*Mathematics and Computer Science*
2024, Vol. 9, No. 2, pp. 26-35
https://doi.org/10.11648/j.mcs.20240902.11

Research Article

# An Optimised Hoffman Algorithm for Testing Linear Code Equivalency

**Olufemi Ololade Olaewe**[1], **Peter Awonnatemi Agbedemnab**[1, *], **Mohammed Muniru Iddrisu**[2]

[1]Department of Information Systems and Technology, C. K. Tedam of University of Technology and Applied Science, Navrongo, Ghana

[2]Department of Mathematics, University for Development Studies, Tamale, Ghana

## Abstract

The Hoffman's algorithm to test equivalency of linear codes is one of the techniques that have been used over the years; it is achieved by a comparison of codewords of the linear codes. However, this comparison technique becomes ineffective in instances where it is applied to linear codes with larger dimensions as it requires much run time complexity, space and size in comparing the codewords of each linear code. This paper proposes an optimised algorithm for testing the equivalency of linear codes, specifically addressing the limitations of the Hoffman method. To assess and compare the efficiencies of the Hoffman algorithm and the optimised algorithm, a set of nine carefully selected linear codes were subjected to equivalency testing. The CPU runtime of both algorithms was recorded using the C++ chrono library. The recorded runtime data was then utilized to create a scatter plot, offering a visual representation of the contrasting trends in CPU runtime between the two algorithms. The plot clearly indicate exponential growth in CPU runtime for the Hoffman algorithm as the length and dimension of the linear codes increases, in contrast, the proposed algorithm showcased a minimal growth in CPU runtime, indicating its superior efficiency and optimised performance.

## Keywords

Linear Codes, Code Equivalency, Hoffman Algorithm, Codewords

## 1. Introduction

Security in communication systems has historically been obtained through cryptographic means which uses concept of coding. Recent research has focused in this aspect and has unveiled ample opportunity for security design [1].

Equivalency of linear code is a major aspect in coding theorem; over the years, algorithms have been developed to aid the test for equivalency of linear codes, for instance, the support splitting algorithm deduces the permutation that exists between equivalent linear binary codes [2]. Testing for equivalency of linear codes is equally necessary and is not catered for by the support splitting algorithm. Code equivalence is a basic concept in coding theory with several applications in code-based cryptography; the McEliece public-key cryptosystem, Girault's Identification scheme and the Cour-

*Corresponding author: pagbedemnab@cktutas.edu.gh (Peter Awonnatemi Agbedemnab)

**Received:** 26 March 2024; **Accepted:** 10 April 2024; **Published:** 29 April 2024

tois-Finiasz-Sendrier (CFS) signature scheme [3], to name a few. The notion of equivalence of linear codes used in code-based cryptography usually involves only permutations as the code alphabet is the binary field. However, this is by far the case in coding theory where for a more general notion of equivalence all isometries of the Hamming space have to be included.

A research work on linear code equivalence as indicated in [4] dealt with the problem of deciding if two finite dimensional linear subspaces over an arbitrary field are identical up to a permutation of the coordinates. The researchers showed that given access to a subroutine that decides if weighted undirected graphs are isomorphic, one may deterministically decide the permutation code equivalence, provided that the underlying vector spaces interest trivially with their orthogonal complement with respect to an arbitrary inner product.

The code equivalence problem is to decide whether two linear codes over F_q are equivalent. Testing for equivalency of linear codes require comparison of codewords of the linear codes i.e. scanning the bits of the codewords, [5]. The comparison technique which was used by the previous study becomes ineffective as the dimension k or length n of the code increases. The factors which are to be considered in measuring efficiency of algorithm are time complexity, space complexity, administrative cost and faster implementation. One of the effective methods for studying the efficiency of algorithm is the Big O-Notation, [6]. The comparison technique becomes ineffective in the sense that when it is subjected to a linear code with large dimension or length, much resources i.e. run time complexity, space and size are required in comparing the codewords of each linear code. Moreover, comparing algorithms with their theoretical time complexity boundaries can be achieved by analysing the behaviour of the algorithm implementation in real environment, [7]. This suggests that to efficiently measure time complexity of an algorithm, it has to be subjected to real and applicable system or data which for this paper is linear code and the algorithm being equivalency test algorithm. Time complexity is known as one of the tasks of a comprehensive algorithm analysis, the objective of analysis is to obtain a function which for a given size of the problem estimates the time needed for the algorithm to execute successfully [8].

