Foundations and Trends® in
sample
Vol. xx, No xx (xxxx) 1–75

DOI: xxxxxx

now
the essence of knowledge

A Universal 4D Model for Double-Efficient Lossless Data Compressions

Philip Baback Alipour[1]

[1] *Dept. of Electrical and Computer Engineering, University of Victoria, Victoria, B.C. V8W 3P6, Canada, phibal12@uvic.ca*

Abstract

This article discusses the theory, model, implementation and performance of a combinatorial fuzzy-binary and-or (FBAR) algorithm for lossless data compression (LDC) and decompression (LDD) on 8-bit characters. A combinatorial pairwise flags is utilized as new zero/nonzero, impure/pure bit-pair operators, where their combination forms a 4D hypercube to compress a sequence of bytes. The compressed sequence is stored in a grid file of constant size. Decompression is by using a fixed size translation table (**TT**) to access the grid file during I/O data conversions. Compared to other LDC algorithms, double-efficient (DE) entropies denoting 50% compressions with reasonable bitrates were observed. Double-extending the usage of the **TT** component in code, exhibits a Universal Predictability via its negative growth of entropy for LDCs $> 87.5\%$ compression, quite significant for scaling databases and network communications. This algorithm is novel in encryption, binary, fuzzy and information-theoretic methods such as probability. Therefore, information theorists, computer scientists and engineers may find the algorithm useful for its logic and applications.

Contents

1

Introduction

One of the greatest inventions made in Computer Science, as a building-block for its logical premise was Boolean Algebra, by the well-known mathematician, G. Boole (1815-1864). Its foundation on Boolean operators enlightened further, the great mathematician C. E. Shannon (1916-2001). In 1938, this leading scholar, with reference to Boolean operators [9], managed to show how electric circuits with relays were a suitable model for Boolean logic [43]. Hence, a model for Boolean logic, as a sequence of 0's and 1's, constituted binary [12]. From there, he measured information by quantifying the involved *uncertainty* to *predict* a random value, also known as *entropy*. He thus inducted this new entropy with codeword to compress data, losslessly. During this venture of computational science in progress, another mathematician came up with fuzzy sets theory, L. A. Zadeh (1921-present), resulting fuzzy logic with its algorithmic constructs and applications [28, 51].

In this paper, we put all of these scholars' findings into one *logic synthesis*. Coding this combinatorial logic by *biquaternions* [23], *self-contains* any randomness occurring in a 4D field, delivering a *universal predictability*. Contrary to the notion of randomness, which states: "the more random, i.e. unpredictable and unstructured the variable is, the

larger its entropy" [25, 39], by "self-containing" the random variable, we then stipulate

---

Hypothesis 1.1. The more random a biquaternion field contains i.e. unpredictable and unstructured the variable in a 4D subspace containment, the smaller its entropy.

In other words, containing complexity, like a scalable cannon containing a cannonball before ejection, allows complexity's dynamic vectors to remain in *containment* to *relatively reach* the end drop coordinates as a unified result, quite akin to the complexity of all of our universe's randomness *contained* in a dot (or a unifying equation, comparatively [22]). If the "complexity vectors" are unleashed from any application, obviously, uncertainty or randomness is emerged.

According to Shannon, "a long string of repeating characters has an entropy rate of 0, since every character is predictable" [12], whereas Hypothesis 1.1 self-contains any randomness coming from a string of non-redundant characters, attaining an entropy of 0 bits per character (bpc). If achieved, Hypothesis 1.1, for an observer of the variable, delivers a Universal Predictability theorem:

Theorem 1.1. As the field's entropy grows negatively, i.e. becoming smaller and smaller, its curve gives an observer of the information variable a predictable output.

To prove this "containment of information variable," from Hypothesis 1.1, resulting Theorem 1.1, we have no need to minimize multi-level logic. In fact, we need to combine logic states correctly using standard and custom operators to obtain *losslessness*. The "variable containment," is later indicated as $y \in xx'$, which further involves *fuzzy binary and-or operators* to confine the output y content representing the input xx' content. This is introduced as FBAR logic, entailing its fixed compression entropy for its information products throughout the following sections.

1.1 Overview

This paper aims to introduce FBAR logic, apply it to information in a model, causing data compression. The compression model is constructed after introducing the theory of FBAR. From there, its usage and implementation in code are discussed. Furthermore, a clarification between model representation and logic is established for both, the FBAR algorithm and its *double-efficient* (DE) *input/output* (I/O) evaluation. The evaluation on the algorithm's efficiency is conditioned by conducting two steps:

(1) data compaction and compression processes, using a new bit-flag encoding technique for a lossless data compression (LDC),
(2) validating data at the other end with the bit-flag decoding technique for a successful lossless data decompression (LDD).

We introduce FBAR logic from its theoretical premise relative to model construction. We further implement the model for a successful LDC and LDD. The general use of the algorithm is aimed for current machines, and its advanced usage denoting maximum DE-LDCs for future generation computers.

This article is organized as follows: Section 2 gives background information on FBAR model, and its universality compared to other algorithms. It concludes with Subsection 2.6 introducing FBAR synthesis with expected outcomes. Section 3 focuses on FBAR LDC/LDD theory, model and structure. It introduces FBAR test on data by model components, functions, operators, proofs and theorems. Section 4 presents implementation. Section 5 presents the main contribution made in this work. Subsection 5.2 describes the experiment on DE performance including results. Subsection 5.3 onward, end the paper with costs, future work and conclusions.

2

The Origin of FBAR Logic

In this section, we review a wide range of existing mathematical theories that are relevant to the foundation of FBAR logic, its model structure arising in lossless data compressions. We also introduce the universal model with a universal equation applicable to LDC algorithms, both in theory and in practice, to perform double-efficient compression as well as communication. Throughout the monograph, the coding theory subsection formulating the four-dimensional model, employs bivector operators to manipulate data symmetrically in the memory's finite field. That is done with real and imaginary parts of bit-state revolutions as high-level 1, or low-level 0 signals, where data is circularly partitioned and stored in the field. We express such operations in form of integrals denoting bivector codes. The memory field equations are integrable when data compaction, compression and four-dimensional field partitioning are both complex and real during communication.

2.1 Motivation and Related Work

We at first questioned the actual randomness behavior coming from regular LDC algorithms in their compression products. No matter how

highly ranked and capable in compressing data observed on dictionary-based LDCs e.g., LZW, LZ77, WinRK, FreeArc [8, 53], they still remain probabilistic for different input types [41]. These algorithms are mainly based on repeated symbols within data content [12, 34]. For example, a compressed output with a *string* = `[16a]bc` is interpreted by the algorithm as `aaaaaaaaaaaaaabc` when decompressed (assuming this was the original data). The *length* of the input string is 16 B, and for the compressed version is 7 B, thus we say a 56.25% compression has occurred. We assess its entropy as Shannon-type inequality, since it minimally involves two mutual random variables [1, 36] for the recurring symbols in context.

For such random behavior performed by LDC algorithms sold on the market, the question was whether it would be possible to somehow confine randomness whilst LDC operations occur. This statement motivated the concept of combining the well-known logics to address randomness, both in theory and in practice.

In modern machines, each ASCII character entry from a set of $\geq 2^{7\,\mathrm{bit}}$code groups, occupies 8 bits or more of space, in which, each bit is either, a low-state or high-state logic. These logic states in combination, build up a character information or their corresponding symbol [35, 38]. To perform the least probability of logic operations, there must be a definite relatedness between binary logic and its in-between states of low and high for each corresponding symbol. In FBAR logic, this could be recognized at its lowest layers of binary logic between AND and OR operations. Once these operators with negation are applied to original data, manipulating a byte length of pure bits e.g., '`11111111`' to obtain original data, 8 bits of 0's and 1's is therefore transmitted. This is possible if bivector operators manipulate data in a 4D subspace $\mathbb{R}^4$ [29], with a minimally 4 fuzzy bits, thereby, 2 pairwise bits producing compressed data. This *encoding-decoding method* further gives a compression on 2-byte inputs as a reversible 1-byte output, denoting a DE-transmission. This transmission, suggests the relatedness implementation or proof of all logics in FBAR model and relationships.

2.2 Relatedness of Logic Types

The relatedness for each character entry on a binary construct is presented by the logical consequence [52] from different models: fuzzy logic [28, 51, 52], binary, and transitive closure [26, 45]. By making this uniformity, FBAR logic is emerged. This logic is possible when packets of Boolean values per character are updated and abstracted into relative states of fuzzy and pairwise logic. When we conceive F, B, AND/OR, each, as a separate field in calculus, we also conclude that each has its own founder, i.e., chronologically: Boole (1848) [9], Shannon (1948) [44] and Zadeh (1965) [51]. Therefore, for establishing a combinatorial logic model, we question that:

- Why not uniting the binary part with the highly-probable states of pairwise logic via fuzzy logic?
- Is there a way to assimilate the discrete version F, B, AND/OR, into one unified version of all, FBAR?
- Would this unification lead to more probability or else, in terms of predictability?
- If predictable, what is the importance of it, compared to random states of codeword results?

