Gray code functionality.
We also implemented some tests using "*pytest*" framework, which appear in test_gray_knuth.py.

5 Results

After applying all the above improvements to the classic Knuth algorithm, our algorithm was able to encode balanced word much more efficiently, and with less redundancy. The following graph describes the redundancy of our algorithm in compare to an optimized implementation of the original Knuth algorithm:



This graph compares the redundancy between our new algorithm implementation and the recursive optimized existing implementation of the algorithm. As our analysis predicted, the original algorithm require more redundancy bits than our improved algorithm. In addition, we can see that the difference is greater when as length of the vector is greater. Thus, when using longer vectors it is much better to use our implementation compared to the original.

References

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

[2] Elie N Mambou and Theo G Swart. A construction for balancing non-binary sequences based on gray code prefixes. *IEEE Transactions on Information Theory*, 64(8):5961–5969, 2017.