Al-Khwarizmi in his write up “About Indian Counting” explained algorithms of four arithmetic operations. Comparing Roman and decimal counting systems. He wrote “... we decided to explain Indian counting with IX letter, which they use to explain any of their digits for ease and brevity facilitating business for any person, who is learning arithmetic”, [9]. The three words used here i.e. easy, brevity and facilitation – in algorithm can be explained as “complexity”. Several works have discussed methods for calculating time complexity for combinational and sequential schemes; some of these also deduced a formula to calculate the time complexity of SH-Model of algorithm, [9]. The concept of time complexity is one that is central to the theory of algorithms. During the last century it was clarified with the development of the theory and practice of computing and the appearance of new models of algorithms. As a result of this clarification, nowadays there are several options of estimation of complexity of algorithm objects. Theoretical analysis is often complicated and has other drawbacks as well. Therefore, empirical analysis of time complexity is preferred and can be used in measuring efficiency in terms of execution time of an algorithm.

There exists data how to measure efficiency/time complexity of algorithms by running algorithms on problems of different sizes as demonstrated in [8]. This suggest that time complexity of algorithm can be measured by executing the algorithm on different sizes of problem which for this study are equivalent linear codes. Another work focused on iterations (loops, and recursive calls) from which to build a tree, [10]. For each node the total number of repetitions which is a function of input size. The tool does not contain an automatic calculation of beehive that fit the measurement but is calculated manually. Yet again, another research recorded how many times a base block is executed; in most cases, this is a line of procedure by method clustering then combines the blocks and finds the linear potential function that fit better, [11]. The tool at the entrance expects test cases of different sizes, to be determined by the user. A new form of graph referred to as complexity plots, where there are complexity classes on the x-axis and size on the y-axis problems was introduced in [12]. The approach is interesting because the values on the x-axis are without units, an estimation is done but with estimations function that best fit the measurements.

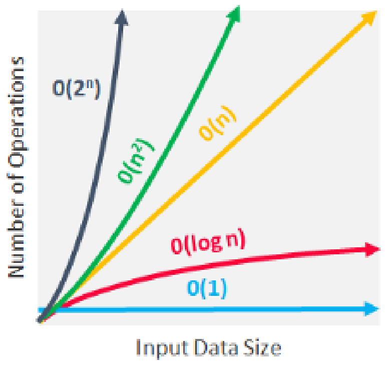

***Figure 1.*** *Possible time complexities of algorithms.*

One of the straightforward definitions for optimisation is “doing the most with the least” [13]. Another study by [14] opined that algorithms with time complexity of O(n2) in most scenarios and particularly for large datasets take a lot of time to execute and should be avoided as it needs a lot of resources thereby deducing equivalency of linear code by comparison of codewords is inefficient as its time complexity, O(k*n), where k and n are the dimension and length of the linear codes respectively. Figure 1 illustrates the possible time complexities of algorithms.

## 2. Materials and Methods

### 2.1. Feasibility and Study Phase

The main aim here is to ascertain whether optimizing the equivalence algorithm in [5] is feasible. After a careful collection of data concerning time of execution of Hoffman's equivalency test algorithm, it was clear that the new algorithm will be both logically and technically feasible. Technically, the new algorithm can be implemented in C++ (see plus-plus) which is very comfortable to program with and has accurate measurement of program execution time.

### 2.2. C++ as a Program Execution Time Measurement Tool

C++ is a general-purpose programming language created as an extension to the C programming language. The use of C++ expanded significantly over time and modern C++ now has object-oriented, generic and functional feature in addition to features for low-memory manipulation. The design of C++ was geared system programming and embedded, resource-constrained software and large systems, with performance, efficiency, and flexibility of use as its design highlights [15]. C++ has a library that was introduced in C++ 11, which is able to find out time taken by different parts of program using std:: chrono, hence the main reason for implementing the new algorithm in using C++ [16]. Furthermore, the std: chrono has two distinct instances which are timepoint and duration. A timepoint represents a point in time whereas a duration represents the interval or span of time. The C++ library enables us to subtract two timepoints to get the interval of time passed in between. The std:: chrono also provides us with three clocks with varying accuracy. The high_resolution_clock is the most accurate and hence it is used to measure execution time.