To address each question, it is essential to establish FBAR logic in a combinatorial sense. In essence, the information models known in Information Theory, must be brought into a standard logical foundation as FBAR, representing their logic states combination, computation, information products and application, respectively.

2.3 The Foundation of FBAR Model and Logic

2.3.1 Logarithmic and Algorithmic Premise

Here onward, we use Table 2.1 notations and definitions. For subspace fields, to store, compress and decompress data, we adapt and refer our main findings to Hamilton (1853) [23], Conway (1911) [15], Lanczos (1949) [29], Bowen (1982) [10], Girard (1984) [20], Lidl and Niederreiter (1997) [31], and Coxeter *et al.* (2006) [16]. Moreover, the algorithmic premise for our algorithms is formulated on the logarithmic preference

of information metric as log base 2, which measures any binary content for a character communicated in a message. Foremost, the premise to achieve self-containment on any information input, is to mathematically elaborate on this compression theorem:

Theorem 2.1. Any probability P on information variable y is 1, if y as a single-character output is contained within the binary intersection limits of its input xx'.

Theorem 2.1 lays out the foundation of self-containing xx' as y in preserving all probability $p(xx') \to P(y) \to 1$ counts, against any "surprisal" as a highly improbable outcome $p(xx') \to 0$ or uncertainty $u \to \infty$ [49]. The current goal is to "self-contain" xx' within the limits of *self-information* I measure on y.

Now consider the definition revisited by Bush (2010) [11] on "self-information" as:"a measure of the information content associated with the outcome of a random variable." Further, "the measure of self-information is positive and additive." In contrast, as we prove in Section 3, *the measure of self-containment is positive and conjunctive, but not additive*. So, any binary content as a given input is partitioned in its *dual space output* [32] when contained by 4D *bivector operators*, or

---

Definition 2.1. Self-information containment is associative in binary states of a given input, preserving its equally combined additive and conjunctive function using 4D bivector code operators, returning a constant size output stored in an array A.

This associativity between logic states in A, returns an information constant as data_{in} in entanglement or a bivector DE-coding. The coding objective is to put all logic states of an information input into two places at once as a unique address in A. The array stores an event as an output character y denoting two original events as input characters xx'. In essence, suppose event $\Gamma = y$ content is composed of two mutually independent events Θ as x content, and Λ as x' content. The amount of information when Γ is "communicated" equals the *combination* of the amounts of information at the communication of

Table 2.1 Notations and terminology for LDC operations.

Notation	Short definition	Example
$\mathcal{C}_r$	Data compression ratio	2:1 compression
$\mathcal{C}$	Compressed data; compression	$\mathcal{C}_{-1} > \mathcal{C}_n\ ,\ n \in \mathbb{N}$
$\mathcal{C}'$	Decompressed data; decompression	$\mathcal{C} \overset{\text{out}\times\text{ref}}{\longrightarrow} \mathcal{C}'$
H	Entropy rate in e.g., Shannon systems	$H_{\mathbb{A}} > H_{\wedge\vee(b)}$
x_i	A bit, byte or character by scale, where $i \in \mathbb{N}$	$\{x_1x_2x_3\ldots\}$
y	Product of a function, or output	$f(x) = y$
$\int$	A sequence of an entailed complement xx' (see $\therefore$), or just concatenated values of x_i	$\int_{\text{in}} = \int(x) = x_1{+}x_2{+}x_3 {+}\ldots = \{x_1x_2x_3\ldots\} = \int_{\text{out}}$
ℓ	Length function on field, string, time, etc.	$\ell\ (xx') = 16$ bits
∞	Infinity; continuous flow of I/O data, in	if $\mathcal{U}_{\text{I/O}} = \{\infty\}$ then
$\emptyset$	*measure theory* [6] measured by chars in the flow. $\emptyset$ denotes a null set	$(\{\infty\} - \int(x))^c$ $= \emptyset \cup \int(x) = \int(x)$
$\mathbb{R}^{2^n}$	A 2^nD-product space with a topology of mapping bit-pairs of input characters into subspace partitions, where $n = 2$ characters.	$\forall xx' \in \mathbb{R}^{2^n}\,;\, n = 2\,,$ $xx' \mapsto \{y\}_{i,j,k,l\in\mathbb{R}^4}\ ,$ $\therefore \hat{\boldsymbol{v}}_y = \frac{1}{\sqrt{2^n}}xx' = 1\text{B}$ $= \left(\frac{1}{\sqrt{2}}x_i\ \frac{1}{\sqrt{2}}x_j\ \frac{1}{\sqrt{2}}x'_k\ \frac{1}{\sqrt{2}}x'_l\right)$
$\hat{\boldsymbol{v}}$	A spatial unit vector = 1 bit, 1 byte, etc.	
$\mathbf{e}_{ij}$	A unit bivector for bit-pair mappings	$\forall \mathbf{e}_{ij} \in \mathbb{C}\ell_4\mathbb{R}^4\,;\, \mathbf{e}_{12}^2 = -1$
A	An array for the residing bits in memory	if $\beta \vdash x_i = 0$ then , $A_{1\times n} = [000\ldots0]$
$\vdash$	A sequent; derived from; yields ...	
β	Binary value or sequence, where $\forall\beta \in f(x) = x \to y = b$	if $\beta \vdash x_i = 0$ and $x'_i = 1$ $\therefore \beta = 01010101$
$\wedge$, $\bigcap$	Logical AND; for sets as Intersection	$1 \wedge 0 = 0\ , 1 \wedge 1 = 1$
$\vee$, $\bigcup$	Logical OR; for sets as Union	$1 \vee 0 = 1\ , 0 \vee 0 = 0$
$\leftrightarrow$	Bi-conditional between states or logic; if and only if; iff	$x \leftrightarrow y \equiv$ $(x \to y) \wedge (y \to x)$
$\equiv$	Equivalence; identical to ...	2 chars $\equiv$ 16 bits
$\therefore$	Logical deduction; therefore ...	if $\{x_1x'_1\} = \{\$2\%1\}$, $\therefore x_1 = \$2\ ,\ x'_1 = \%1$
component	Algorithm component as an I/O object, **P** as a program with filter, **G** as a grid file, **TT** as a translation table	$\int \overset{\text{in}}{\to} \boxed{\mathbf{P}} \overset{\text{out}}{\to} \mathcal{C}$
$\otimes$	Strong conjunction on array values; matrix vector or finite field product	$\{8\,\text{bits}\} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 2 & 3 \end{bmatrix} =$ $\begin{bmatrix} 8\,\text{bits} & 0 & 0 \\ 8\,\text{bits} & 8\,\text{bits} & 0 \\ 8\,\text{bits} & 8\,\text{bits} & 8\,\text{bits} \end{bmatrix}$ $= \{6\ \text{bytes}\}$

[a] These notations are used in defining LDC operations between algorithmic components, model and logic. Those notations that are not listed here, are defined throughout the text, or in the ending section 'Notations and Acronyms', before 'References'.

[b] Some notations imply bivectors [32], entropy and complexity (Sections 3.2, 4-5.2).

event Θ and event Λ, simultaneously.

Coding objective. The strong conjunction '$\otimes$' of xx' contents must satisfy their binary "combination": x as $\geq 1\,\mathrm{B}$ intersected with x' as $\geq 1\,\mathrm{B}$, gives a $y = \frac{1}{\sqrt{2^n}}xx' = \frac{1}{\sqrt{2}}x\frac{1}{\sqrt{2}}x' = \frac{1}{2}xx'$ content size in a locally-compact space $\mathbb{R}^N$ on A. Taking into account that y would be the compressed content of xx', located in a specific subfield address. The subfield as $\mathbf{F}_{xx'}$ is a dual space partitioning x and x' bit-pair values into 4 bivector dimensions, hence the notion $\frac{1}{\sqrt{2^n}}xx'$, partitioning $N = 4$ into 2^n. The field's range is of single-byte addresses or rows r available to store y, such that

$$I(\Theta \cap \Lambda) \cap \max\{\mathbf{F}_{xx'}\} = \{I(\Theta) + I(\Lambda)\} \otimes A_{r\times N} = I(\Gamma) \otimes A_y = I\,(A_\Gamma)$$

This relation portrays Definition 2.1, and its rows or address limit is established by

---

Lemma 2.1. A single-byte input x has $2^8 = 256$, $\{0, 1\}$ bit combinations, r rows 0 to 255. Thus, for a 2-byte input xx', we possess $k = 2^{(8+8)} = 65536$ combinations as the maximum range of its finite field $\mathbf{F}_{xx'} \in \mathbb{R}^4$ addresses, building an array $A_{k\times 4}$.