### 2.3. Source of Data and Data Analysis

The primary data that is used in this paper constitutes time complexities of the [5] algorithm as well as that of the established algorithm which was recorded on execution of the two algorithms on different sizes of linear code.

The most efficient algorithm can be said to be one that takes the least amount of execution and memory usage possible while yielding accurate result. To measure the runtime of an algorithm with the aim of determining its efficiency one must implement the algorithm in an actual programming language [17]. To ensure accuracy in the Central Processing Unit (CPU) time recordings, the two algorithms were implemented as a C++ application as it has CPU runtime measure libraries. Scatter plot which is an analysis tool provided by MATLAB and also a visualization tool is used in analysing the data. The analyses of these data give room to measure efficiency and accuracy of the two algorithms.

### 2.4. Scatter Plots

Quantification leads to precision provided by numbers. Numerical representation of behaviour of quantitative measurement serves as the medium through which all analysis occurs, [18]. Conducting research on analysis, [18] indicated that out of 4313 reviewed graphs, 3560 had a quantitative scaled vertical axis. In other words, 83% of the reviewed graphs prominently displayed behaviour as a quantity. In analysis, data is graphed for each participant during a study with trend, level and stability of data assessed within and between conditions [19].

A scatter plot is used in visualizing the data collected graphically as it is commonly used to display change over time as a series of data points. The scatter plot therefore enables researchers to determine the relationship between sets of values with one dataset always being dependent on the other set. Scatter plots are powerful visual tools that illustrate trends in data over a period of a particular correlation. Furthermore, Scatter plots is a useful tool in that it shows data variables and trends clearly and can enable researchers to make predictions about the results of data not yet recorded hence its adoption to compare efficiency of (Hoffman et al., 1991a) algorithm and the proposed algorithm.

### 2.5. Linear Code Generation

Two linear codes of the same length can be combined to develop a third code which will be twice the length in a way similar to the direct sum of the code's construction. The dimension of the newly formed code can therefore be a $[2n,\ k_1 + k_2,\ \min(2d_1 + d_2)]$ linear code [20]. Generator matrix of the newly formed linear code is given as.

$$\begin{pmatrix} G_1 & G_1 \\ 0 & G_2 \end{pmatrix} \tag{1}$$

The [4,1,4] linear code as used by [20] is adopted as the primary linear code in the paper.

### 2.6. Gilbert Varshamov Bound

To determine if the generated linear codes exists, Gilbert Varshamov bound as implemented by (Hoffman et al., 1991b) is adopted:

Theorem: There exist a linear code of length $n$, dimension $k$ and distance $d$ if

$$\binom{n-1}{0} + \binom{n-1}{1} + \cdots + \binom{n-1}{d-2} < 2^{n-k} \tag{2}$$

Corollary: If $n \neq 1, d \neq 1$ then there exists an $(n,k,d)$ linear code with

$$|c| \geq \frac{2^{n-1}}{\binom{n-1}{0}+\binom{n-1}{1}+\cdots+\binom{n-1}{d-2}} \tag{3}$$

The corollary is a lower bound for the number of words in a linear code of length $n$ and distance $d$.

## 2.7. Hamming Bound

Theorem: Let $C$ be a code of length $n$ and d$= 2t + 1$ then the number of words in the code is given as

$$|c| \leq \frac{2^n}{\binom{n}{0}+\binom{n}{1}+\cdots+\binom{n}{t}} \tag{3}$$

The Hamming bound is an upper bound for the number of words in a linear code of length n and distance $d = 2t + 1$.

## 3. Results and Discussion

### 3.1. The Proposed Linear Equivalency Test Algorithm

The proposed algorithm is based on [5] equivalency test algorithm but it does not compare corresponding words unlike the case presented in [5]. The aim is to improve efficiency and accuracy of Linear code equivalent test algorithm. It ignores the zeros in the codewords and take note of the position of the ones in the codewords before determining the equivalency. Not considering the zero codewords further makes the algorithm more efficient as it reduces the number of bits to work with. However, to test the accuracy of the algorithm, it has been subjected to codes of $2^n$ length where $n = 2,3,4$.

*Steps of the proposed algorithm:*

Check if the two codes, C and $C'$ are linear codes by verifying that the two codes satisfy the properties of a linear code.