The lemma holds even if all pairs of bytes input, xx', in the order of $2, 4, 8, \ldots, 2^n$ as the number of characters are compressed into 1 byte output, y. Once a *translation* on the *intersection of combinations* are decoded, a lossless decompression is gainable from the given *Coxeter order* [16] In:Out as xx':y= 2:1, 4:1, 8:1, $\ldots, 2^n$:1. Henceforth, this ordered sequence of ratios is called "double-efficiency" for any transmitted data as compression relative to its successful decompression. Reference to Lemma 2.1 in aim of proving Theorem 1.1, we then propose

---

Proposition 2.1. If a binary intersection of x and x' by a 4D bit-flag function φ produces y, translating the intersected bivector combinations by a ϑ function, conversely produces xx'. These flags have a physical space occupation of $\begin{pmatrix} h^2\mathbf{e}_{12}^2 & h^2\mathbf{e}_{34}^2 \end{pmatrix}$.

We investigate this new type of logic directed to one LDC-DD algorithm:

Algorithm 2.1.

$\forall P(y) \subset P(xx')$; if $\varphi(xx') = \{\log_2 65536 \cap \log_2 65536\} = 8\,\text{bits} = y$
$\in 65536 \begin{pmatrix} h^2\mathbf{e}_{12}^2 & h^2\mathbf{e}_{34}^2 \end{pmatrix} = \begin{pmatrix} 2\,\text{bits} & 2\,\text{bits} & 2\,\text{bits} & 2\,\text{bits} \end{pmatrix}_{64\,\text{K}}$,
then $P\left(\vartheta(y)\right) \to P(xx') = \vartheta\left(\log_2 65536 \cap \log_2 65536\right) \to \{\log_2 65536 \cup$
$\log_2 65536\} = 16 \times 16 \times 16 \times 16bit - flags \to 16\text{bits}$ original data.
Thus, probability $P\left(\vartheta(y)\right) \to P(xx') = 1$.

Proposition 2.1 gives a predictable outcome for all probability scenarios on the compressed y with a P representing xx' contents for a lossless decompression. Predictability is achieved only if the *complete algorithm*, Algorithm 2.1, runs according to its $\cap$ and $\cup$ operations. It should *configure bit-flag* φ and *execute code translation* ϑ function with relevant operators on xx' and y. The general use of $\cap$ and $\cup$ operators are expressed by the universal FBAR equation in Section 2.4.

What we mean about "universal predictability" is that irrespective of the number of inputs, the output is predictable before logic state combinations. As a result, *invariant entropies of higher order* (*negentropy* [25]) with more complex coding, becomes predictable. We synthesize all of the presented logics into this combinatorial logic via logical operators as categorized in Section 2.6.

2.4 A Universal FBAR Coding Model and Equation

We first commence with an assumption

Assumption 2.1. Let an information input x to our machine lie in the interval $[0, 1]$. Assume function f operates on x between its logic states as binary, otherwise fuzzy, producing x with a new value. Let the machine produce this value via standard logical operators as: and $\wedge$, or $\vee$, union $\cup$, intersection $\cap$, and negation $\neg$.

---

and then we define its universal relation, once proven in terms of

Definition 2.2. Relation $\mathfrak{R}$ is a universal relation, if presented with union $\cup$, and intersection $\cap$, between fuzzy set $\tilde{\mathcal{A}}$ and binary set $\mathcal{A}$ simultaneously. If $\mathfrak{R}{=}\cup$, $\mathcal{A}\mathfrak{R}\tilde{\mathcal{A}}$ succeeds in many pairwise states $\mathcal{A}|\tilde{\mathcal{A}}$. Conversely, if $\mathfrak{R}{=}\cap$, $\{\mathcal{A}|\tilde{\mathcal{A}}\}\mathfrak{R}\tilde{\mathcal{A}}$ succeeds in binary states 0 or 1. Thus, a fuzzy-binary function Φ for all $\mathfrak{R}$'s is given by

$$\begin{aligned}\Phi_{\wedge\vee}(x) &= \mathcal{A}\mathfrak{R}\tilde{\mathcal{A}}\mathfrak{R}\left\{\mathcal{A}|\tilde{\mathcal{A}}\right\}\mathfrak{R}\mathbb{C}\ell_4\left(\mathbb{R}^{2^n}\right)\\ &\equiv \{0,1\}\leftrightarrow\{[0,1]\}\leftrightarrow\{00,01,10,11\}\ ,\ n=|\mathcal{A}\cup\tilde{\mathcal{A}}|\leq 4\end{aligned} \tag{2.1}$$

where both finite sets $\mathcal{A}$ and $\tilde{\mathcal{A}}$ membership values are contained by dual carriers as bivectors in a partitioned real field $\mathbb{R}^{2^n}$, projecting bit-pair values from a real or complex plane in $\mathbb{C}(\mathbb{R})$ with a dimensional length ℓ_i. Based on the *inclusion-exclusion principle* [5], n is equal to the number of elements in the union of the $\mathcal{A}$ and $\tilde{\mathcal{A}}$ as the sum of the elements in each set respectively, minus the number of elements that are in both, or $|\mathcal{A}\cup\tilde{\mathcal{A}}|=|\mathcal{A}|+|\tilde{\mathcal{A}}|-|\mathcal{A}\cap\tilde{\mathcal{A}}|$.

Furthermore, from the well-known scholars, we plug the latter definition into their scalar bivector definitions, hence deducing

Definition 2.3. In Eq. (2.1), a scalar element $h=\sqrt{-1}$ by Conway (1911) [15], its dual and quadruple forms, $h^2=-1$ and $h^4=1$, respectively satisfy combinatorial operators during 2^nD and 4D bit-pair spatial partitioning and projections. The 2^nD type projections are of Coxeter group in the order 2, 4, 8, ... by Coexeter *et al.* (2006) [16], and configure matrix vertices to store, compress and decompress data in a hypercube.

Now, we begin the proof of the universal model by a theorem,

Theorem 2.2. Relation $\mathfrak{R}$ is universal, iff $\mathfrak{R}=\{\cup,\cap\}$ on all logic states stored in a dual space, yield from a 2^nD $\leftrightarrow$ 4D bivector field, where n is the possible number of pairable states. This produces a combinatorial fuzzy-binary and-or equation.

The following equations prove the relatedness of all probable logics from one side of Eq. (2.1) to another:

Proof. The fuzzy unit in its membership function $\mu(x)$ with a numerical range covering the interval [0, 1] operating on all possible values, gives minimally $n \geq 3$ possible states [40, 51]. Binary, however, in its set is discrete for a possible 0 or 1. Let for $\mu(x)$, fuzzy membership degrees close to 0 converge to 0, and those close to 1 converge to 1 as an independent state with a periodic projection (an integral), stored onto a closed surface of 2D planes for a bivector decision $\hat{\boldsymbol{\mu}}$. This decision is derived from such input values projected into a dual space forming a hypercube. That is the space $\bigwedge^2 \mathbb{R}^4$ dual to itself in $\mathbb{C}\ell_4(\mathbb{R})$ by Lounesto (2001) [32], where every plane of data (in form of bitpairs) is orthogonal to all vectors in its dual space. The projection for a given bit is done by 2^n-bivectors, partitioning the input as bitpairs into the space. Now having a *dual space output*, by using the Pythagoras' theorem, the output covers a projection of x from either set $\tilde{\mathcal{A}} = \{\approx 0, \approx 1\}$ or $\mathcal{A} = \{1, 0\}$, as a *hypotenuse transformation* ϑ, in terms of

$$\begin{aligned}
\vartheta\left(\hat{\boldsymbol{\mu}}_{\mathcal{A},\tilde{\mathcal{A}}}\right) &= \hat{\boldsymbol{\mu}}_x \sqrt{\left(h_{12}\mathbf{e}_{12} + h_{34}\mathbf{e}_{34}\right)^2} \equiv \left\{ \hat{\boldsymbol{\mu}}_x \|\mathbf{e}\| = \hat{\boldsymbol{\mu}}_x \sqrt{\mathbf{e}\cdot\mathbf{e}} \right\} \\
&= \hat{\boldsymbol{\mu}}_x \sqrt{h_{12}^2\mathbf{e}_{12}\mathbf{e}_{12} + h_{1234}^4\mathbf{e}_{12}\mathbf{e}_{34} + h_{1234}^4\mathbf{e}_{34}\mathbf{e}_{12} + h^2\mathbf{e}_{34}\mathbf{e}_{34}} \\
&= \hat{\boldsymbol{\mu}}_x \sqrt{-\mathbf{e}_{12}^2 + 2\mathbf{e}_{1234}^2 - \mathbf{e}_{34}^2} = \hat{\boldsymbol{\mu}}_x \sqrt{2 + 2\mathbf{e}_{1234}^2} \\
&= 1\sqrt{4} = 2 \text{ bit states per } \hat{\mu} \text{ vector} \vdash \left\{\mu_{\tilde{\mathcal{A}}}, \mu_{\mathcal{A}}\right\}, \\
\therefore \forall x \in \int_{S_A} \hat{\boldsymbol{\mu}}_x \,\mathrm{d}\,\alpha &\begin{cases} \lim_{x\to\min(x)} f\left(\hat{\boldsymbol{\mu}}_{\mathcal{A}\cap\tilde{\mathcal{A}}}(x)\right) = (0 \leftarrow \{\approx 0\})_{\mathbf{e}_{ij}} = 0 \,, \\ \lim_{x\to\max(x)} f\left(\hat{\boldsymbol{\mu}}_{\mathcal{A}\cup\tilde{\mathcal{A}}}(x)\right) = (1 \leftarrow \{\approx 1\})_{\mathbf{e}_{ij}} = 1 \end{cases}
\end{aligned} \tag{2.2a}$$

where $\mathbf{e}_{ij}^2 = -1$, and its product $\mathrm{d}\alpha$, is the area element of array surface S_A, occupied by a bit or x, thus its full frequency occupation is 2-bit states and converges to a $\pm 2k\pi$ radian of the projections made onto hypercubic planes (lattices). We obtain this by coding a *surface-volume integration*, modeled in Fig. 2.1. It shows that a *compression hypercube* Q_y containing encoded data is formed like a *tesseract* [10], when at least $\theta = 4\pi$ radians occur. It minimally contains bit-pair values, and

maximally 2 B, or a closed pair of characters as an xx' message, stored into two places at once. Hence, by a *cylinder method* [3], we then elicit a combinatorial integral

$$\begin{aligned} V_{Q_y} &= \hat{\boldsymbol{\mu}} \int_{S \to V_A} \vartheta\left(x, \mathbf{e}_{ij}\right) \, \mathrm{d}\, \alpha = 2\pi \int_{\frac{x}{\ell_4}}^{x'} \rho_x \phi_x \, \mathrm{d}\, x = 2\pi \int_{\mathbf{e}_{12}}^{\mathbf{e}_{34}} x \left|\vartheta\left(\hat{\boldsymbol{\mu}}_x\right)\right| \, \mathrm{d}\, x \\ &= \Delta \mathbf{e}\left(\circ, \mathbf{s}\right)_A = \hat{\boldsymbol{\mu}}_x \mathrm{d} S_\circ \to \hat{\boldsymbol{\mu}}_x \mathrm{d} V_\circ = \mathrm{d}\rho \hat{\boldsymbol{\rho}} + \rho \mathrm{d}\theta \hat{\boldsymbol{\theta}} + \mathrm{d}\phi \hat{\boldsymbol{\phi}} \to \rho \mathrm{d}\rho \mathrm{d}\theta \mathrm{d}\phi \\ &= \tfrac{1}{2} \pi \hat{\boldsymbol{\mu}}_x \|\mathbf{e}\| \to 2\pi \hat{\boldsymbol{\mu}}_x \|\mathbf{e}\|^2 \\ &= S_\bullet \xrightarrow[\pm\pi]{\cap} V_\bullet = \left(\tfrac{\sqrt{4}\pi}{2} \cap \tfrac{4\pi\sqrt{4}}{\pm\pi} \cap 2\pi\sqrt{4}^2\right) \approx (3, 25] \cap \left|\tfrac{25}{\pm\pi}\right| \\ &= (3, 8] \text{ bits} \in A_{\mathbb{R}^4} \end{aligned} \tag{2.2b}$$

where ρ_x is the area radius equal to the binary length of input x, in this case, quantified as a planar binary sum inclusion $\Phi_\wedge \sum \beta(x) = [2, 4)$ bits, and ϕ_x is the input projection equal to the magnitude of bivectors $\mathbf{e}_{ij}$ from Eq. (2.2a), in this case $\phi_x = \|\mathbf{e}\|$. Line element $\mathbf{s}$ by code integrates the ρ and ϕ quantities to form a cube Q with volume V by the bivectors when traversed, or, $\mathbf{s}\Delta\mathbf{e} = \mathbf{s}\left|\mathbf{e}_{34} - \mathbf{e}_{12}\right|$. Orthogonal vectors $\hat{\boldsymbol{\rho}}, \hat{\boldsymbol{\theta}}, \hat{\boldsymbol{\phi}}$, via $\hat{\boldsymbol{\mu}}$, denote $[2, 4]$ dimensions to store and sort data into 1 or more empty vertices $\circ$, of array A. The left integral result denotes an occupiable surface $S_\circ > 3$ bits of x via ρ_x, projected onto $\circ_x$ (stored as $\bullet_x$ via ϕ_x) producing $S_\bullet$ before forming V_{Q_x} via $\mathbf{e}_{ij}$. The overlapping results via $\cap$, denote a compressed volume of filled vertices, or $V_\bullet = \left|\frac{25}{\pm\pi}\right| \approx 8$ bits as: two cubes having 16 vertices built by the bivectors containing two input characters in a simultaneous $\pm\pi$-communication inside a big cube as 0's and 1's entanglement. This cube self-contains the subcubes in its subfield, a decodable xx' data packed into a y as its 8 outer-vertices (bits), in total 24 co-intersecting vertices involved. The 4D model is illustrated in Fig. 2.1.

Further associating both fuzzy results with binary, each as an independent state, in total builds four simultaneous bit-pairs (each subcube of 8 bits), thus giving

$$\begin{aligned} \Phi_\vee(x) &= \left(\forall x = 0 \in \mathcal{A} \cup \tilde{\mathcal{A}} \,|V_{Q_x}\right) \vee \left(\forall x = 1 \in \mathcal{A} \cup \tilde{\mathcal{A}} \,|V_{Q_x}\right) \\ &= \{0, 1\} \to \{[0, 1]\} \to \{\ 00, 01, 10, 11\} \end{aligned} \tag{2.3a}$$

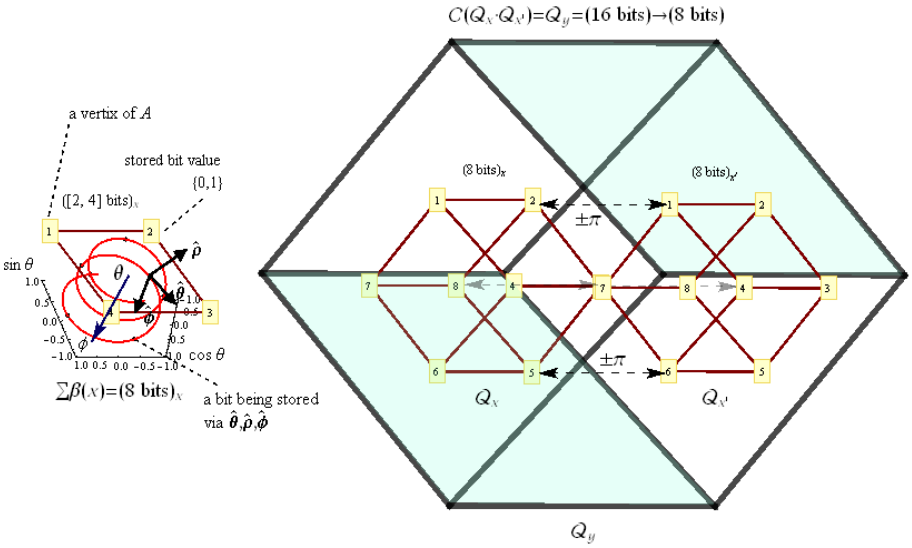


Fig. 2.1 Illustrates a geometric model of bit-pair projections over θ forming a side-by-side tesseract. (a) shows a projection from the θ plane with a growing revolution that forms a volume containing 8 bits; (b) shows two sub-cubes denoting a bit-set of at least 16 bits under compression within an 8-bit cube. The equation above the hypercube represents this compression. The two-input $\pm\pi$-communication denoting entangled bit-set states, is viable for superdense coding operators to store data into a single *qubit* [7], as our future quantum compression model.

This is pairwise logic for many possibly contained (compressed) values of fuzzy as well as binary, and in its default set covers 8 states. By using a fuzzy unit on each bit-pair, we abstract the pairwise version to binary, which is an inverse process, or

$$\begin{aligned}\Phi_{\wedge}(x) &= \left(\forall x = 0|\{0,1\} \wedge \forall x = 1|\{0,1\} \in \left\{\mathcal{A}|\tilde{\mathcal{A}}\right\} \cap \tilde{\mathcal{A}}\,|V_{Q_x} \to S_{Q_x}\right) \\ &= \{0,1\} \leftarrow \{[0,1]\} \leftarrow \{\ 00, 01, 10, 11\} \end{aligned} \tag{2.3b}$$

The *fuzzy-binary and* function $\Phi_{\wedge}$ uses logical operators *and-or*, *negate* and *closures* e.g., *transitive closure* [26] to generate crisp logic. Combining Eqs. (2.3a) and (2.3b), further outputs a *fuzzy-binary and-or* $\Phi_{\wedge\vee}$ or Eq. (2.1), such that

$$\{(\Phi_{\vee} \to \Phi_{\wedge})(x)\} \wedge \{(\Phi_{\vee} \leftarrow \Phi_{\wedge})(x)\} = \Phi_{\wedge\vee}(x)\,\big|S_{Q_x} \leftrightarrow V_{Q_x} \leftrightarrow V_{Q_y} \tag{2.3c}$$

where $\Phi_{\wedge\vee}(x)$ usage in the hypercube model, appears valid in its compression ratio

$$\begin{aligned}\mathcal{C}_r(x) &= \frac{\mathcal{C}(Q_x \cdot Q_{x'})}{Q_x + Q_{x'}} = \frac{Q_y}{\mathcal{C}'(Q_y)} = \Phi_{\wedge\vee} \sum \beta(x) \vdash |xx'| \leftrightarrow |y| \\ &= (16\,\text{bits}) \leftrightarrow (8\,\text{bits}) = (2\,\text{B}) \leftrightarrow (1\,\text{B}) \\ &= 2{:}1 \text{ compression.} \end{aligned} \tag{2.3d}$$

□

Remark 2.1. Fuzzy convergence between maximum and minimum of $x \in [0, 1]$ implies to many-valued logic [21], now in abstraction by $\wedge, \vee$ operators.