Locate the positions of 1’s in the codewords of C and C '.

Find $d = d_1 - d_2$, where $d_1$ and $d_2$ are the positions of 1’s in codewords of C and $C'$ respectively.

Compute sum of d’s denoted by $\sum d$

If $\sum d = 0$ then $C \cong C''$ otherwise they are not equivalent.

Examples of linear codes and its equivalence by the algorithm in [5] are presented in tabular form tables 1, 2 and 3.

*Table 1. Example of Linear code of length 2 with A and B representing each bit and its equivalent code derived from pattern 2,1.*

| A | B | A | B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

*Original Code | Derived Code.

*Table 2. Example of Linear code of length 3 with A, B and C representing each bit and its equivalent code derived from pattern 2,3,1.*

**Original code*

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

**Equivalent code with pattern 2,3,1*

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

*Table 3. Example of Linear code of length 4 with A, B, C and D representing each bit and its equivalent code derived from pattern 1,3,2,4.*

**Original code*

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |

| A | B | C | D |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |

**Equivalent code with pattern 1,3,2,4*

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |

### 3.2. Illustrative Example (Simulation)

Table 4 illustrates the usage of the proposed algorithm using linear code in Table 3 when n= 4 i.e. A, B, C and D

***Table 4.*** *Simulation of Proposed Algorithm.*

*Original code*

| A | B | C | D |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 |

*Equivalent code with pattern {1,3,2,4}*

| A | B | C | D | Position of 1's in C | Position of 1's in C ' | Diff. | SUM |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 4 | 4 | 0 | 0 |
| 0 | 1 | 0 | 0 | 3 | 2 | -1 | -1 |
| 0 | 1 | 0 | 1 | 3 4 | 2 4 | -1 0 | -2 |
| 0 | 0 | 1 | 0 | 2 | 3 | +1 | -1 |
| 0 | 0 | 1 | 1 | 2 4 | 3 4 | +1 0 | 0 |
| 0 | 1 | 1 | 0 | 2 3 | 2 3 | 0 0 | 0 |
| 0 | 1 | 1 | 1 | 2 3 4 | 2 3 4 | 0 0 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 4 | 1 4 | 0 0 | 0 |
| 1 | 1 | 0 | 0 | 1 3 | 1 2 | 0 -1 | -1 |
| 1 | 1 | 0 | 1 | 1 3 4 | 1 2 4 | 0 -1 0 | -2 |
| 1 | 0 | 1 | 0 | 1 2 | 1 3 | 0 1 | -1 |
| 1 | 0 | 1 | 1 | 1 2 4 | 1 3 4 | 0 1 0 | 0 |
| 1 | 1 | 1 | 0 | 1 2 3 | 1 2 3 | 0 0 0 | 0 |
| 1 | 1 | 1 | 1 | 1 2 3 4 | 1 2 3 4 | 0 0 0 0 | 0 |

It is noticed that the algorithm sums up to which indicates that the two codes are equivalent.

## 3.3. The Equivalence Test Algorithm by [5]

*Input*

$$C = \{x_1, x_2, x_3, \ldots, x_n\}$$

$$C' = \{y_1, y_2, y_3, \ldots, y_n\}$$

*Output*: $c$ and $c'$ are equivalent or are not equivalent

*Steps*

1) Check if $C$ and $C'$ are linear codes
2) Check $C'$ and verify if its codewords are all permuted from $C$ with same pattern.
3) If step 2 check is true $C$ and $C'$ are equivalent otherwise not equivalent
4) End program

The linear code equivalence problem in [5] is solved by comparing the codewords of the linear codes in question. Executing the algorithm on large length/dimension of linear codes will result in a situation where more resources will be required and the cost of implementation will also be high.

*Illustrative example of* [5] *algorithm*

$$C = \{0000, 1011, 0101, 1110\}$$

$$C' = \{0000, 1110, 0101, 1011\}$$

By comparing the codewords of $C$ and $C'$, 1432 arrangement of $C$ is constant in $C'$ hence the two codes are equivalent.

## 3.4. Performance Evaluation on Algorithm Runtime

The runtime of the proposed algorithm was compared to the existing algorithm by [5], using different length and dimensions of codewords. This comparison is shown in Table 5.

*Table 5. Runtime Comparison of the Hoffman Algorithm and the Proposed Algorithm.*

| Length (n) | Dimension (k) | Hoffman Algorithm Runtime (μs) | Proposed Algorithm Runtime (μs) |
|---|---|---|---|
| 8 | 2 | 995 | 541 |
| 16 | 4 | 1031 | 548 |
| 32 | 8 | 1601 | 550 |
| 64 | 16 | 1998 | 557 |
| 128 | 32 | 15715 | 4006 |
| 256 | 64 | 48401 | 12764 |
| 512 | 128 | 239097 | 65436 |
| 1024 | 256 | 1580359 | 489754 |
| 2048 | 512 | 12166397 | 3510342 |

Table 5 gives CPU the runtime on executing the Hoffman equivalent test algorithm and the proposed test algorithm on various lengths and dimensions of linear codes.

The CPU runtime recorded for the Hoffman Equivalent Test algorithm suggests that the time taken to deduce equivalency of linear code using the algorithm grows exponentially whilst that of the proposed algorithm increases constantly.

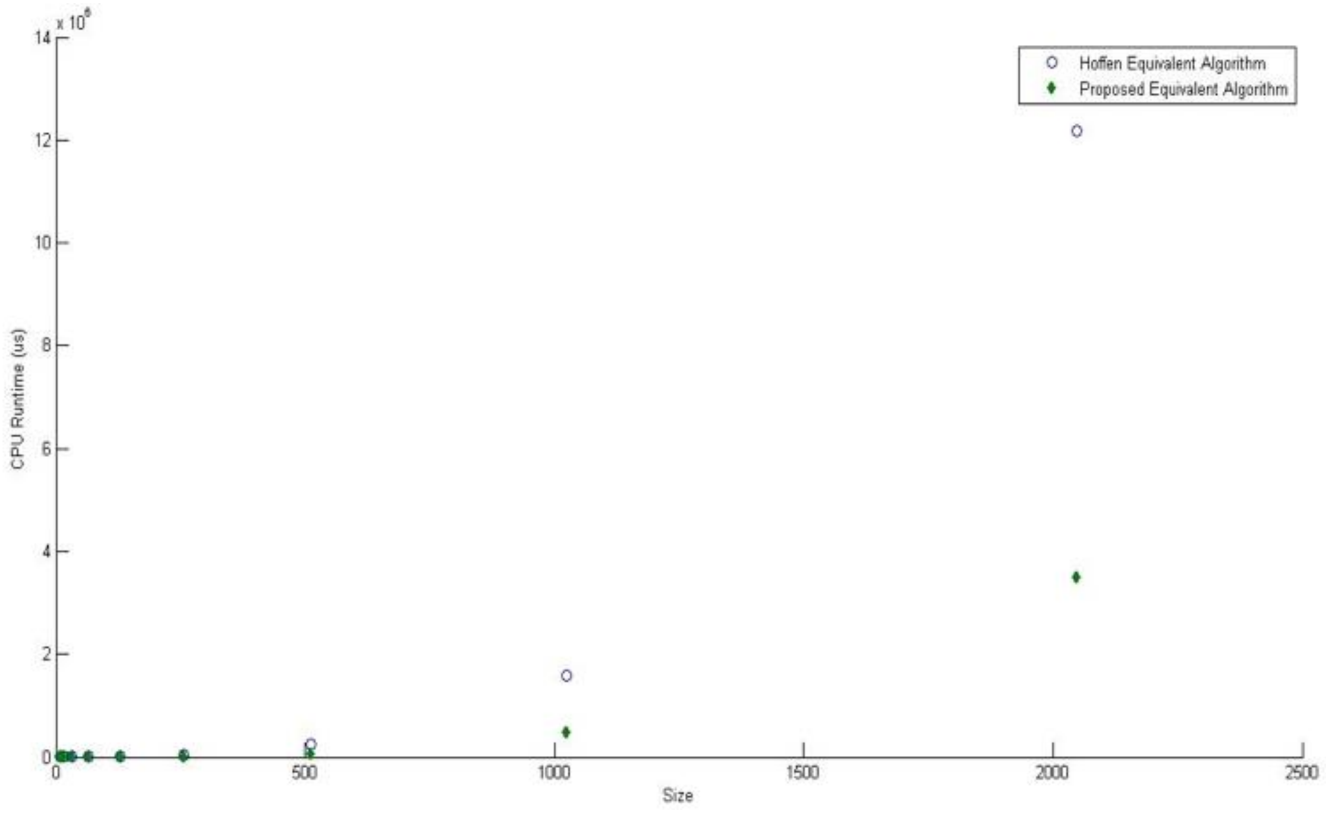

*Figure 2. Scatter plot showing relationship between the algorithms CPU runtimes against the various length of Linear codes.*