---

and

Remark 2.2. In recognition of Eqs. (2.2)-(2.3), the $\mathfrak{R}$ relationships in Eq. (2.1), denote an encoded-decoded, compressed-decompressed data projected into a hypercube, or conversely from its dual space of at least two subcubes on either end of the equation.

2.5 Compression Products Aimed by the FBAR Algorithm

To deliver a DE-transmission, our first approach was to study textual samples with binary constructs for a lossless compression, similar to the approach made by Shannon (1948) [44] on English alphabet. Of course, with a main difference. We used standard characters in ASCII, with their 256 bit-byte combinations (2^8 bits, Lemma 2.1) on both binary and text. The resultant logic could be employed in the order of integer multiplications. For example, the RGB colors satisfy a huge number of possible combinations i.e., a 3-table consisting $256_{\text{R}} \times 256_{\text{G}} \times 256_{\text{B}} \approx 16.7$ million combinations. This integer in turn supports other data types or non-English spoken languages (Unicode tables). In this paper, however, the primary scope for the first version of FBAR is $256 \times 256 = 65536$ combinations. Its future versions, hypothetically grow toward much greater numbers beyond a 3-table, i.e., a 4-table for

a $\mathcal{C}_r$ = 8:1 or 87.5% compression as $65536^{4\,\text{tables}} = 16$ exabytes (EB). The latter is convenient for managing *very large databases* $\geq 1\,\text{TB}$. Such hypotheses are discussed in our future work section, Section 5.3, and translation table in Section 4.

2.6 FBAR Synthesis

Let an algorithm synthesize any logic state known to quantify information. To quantify, we need operators that operate on logic states. Those operators would be:

(1) Boolean logic: Boolean operators as *and*, *or*, and *not* or *negate*. Boole (1848) [9]
(2) Fuzzy logic: Fuzzy operators as *fuzzy-and*, *fuzzy-or*, and *not* or *negate*. Zadeh (1965) [51]
(3) Fuzzy-binary and-or (FBAR) combinatorial logic: synthesize all two above as *fuzzy-binary-and*, *fuzzy-binary-or*, and *not* or *negate*. This gives a lossless
    (i.) dynamic FBAR encoding and compression,
    (ii.) static FBAR encoding and compression,
    (iii.) FBAR decoding and decompression. Alipour and Ali (2010) [2]

In this paper, we focus on 3.ii and 3.iii methods used in the algorithm to achieve "universal predictability." This could only happen if both methods are conducted in terms of FBAR *pairwise logic*: synthesize all three logics via *and-or* and *negate* operators on pairs of 0, near 0, 1, and near 1 states to *minimally transmit* 8 bits, or

(1) * a possible combination of 8 bits or $\{00, 01, 10, 11\}$ denoting 4 possible bit-flag combinations (1-bit operators) on the four bit pairs (1-character input)
    (i.) FBAR bit-flag operators **z** for zero, **n** for negate, **i** as impurity, and **p** as purity operators on each pair of bits (definitions are given in Section 3.2)

(ii.) Employing **znip** operators in code, their minimum dimensional intersection as $\{\mathbf{z}, \mathbf{n}, \mathbf{i}, \mathbf{p}\} \times \{\mathbf{z}, \mathbf{n}, \mathbf{i}, \mathbf{p}\}$ allows such "transmissions" occur.

This in total gives 4 *fuzzy-binary* or *fb* bits (definitions are given in Section 3.2)

(2) * Thus, a minimum of 8 bits is transmitted via 4 bits. Since we need to represent data through standard 8-bit characters, the 4*fb* bits is concatenated with another 4*fb* bits, thus a pair of characters or 16 bits input are transmitted via 8*fb* bits output. This is an FBAR compression product.

2.6.1 Expected Outcomes

From (2)*, DE entropies are emerged denoting 50% and higher fixed compressions, regardless of the number of inputs given to the algorithm. This is theorized before our FBAR technique in Section 3, such as **znip** bivector operators are introduced and benefited from their *product elements* and *conjunctive normal form* (CNF) *conversions* [14, 27], thereby tested and employed within the algorithm in Section 4. Also, by referring to *Hamming distance* and *binary coding* [24, 46], a preamble on *prefix coding* to encode and decode data by operators is formulated. Higher fixed compressions are hypothesized for a negative entropy growth, once the minimum degree of a DE-compression or 50% is proven in theory and in practice. In Section 4, we demonstrate these entropies by proving the minimum degree of our DE-technique relative to its 4D-model of I/O data. We then prove decompression by satisfying the decoding method employed for the compressed data. Decompression is done by referring to byte addresses in the compressed file denoting the original input in the last two DE possibilities, (1)* and (2)*. We evaluate all of the hypotheses in the FBAR technique to demonstrate the validity of our concept. The efficiencies of compression in Theorem 1.1, are further evaluated through complexity measures on the algorithm's code with bitrate performance. This quantity is measured for LDC *temporal* and *spatial* operations during I/O data access and process between compression and decompression states.

3

FBAR Compression Theory

In this section, we formulate the FBAR synthesis in form of theorems and proofs aligned with its foundation from Section 2.3. Then, a 4D compression-decompression model is constructed by using FBAR operators and conversion functions on I/O data, as an improvement to the universal model from Section 2.4. The model should exhibit DE predictable values. It also encloses the DE values in form of bits, from a compressed form, to its decodable or decompressed form, in a lossless manner.

3.1 Reversible FBAR Compression Theorem and Proof

A reversible FBAR compression theorem begins with an assumption:

Assumption 3.1. Suppose for every x character input we have a righthand character x', outputting a sequence $\int = (xx')$. We assume our machine compiler compiles data on 8-bit words. We also assume x and x' are from the ASCII table with a range of 0-255 characters. Let also any sequence of words be quantified by a length function ℓ. Thus, the length of $\int$ in bits is $\ell(\int)$ =16 bits or 2 bytes ASCII.

Using Assumption 3.1 for a logical consequence, we hence submit a bit manipulation theorem on the given sequence $\int$, in terms of

---

Theorem 3.1. Let the machine store a 2-byte binary input xx', as information. Once we manipulate a pure byte sequence $\beta =$`11111111` to obtain xx' by four single bit-flags, one y character is produced denoting the xx' content, equal to 8 bits.

Theorem 3.1 is analogous to the notion of Hamming distance [24, 38]: "the minimum number of substitutions required to change one string to another." However, our case has no relevance to Hamming error-correction characteristics, and concerns *the number of fuzzy pairwise bit substitutions*. From Assumption 3.1, the notion of storing any sequence $\int$ in Theorem 3.1 becomes valid in terms of an array quantifying the contents of $\int$, or by convention

---

Remark 3.1. The machine stores data in form of an array A on sequence $\int$. Either x or x' from $\int$, is of ASCII type measured in bpc, or entropy rate H.

Hence, a bivector product $\mathbf{e}_{ij}$ on subfields by Lounesto (2001) [32], via its dual scalar element h from Definition 2.3, self-contains and quantifies input xx' in terms of

---

Definition 3.1. The y character is stored in one of the rows r in array $A_{r\times 4}$, where r satisfies a possible number of ASCII combinations for xx'. For all y, an $x \times x'$ combination produces $r = 256 \times 256 = 65536$ rows with a subspace scalar of $h^2\mathbf{e}_{ij}^2$.

The r for xx' by Definition 3.1, further shows the following to be true, if and only if, an interactive proof on FBAR logic is presentable (Section 3.1.1). Hence

---

Proposition 3.1. The y in A, holds single bit-flags that occupy the four columns i, j, k and l, as biquaternion products from Proposi-

tion 2.1, in the $r \times 4$ array, or

$$A_{r\times 4} = A_{r\times(i\ j\ k\ l)}\ , \quad \overset{1}{(i}\ \overset{2}{j}\ \overset{3}{k}\ \overset{4}{l)} = h^2 \begin{pmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 & \mathbf{e}_4 \end{pmatrix}^2 = \begin{pmatrix} h^2\mathbf{e}_{12}^2 & h^2\mathbf{e}_{34}^2 \end{pmatrix} \tag{3.1a}$$

where the bit-flags field is displayed by

$$\therefore xx' \longrightarrow y \in \{r\times(i\ j\ k\ l)\} = 65536\times(1_x1_{x'}\ 1_x1_{x'}\ 1_x1_{x'}\ 1_x1_{x'}) \tag{3.1b}$$

and dimensionally measured by length ℓ as

$$\therefore \ell\,(A_{r\times 4}) = 65536 \times (2\,\text{bits}\ 2\,\text{bits}\ 2\,\text{bits}\ 2\,\text{bits}) \tag{3.1c}$$

holding input data xx', by a y in the same field, in terms of

$$\therefore \ell\,(A_{r\times 4}) = 65536\ \times 8\ \text{bits} = 64\ \text{kilobytes} \tag{3.1d}$$

where $y = 8$ bits $\in A_{r\times 4} = 64\,\text{K}$, affirming that A is static. If the bivector $\mathbf{e}_{12}^2 = -1$ for a pre-occupying character x, then its dual scalar is $h^2 = -1$, otherwise $h^2 = 1$ for the post-occupying character x' by $\mathbf{e}_{34}^2 = 1$. Both conditions determine the subspace property on each xx' input as a superposing pair under compression. Thus, the compression products are orthogonally projective, positive and non-commutative.

Proof. Suppose a φ symbol denotes bit-flags for all $\mathbf{e}_{1234}$ in the $r \times 4$ array. According to Assumption 3.1, for the number of ASCII combinations on r, a total of $4\times4\times4\times4 = 256$ on x, and 256 on x', satisfies 65,536 unique flag combinations. Its unit vector $\hat{\boldsymbol{v}}_\varphi$ whose coordinates are in one of the 1×4 array dimensions, has a length of 1 bit with a scalar occupation. Thus, the y character is stored in the xx' intersection $\cap$, where φ values meet. This gives y a different content not equal to φ, but representing the exact location of xx' in the sequence as well as content when φ flags are *translated*. We create a static *translation table* to decode these flags based on where the y character is stored i.e.

the *address* with a *reference point*, or

$$\underbrace{\Big(\overbrace{\big(\,1_x\quad 1_x\quad 1_x\quad 1_x\,\big)\quad\big(\,1_{x'}\quad 1_{x'}\quad 1_{x'}\quad 1_{x'}\,\big)}^{8\text{ bits}}\Big)}_{\substack{\Big(\underbrace{(\quad\overbrace{4}\quad\times\quad\overbrace{4}\quad)}_{\text{1st 16 combinations}}\cap\underbrace{\overbrace{(4\times 4)}^{8\text{ bits}}}_{\text{2nd 16}}\Big)\\ y\text{ address}\cdots}}$$

$$\underbrace{\cap\Big(\underbrace{\overbrace{(4\times 4)}^{8\text{ bits}}}_{\text{3rd 16}}\cap\underbrace{\overbrace{(4\times 4)}^{8\text{ bits}}}_{\text{4th 16}}\Big)}_{\cdots\, y\text{ address}} = 8\text{ bits} \tag{3.2a}$$

stored as character y in one of the 65,536 rows (*prefix addresses*) representing one of the four-dimensional combinations. These combinations are either 1st, 2nd, 3rd or 4th 16 combinations of bit-flags, for xx'. Equation (3.2a) validates Proposition 3.1 equations for specific address and flags configuration. Specifically,

$$\forall\varphi\in A_{i,j,k,l}|\,\hat{\boldsymbol{v}}_{\varphi}=(1,0,0,0)\vee(0,1,0,0)\vee(0,0,1,0)\vee(0,0,0,1)\ , \tag{3.2b}$$

such that

$$\forall y\in A_{1\times 4}\subset A_{r\times 4}\mid\varphi\cdot A_y=\lambda_{xx'}+\varrho_{xx'}=(i,j,k,l)_{xx'}+(i,j,k,l)_{xx'} \tag{3.2c}$$

So, the translation table is a precalculated (prefix) rows-by-columns file on bit-flags, giving a reference point for the stored character y. The reference is a specific bit-flag combination from the 65,536 possible rows, constituting the y address. The bit-flags set φ, represents all 16 bits content of xx', by manipulating a pure binary sequence $\beta=$ `1111 1111` recursively. This is shown in Eqs. (3.5). The byte is manipulated to obtain the binary content of xx'. From Eq. (3.2c), let this manipulation start with the left-most bit to the right-most bit, operating on the left-byte λ and the right-byte ϱ of sequence $\int$. This gives a y product on

the xx' input, and is expressed by the following compression function:

$$\begin{aligned} &g \circ f : \int_{\text{in}} \to A \\ &xx' \mapsto g(f(xx')) = f(y, \varphi)\,, \\ &f(y, \varphi) = y \times \varphi_{xx'} = y(\varphi_x + \varphi_{x'}) = y_{\varphi_x} + y_{\varphi_{x'}} = y_{\varphi_{xx'}} \end{aligned} \tag{3.3}$$

Let f be a function composition that maps the contents of $\int$ to the contents of array A, holding the same xx' contents via bit-flags φ. The bit-flags are occupied in form of character y. A two-variable function $f(y, \varphi)$ expresses the y character carrying flags in array A, or its respective field $\mathbf{F}_y$. Its length is specified by

$$\begin{aligned} \ell(y_\varphi) = \varphi A_y =& \Big[\begin{pmatrix} 1_i & 1_j & 1_k & 1_l \end{pmatrix}_x \cdot \begin{pmatrix} 1_i & 1_j & 1_k & 1_l \end{pmatrix}_x \Big] \\ &+ \Big[\begin{pmatrix} 1_i & 1_j & 1_k & 1_l \end{pmatrix}_{x'} \cdot \begin{pmatrix} 1_i & 1_j & 1_k & 1_l \end{pmatrix}_{x'} \Big] \\ &= \mathfrak{V}(x) + \mathfrak{V}(x') = \hat{\boldsymbol{\mu}}_x \|\mathbf{e}\|^2 \left(2\,\hat{\boldsymbol{\mu}}_{\beta_x} + 2\,\hat{\boldsymbol{\mu}}_{\beta_{x'}}\right) \\ &= 4\,\mathfrak{V} + 4\,\mathfrak{V} = 8\,\text{bits} \end{aligned} \tag{3.4}$$

In Eq. (3.4), the dual $\mathfrak{V}$ manipulation method is of bivector type $|\mathbf{e}_{ij}|$ or $|\mathbf{e}_{kl}|$ from the norm $\|\mathbf{e}\|^2$ in Eqs. (2.2b), and so its product $\mathbf{e} \cdot \mathbf{e}$, is of a data-decoding process. The method is derived to alter the partitioned bit-pair values from β in $\mathbf{F}_y$ to obtain original data, as if its values are of *Pythagorean identity* [30] such that $8\,\hat{\boldsymbol{\mu}}_\beta = \sum_{n=1}^{8} \int (\sin^2\theta + \cos^2\theta)_n =$ 11111111 stored as a byte, or

$$\begin{aligned} &\{4\,\mathfrak{V}\,\text{manipulations on } y = 8\,\hat{\boldsymbol{\mu}}_\beta = 11111111 = \beta \text{ to obtain } x + \\ &\quad 4\,\mathfrak{V}\,\text{manipulations on } y = 8\,\hat{\boldsymbol{\mu}}_\beta = 11111111 = \beta \text{ to obtain } x'\} = 8\,\mathfrak{V}\,, \end{aligned} \tag{3.5a}$$

$$\begin{aligned} \therefore \varphi &\left(\underbrace{\overbrace{(11)}^{\frac{1}{4}y=y_i}}_{1\text{bit}_i \times 1\text{bit}_i} \underbrace{\overbrace{(11)}^{\frac{1}{4}y=y_j}}_{1\text{bit}_j \times 1\text{bit}_j} \underbrace{\overbrace{(11)}^{\frac{1}{4}y=y_k}}_{1\text{bit}_k \times 1\text{bit}_k} \underbrace{\overbrace{(11)}^{\frac{1}{4}y=y_l}}_{1\text{bit}_l \times 1\text{bit}_l} \right)_{xx'} = y_{xx'} \\ &= 8 \text{ bits} \to \underbrace{xx'}_{16\text{bits}} \end{aligned} \tag{3.5b}$$

Thus, to store more y characters in the field of rows, $r \times 4$, we establish a finite field $\mathbf{F}_y$ with n elements. Therefore, $\mathbf{F}_y = \{y_1, y_2, \ldots, y_n\}$,

represents a compression of sumset $\sum \int = (x_1 x'_1 + x_2 x'_2 + \ldots + x_m x'_m)$, achieving

$$\ell\left(\mathbf{F}_y\right) = \ell\left(\sum_{i=1}^{m} x_i x'_i\right)_{\text{in}} \overset{\sum_{r=1}^{65536} A_{r\times 4}}{\longrightarrow} \ell\left(\sum_{i=1}^{n} y_i\right)_{\text{out}} = \frac{1}{2}\sum \int \tag{3.6}$$

Equations (3.5) and (3.6), show a fixed degree of double-efficiency over y, generating a decodable 50% compression. □

Proposition 3.2. The addressability of any original data is self-embedded in a grid file with 65,536 addresses, Eqs. (3.1). For each double-character xx' input, one specific address is occupied by a character y output, Eqs. (3.5). Translating the occupied address via a table whose row content returns original content, is by translating bit-flag combinations on a pure byte $\beta =$ `11111111` obtaining xx' from Eqs. (3.4)-(3.5).