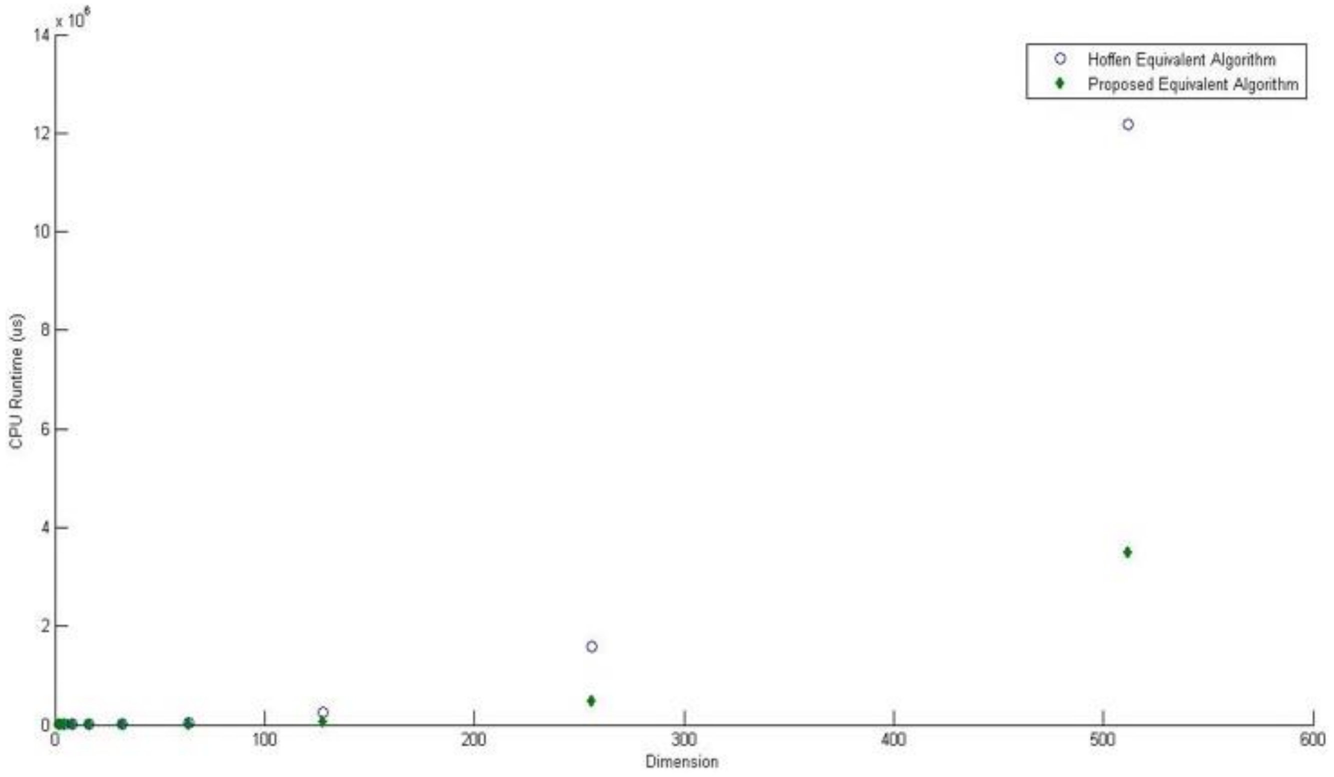

***Figure 3.*** *Scatter plot showing relationship between the algorithms CPU runtimes against the dimensions of Linear codes.*

Figures 2 and 3 clearly show that with Hoffman equivalent test algorithm, the CPU runtime increases rapidly as the Length (n) and dimension (k) of linear codes increase as oppose to that of the proposed algorithm hence the proposed algorithm is therefore more efficient in deducing equivalency of linear codes.