3.1.1 Interactive Proof

Proof. Suppose by default, we establish an ASCII combination on an xx' input. The total number of combinations is 256 ASCII characters for x, and 256 ASCII characters for x'. Thus, $f(y) = 256\,x \times 256\,x' = 65536\,xx'$. This gives an intersection of the combinations in total 65,536 8-bit addresses (64 KB). We prove the intersections using logic and subspace topology on array A. The combinations are integrable in space where bits reside as stored and then manipulated. Let a compact Hausdorff space [42] contain a pure byte sequence $\beta =$ `11111111` for manipulation to obtain xx'. The manipulation as a filter is done at a target point where space is locally compact. This results in compacted bits by associating all possible fuzzy pairwise bit manipulations, using bitwise operators OR `|`, and AND `&` in code. Therefore, the manipulation '`11111111`' for a 2-byte content xx' is '`11111111 + 11111111`'. The association of manipulation is via bit-flags giving left-byte intersected with right-byte. This association of two 8-bit sets gives an 8-bit output.

Interactively, $\forall\ xx' \mapsto \{y\}_{i,j,k,l\in\mathbb{R}^4}$ we prove:

$$xx' \overset{\text{store}}{\longrightarrow} A_y = \frac{xx'}{\|\mathbf{e}\|} = \frac{xx'}{\sqrt{\mathbf{e}\cdot\mathbf{e}}} = \frac{xx'}{\sqrt{4}} = \left(\frac{1}{\sqrt{2}}x_i\ \frac{1}{\sqrt{2}}x_j\ \frac{1}{\sqrt{2}}x'_k\ \frac{1}{\sqrt{2}}x'_l\right) \tag{3.7a}$$

$$\therefore xx' \overset{\text{store}}{\longrightarrow} \boxed{\mathbf{P}} \overset{\text{filter}}{\longrightarrow} f(y,\varphi) = \sqrt{|e_{12}|^2 + |e_{34}|^2} = \sqrt{\frac{1}{2}x^2 + \frac{1}{2}x'^2} = 8\,\text{bits} \tag{3.7b}$$

Using the law of associativity in logic, the manipulative bits for xx' appear as

$$\begin{aligned}\therefore &\left(\{8\,\text{bits}_x\} \in \underbrace{i \times j}_{2\,\text{dimensions}} \ni \{8\,\text{bits}_x\}\right) \cap \\ &\left(\{8\,\text{bits}_{x'}\} \in \underbrace{k \times l}_{2\,\text{dimensions}} \ni \{8\,\text{bits}_{x'}\}\right) = \mathfrak{V}(\beta) \\ &= 8\text{ dual }\mho\text{ manipulations} = 8\,\text{bits} = y \times \varphi_{xx'} \in \mathbb{R}^4\end{aligned} \tag{3.7c}$$

and the address of y for xx' is found via 1-bit flags φ operating on β, such that

$$\begin{aligned}\therefore \ell(\varphi_{xx'}) &= \left(\{\underbrace{4_{\varphi\forall x}}_{4\,\text{flags}}\} \in i \times j \ni \{\underbrace{4_{\varphi\forall x}}_{4\,\text{flags}}\}\right) \cap \left(\{\underbrace{4_{\varphi\forall x'}}_{4\,\text{flags}}\} \in k \times l \ni \{\underbrace{4_{\varphi\forall x'}}_{4\,\text{flags}}\}\right) \\ &= 4\,\text{bits}\end{aligned} \tag{3.7d}$$

where a storage field $\mathbf{F}$ covering all bit-flags φ is quantified in terms of

$$\ell\left(\mathbf{F}_\varphi\right) = \ell\left(\prod_{e=1}^{4} 4\begin{pmatrix} 1_i & 1_j & 1_k & 1_l \end{pmatrix}_e\right) = \prod_{e=1}^{4}\left(4_i + 4_j + 4_k + 4_l\right)_e$$

$$= 16_1 \times 16_2 \times 16_3 \times 16_4 = 16^4\text{bits} = 64\,\text{kilobytes} \tag{3.7e}$$

So, $i \times j \times k \times l$ represents a binary address as a four-dimensional flag or a byte in a spatial field $\mathbb{R}^4$ topology. Therefore, xx' is distributed in 4 dimensions i, j, k and l by storing 1 y character in the corresponding

row. Now for the "storage field," suppose we create a *grid file* $\boxed{\mathbf{G}}$ as a portable file with an empty space of 65,536 $i \times j \times k \times l$ rows, satisfying all possible combinations. This grid in specification should cover the y field or $\mathbf{F}_y$ as well as bit-flag combinations. The y field has a limit to store 1 to n, of y characters corresponding to more inputs of $\int$. Let this be a sumset $\sum \int$ from Eqs. (3.5). Thus, we specify these possible address combinations with a multi-sum on the available dimensions of 1-bit flags φ, covering field $\mathbf{F}_y$, or

$$A_y \cap A_\varphi = \mathbf{F}_y \times \mathbf{F}_\varphi = \mathbf{F}_y \otimes \sum_{1 \leqslant i,j,k,l \leqslant 16} \varphi_{xx'} = \boxed{\mathbf{G}} \tag{3.8}$$

This, specifically builds our grid file multiplied with the sequences input $\int$ transposed matrix, as follows

$$\begin{aligned} \boxed{\mathbf{G}} &= \{y_1, y_2, \ldots, y_m\} \\ &\times \begin{bmatrix} 1 & 2 & 3 & \cdots & 16 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots & 16 \\ 1 & 1 & 1 & 1 & 1 & 2 & 3 & \cdots & 16 & 1 & 1 & 1 & \cdots & 16 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 & \cdots & \cdots & 16 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \cdots & 16 \end{bmatrix}^{\mathrm{T}}_{\sum \int} \\ &= \{8\,\text{bits}, 8\,\text{bits}, \ldots, 8\,\text{bits}\} \times 65536_{ijkl} = \int_{\text{out}} \end{aligned} \tag{3.9a}$$

Evidently, the way y_i is stored and configured, is later decodable for an LDD, where

$$\left\{ \int_{\text{out}} \in \boxed{\mathbf{G}} \,\middle|\, 1\, y \leq s \leq \ell\left(\mathbf{F}_y\right) \right\} \;,\; \ell\left(\boxed{\mathbf{G}}\right) = \; \ell\left(\mathbf{F}_y\right) + 64\,\text{kilobytes} \tag{3.9b}$$

Ergo, by default, we occupy a $y = 8$ bits for an $xx' = 16\,\text{bits}$, since Eqs. (3.9) intersected 8-bit flags for x with 8-bit flags for x' in the range of available rows in the grid file. So, if we exemplify a string of characters "`resolved`," the sequence $\int$ becomes $\sum \int = x_1 x_1' + x_2 x_2' + x_3 x_3' + x_4 x_4' =$ `resolved`. Thus, the allocated bit-flags with respect to byte addresses display,

$$\begin{aligned} \sum \int = 64\,\text{bits} \overset{\text{store as}}{\longrightarrow} \mathbf{F}_y &= \{y_1, y_2, y_3, y_4\} \subset A_{65536 \times 4} \\ &= 32\,\text{bits} \in \|r\| = \frac{[1,4]}{65536}\ \text{rows}\ , \end{aligned}$$

$$\text{where } |\mathbf{F}_y| = 4 \ |y_i \in [1 \times 1 \times 1 \times 1 \ , \ 16 \times 16 \times 16 \times 16] \tag{3.9c}$$

In this case, the rows magnitude $\|r\|$ is up to $\sqrt{1^2 + 4^2} = \sqrt{17} = 4.12$ out of 65,536 rows, since we have a cardinality of $4y$'s occupying the array or storage field $\mathbf{F}_y$, with a specific address to decode 64 original bits. This gives a decodable 50% compression plus a static size of 64 kilobytes. □

3.1.2 FBAR Compression-Decompression Theorems

Following the Compression Proof 3.1.1, we specify an FBAR logic on I/O bit manipulations, delivering an FBAR compression theorem as follows:

Theorem 3.2. Let the machine store a 2-byte binary input, xx', as information. Once we manipulate a pure byte sequence '`11111111`' to obtain xx' by bitwise and-or, negate [9], and close it by fuzzy transitive closure [26], one y character is produced denoting the xx' content, equal to 8 bits.

From Theorem 3.2, following Theorem 3.1 proof, we deduce an FBAR compression $\mathcal{C}$ corollary,

---

Corollary 3.1. The four combined operations, pairwise and-or, negate and fuzzy transitive closure on sequence $\int$, give a 50% fixed compression.

From Theorem 3.2, we further deduce a complete FBAR compression $\mathcal{C}$ corollary

---

Corollary 3.2. The FBAR four combinatorial operations on any sequence $\int$ from $\sum \int = (x_1 x_1' + x_2 x_2' + \ldots + x_m x_m')$ give the same compression ratio 2:1, or 50%.