## 3.5. Existence of Linear Codes Test

The following $[n, k, d]$ linear codes were used to measure the efficiency of Hoffman's algorithm and the proposed algorithm.

$$[8,2,4]$$

$$[16,4,4]$$

$$[32,8,4]$$

$$[64,16,4]$$

$$[128,32,4]$$

$$[256,64,4]$$

$$[512,128,4]$$

$$[1024,256,4]$$

$$[2048,512,4]$$

The chosen linear codes were tested to ascertain if they do exist using the theorem of Gilbert Varshamov bound as follows:

$$n = 8, k = 2, d = 4$$

$$= \binom{7}{0} + \binom{7}{1} + \binom{7}{2}$$

$$= 1 + 7 + 21$$

$$= 29$$

$$Also, 2^{8-2}$$

$$= 2^6 = 64$$

$$29 < 64\ true$$

Therefore, the linear code [8,2,4] does exists.

$$n = 16, k = 4, d = 4$$

$$= \binom{15}{0} + \binom{15}{1} + \binom{15}{2}$$

$$= 1 + 15 + 105$$

$$= 121$$

$$Also,\ 2^{16-4}$$

$$2^{12} = 4096$$

$$121 < 409\ \{true\}$$

Hence the linear code [16,4,4] exists.

$$n = 32, k = 8,\ \ n = 4$$

$$= \binom{31}{0} + \binom{31}{1} + \binom{31}{2}$$

$$= 1 + 31 + 465$$

$$= 497$$

$$Also, 2^{n-k}\ = 2^{32-8}$$

$$= 2^{24}$$

$$= 16{,}777{,}216$$

$$497 < 16{,}777{,}216\ \{true\}$$

This implies linear code [32,8,4] exists.

$$n = 64, k = 16, d =\ 4$$

$$= \binom{63}{0} + \binom{63}{1} + \binom{63}{2}$$

$$= 1 + 63 + 1953$$

$$= 2017$$

$$Also, 2^{64-16}$$

$$2^{48} =\ 281474976710656$$

$$2017 < 281474976710656\ \{\text{true}\}$$

This implies that linear code [64, 16, 4] exists

From the existence proof of the first four linear codes using the Gilbert Varshamov theorem, it can be concluded that higher dimensions in the format $[2n, k_1 + k_2, \min(2d_1, d_2)]$ will always exist.

## 4. Conclusions

This paper sought to optimise the Hoffman equivalent test algorithm and establish an efficient algorithm that can determine equivalency of linear codes and further use the optimised algorithm to enhance data access security. After, the generation of the optimised equivalent test, we then employed the approach in [20] for generating generator matrices from existing linear codes with parameter $[n, k, d]$ to generate our linear codes. Further, the efficiencies of the Hoffman equivalency test algorithm and that of the proposed algorithm were subjected to check for equivalency of linear codes with parameters [8,2,4], [16,4,4], [32,8,4], [64,16,4], [128,32,4], [256,64,4], [512,128,4], [1024,256,4] and [2048,512,4]; the CPU runtimes of the two algorithms were measured using the C++ Chrono library and the results visualised with scatter plots. It was clearly observed that the CPU runtime of the Hoffman algorithm increases rapidly as the dimension and size of the linear codes increase whiles the increase in runtime of proposed algorithm is minimal. Thus, the proposed algorithm is a more efficient route for the equivalency test of linear codes.

## Abbreviations

| | |
|---|---|
| CFS: | Courtois-Finiasz-Sendrier |
| CPU: | Central Processing Unit |
| F_q: | Denotes a Finite Field of q Elements |

## Author Contributions

**Olufemi Ololade Olaewe:** Conceptualization, Software, Methodology, Writing – original draft

**Peter Awonnatemi Agbedemnab:** Supervision, Methodology, Writing – original draft

**Mohammed Muniru Iddrisu:** Supervision, Methodology

## Conflicts of Interest

The authors declare no conflicts of interest.

## References

[1] W. K. Harrison, J. Almeida, M. R. Bloch, S. W. McLaughlin, and J. Barros, "Coding for secrecy: An overview of error-control coding techniques for physical-layer security," *IEEE Signal Processing Magazine*. pp. 41–50, 2013. https://doi.org/10.1109/MSP.2013.2265141

[2] N. Sendrier, "Finding the permutation between equivalent linear codes: the support splitting algorithm," *IEEE Trans Inf Theory*, pp. 1193–1203, 2000, https://doi.org/10.1109/18.850662

[3] N. T. Courtois, M. Finiasz, and N. Sendrier, "How to Achieve a McEliece-Based Digital Signature Scheme," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 2248, pp. 157–174, 2001, https://doi.org/10.1007/3-540-45682-1_10

[4] M. Bardet, A. Otmani, and M. Saeed-Taha, "Permutation Code Equivalence is Not Harder Than Graph Isomorphism When Hulls Are Trivial," *IEEE International Symposium on Information Theory - Proceedings*, vol. 2019-July, pp. 2464–2468, Jul. 2019, https://doi.org/10.1109/ISIT.2019.8849855

[5] D. G. Hoffman, Wal, D. A. Leonard, C. C. Lidner, K. T. Phelps, and C. A. Rodger, *Coding Theory: The Essentials*. USA: Marcel Dekker, Inc., 1991.

[6] S. Gayathri Devi, K. Selvam, and S. P. Rajagopalan, "An abstract to calculate big o factors of time and space complexity of machine code," in *IET Conference Publications*, 2011, pp. 844 – 847. https://doi.org/10.1049/cp.2011.0483

[7] T. Dobravec, "Estimating the time complexity of the algorithms by counting the Java bytecode instructions," in *2017 IEEE 14th International Scientific Conference on Informatics, INFORMATICS 2017 - Proceedings*, 2018, pp. 74–79. https://doi.org/10.1109/INFORMATICS.2017.8327225

[8] M. Zugelj, "Empirical Analysis Complexity of Algorithm," University of Ljubljana, 2019.

[9] M. Cherkaskyy and H. K. Murad, "Modern Problems of Radio Engineering, Telecommunications and Computer Science," Institute of Electrical and Electronics Engineers (IEEE), Jul. 2006, pp. 45–45. https://doi.org/10.1109/tcset.2002.1015835

[10] D. Zaparanuks and M. Hauswirth, "Algorithmic profiling," in *ACM SIGPLAN Notices*, 2012, pp. 67–76. https://doi.org/10.1145/2345156.2254074

[11] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2007*, 2007, pp. 395–404. https://doi.org/10.1145/1287624.1287681

[12] J. Thiyagalingam, S. Walton, B. Duffy, A. Trefethen, and M. Chen, "Complexity plots," *Computer Graphics Forum*, pp. 111–120, 2013, https://doi.org/10.1111/cgf.12098

[13] T. R. Kelley, "Optimization, an Important Stage of Engineering Design," 2010, Accessed: Oct. 22, 2021. [Online]. Available: https://digitalcommons.usu.edu/ncete_publications

[14] L. Diego, "Essential Programming | Time Complexity," Towards Data Science. [Online]. Available: https://towardsdatascience.com/essential-programming-time-complexity-a95bb2608cac

[15] B. Stroustrup, *the C++ Programming Language 4Th Edition*. 2013.

[16] "Measure execution time of a function in C++ - GeeksforGeeks." Accessed: Oct. 22, 2021. [Online]. Available: https://www.geeksforgeeks.org/measure-execution-time-function-cpp/

[17] Khan Academy, "Measuring an algorithm's efficiency | AP CSP (article)." Accessed: Oct. 22, 2021. [Online]. Available: https://www.khanacademy.org/computing/ap-computer-science-principles/algorithms-101/evaluating-algorithms/a/measuring-an-algorithms-efficiency

[18] R. M. Kubina, D. E. Kostewicz, K. M. Brennan, and S. A. King, "A Critical Review of Line Graphs in Behavior Analytic Journals," *Educational Psychology Review*. pp. 583–598, 2017. https://doi.org/10.1007/s10648-015-9339-x

[19] J. D. Lane and D. L. Gast, "Visual analysis in single case experimental design studies: Brief review and guidelines," *Neuropsychological Rehabilitation*. pp. 445–463, 2014. https://doi.org/10.1080/09602011.2013.815636

[20] A. Ibrahim, P. Chun, and N. Kamoh, "A New [14 8 3]-Linear Code From the Aunu Generated [7 4 2] -Linear Code and the Known [7 4 3] Hamming Code Using the (U|U+V) Construction," *Journal of Applied & Computational Mathematics*, vol. 07, no. 01, pp. 1–3, 2018, https://doi.org/10.4172/2168-9679.1000379