Complementing Theorem 3.2 with a decompression $\mathcal{C}'$ theorem, we deduce a complete FBAR compression-decompression theorem or $\mathcal{C}\mathcal{C}'$

Theorem 3.3. Let a 1-byte y output holding bit-flags of φ, represent a 2-byte binary input. This gives a 50% fixed compression. During compression, byte addresses are stored in a static address as $(4{\times}4) \times (4{\times}4)$ single bit-flags, once accessed and decoded via a translation table on y and φ, a lossless decompression is obtained.

and obtainable by

Proposition 3.3. Sequence $\int$ is compressed into y whilst reconstructable via bit-flags held by y, where $\ell(y) = \ell\left(\frac{xx'}{2}\right) = \frac{\int}{2}$ or 50% fixed $\mathcal{C}$ on xx' inputs. Once inputs are equally compressed via bit-flags in different locations of y, any $\int_{\text{in}}$ is losslessly reconstructed. The specific address is where y is stored amongst the 65,536 rows.

3.2 4D Bit-Flag Model Construction

3.2.1 I/O Operators Construction

To satisfy the practical application of our proof for Theorem 3.2, we need to construct the algorithm components with relevant operators to conduct an LDC-DD.

Proof. Let a grid file component $\boxed{\mathbf{G}}$ be constructed according to Eqs. (3.9), with a translation table $\boxed{\mathbf{TT}}$. To construct these two components, we intersect the dimensions of bit-flags φ with each other in the 1×4 matrix, for each LDC I/O. We partition φ according to Eqs. (3.4), (3.7c) and (3.7e) in i, j, k, l dimensions of $\boxed{\mathbf{G}}$, resulting *four bivector operators*. Let those operators be **z**, **n**, **i**, **p**, where their paired combinational form, **zn** and **ip**, constructs in total the four dimensions. The static range [**zzzz** , **pppp**] for $[1, 65536]$ rows, stores only one dynamic character output y, per two-character input xx':

$$\begin{aligned} &\exists\, xx' = 16\,\text{bits} \in i \times j \times k \times l \ni y = 8\,\text{bits}\,, \\ &\text{iff } h^2\mathbf{e}^2_{1234} = \mathbf{znip} \overset{\text{operates on}}{\longrightarrow} xx' \text{ for compression}\,. \end{aligned} \tag{3.10a}$$

Notation $\ni$, here, follows the set membership notation $\in$, symmetrically, such that y is identifiable in the i, j, k, l bivector dimensions.

Now, we adapt the grid file range on the rows from Eqs. (3.9) to bit-flag operators, giving

$$\begin{aligned}&\mathbf{znip} \overset{\text{operates on}}{\longrightarrow} xx' = y \text{ , where}\\ &\mathbf{znip} \in i \times j \times k \times l = [\mathbf{zzzz} \ , \ \mathbf{pppp}]\\ &= [1 \times 1 \times 1 \times 1 \ , \ 16 \times 16 \times 16 \times 16] = [1 \ , \ 65536] \end{aligned} \tag{3.10b}$$

where **z** is a zero or neutral operator, **n** a negate operator, **i** an impurity operator, **p** a purity operator, operating on bits. Suppose by definition, we apply *and-or logic* ($\wedge\vee$) to the **z** operator on bit pairs, thus

---

Definition 3.2. For all **z**, **z** on a pair of bits in the x or x' binary, returns the same bits when and-or applied, where this applies to all remaining pairwise combinations, or

$$\mathbf{z}(0\wedge\vee 0) = 00,\ \mathbf{z}(1\wedge\vee 1) = 11,\ \mathbf{z}(0\wedge\vee 1) = 01,\ \mathbf{z}(1\wedge\vee 0) = 10. \tag{3.11a}$$

Then negation holds good for all pairwise bit combinations

Definition 3.3. For all **n**, **n** on a pair of bits in the x or x' binary, negates the bits when and-or applied, where this applies to all remaining pairwise combinations

$$\begin{aligned}&\mathbf{n}(\neg(0 \wedge \vee 0)) = 11,\ \mathbf{n}(\neg(1 \wedge \vee 1)) = 00,\ \mathbf{n}(\neg(0 \wedge \vee 1)) = 10,\\ &\mathbf{n}(\neg(1 \wedge \vee 0)) = 01. \end{aligned} \tag{3.11b}$$

From the laws of Boolean algebra covering $\wedge$, $\vee$ and $\neg$ operators, the defined **z** and **n** operators in consequence hold good as axioms:

Axiom 3.1. Operator **n** is antecedent to negate all pairwise bit combinations. The consequent output is always the opposite of the given input.

---

and

Axiom 3.2. Operator **z** is antecedent to pass all pairwise bit combinations. The consequent output is always as same as the given input.

We can now make the definitions of Eqs. (3.11) and (3.12) quite explicit as follows:

Definition 3.4. For all **z**, **z** on any pair of bits in the binary of x or x', returns the same bits, where this applies to all remaining pairwise combinations.

and

Definition 3.5. For all **n**, **n** on any pair of bits in the binary of x or x', returns negated bits, where this applies to all remaining pairwise combinations.

Now, suppose by definition, we apply *transitive closure* $\overset{\curvearrowright}{\longrightarrow}$, to the **i** operator on bit pairs, acting as "operates on" from Eqs. (3.10), such that

Definition 3.6. For all **i**, bit pairs in the binary of x or x' is either 01 or 10. **i** closes with 1 for 01, and 0 for 10, or

$$\mathbf{i}(01) = 0 \overset{\curvearrowright}{\longrightarrow} 1 = 1, \quad \mathbf{i}(10) = 1 \overset{\curvearrowright}{\longrightarrow} 0 = 0. \tag{3.12a}$$

and if applied to the **p** operator, we then have

Definition 3.7. For all **p**, bit pairs in the binary of x or x' is either 00 or 11. **p** closes with 1 for 11, and 0 for 00, or

$$\mathbf{p}(11) = 1 \overset{\curvearrowright}{\longrightarrow} 1 = 1, \quad \mathbf{p}(00) = 0 \overset{\curvearrowright}{\longrightarrow} 0 = 0. \tag{3.12b}$$

The bit-pair operators from Eqs. (3.12a) and (3.12b), generate conflicting binary products in terms of

---

Paradox 3.1. Operator **i** is coincident with operator **p** in bit-pair products that end with 1 or 0. The consequent output after closure is $\mathbf{i}(10) = \mathbf{p}(00)$, $\mathbf{i}(01) = \mathbf{p}(11)$.

---

which is in code, addressed by

Solution 3.1. We first consider a pure sequence of bits, '`11111111`'. We manipulate the bit-pairs in the sequence with **ip**, then its result by **zn** combinations for original data xx'. Example (3.1) shall prove this.

Combining the **z** and **n** axioms with the **i** and **p** solution, further delivers

---

Definition 3.8. Combining **z**, **n**, **i**, and **p**, gives a combination of FBAR operators locatable in 4 dimensions making a possible 65,536 **znip** combinations. Each occupying y in one of the 65,536 rows of the 4 dimensions, represents a **znip** combination corresponding to an xx' input. These combinations are:

ip as an impure or pure pairwise bits' dimension providing 16 combinations:

iiii iiip iipi ipii piii iipp ippi ppii pipi ipip piip ippp pipp ppip pppi pppp (3.13a)

zn as a zero or negate pairwise bits' dimension providing 16 combinations:

zzzz zzzn zznz znzz nzzz zznn znnz nnzz nznz znzn nzzn znnn nnnz nznn nnzn nnnn (3.13b)

Now, we establish a static solution using Eq. (3.8) and the combinations above for an LDC operation

Solution 3.2. We build a grid file $\boxed{\mathbf{G}}$ based on these available combinations. The combinations constitute the finite field addresses $\mathbf{F}_y \times \mathbf{F}_\varphi$ as our static solution. This field gives a static number of rows and addresses for each compressed character y_i.

and the dynamic solution continuing the previous LDC operation is,

Solution 3.3. The $\boxed{\mathbf{G}}$ grows in file size as a dynamic field of compressed characters y_i in length, when more than 1 character is stored beyond its static range. So, we store the 1 y character by the program in one of the 65,536 rows representing original data (two characters) within the intersected columns of the dimensions.

and its dynamic solution for an LDD operation, respectively, would be

Solution 3.4. We access $\boxed{\mathbf{G}}$ by program code, invoking a comparator subroutine in our code. A $\mathcal{C}'$ is achieved by traversing the $i \times j \times k \times l$ dimensions as the grid's Hamming distance d for each data read between fields $\mathbf{F}_y$ and $\mathbf{F}_{xx'}$ via $\mathbf{F}_\varphi$.

So the question remains that: where should we establish distance d between the prefix encoding and decoding levels of our $\mathcal{C}$ and $\mathcal{C}'$ operations?

To answer this question, we need to succeed in Solution 3.4. For this, we recall Eqs. (3.7)-(3.9), and consider the Hamming distance d definition by Symonds (2007) [48], thus measuring our d as follows:

Definition 3.9. Hamming distance d is measured between n-compressed characters stored with a minimum dynamic space of 64 K = 65,536 static rows of $\boxed{\mathbf{G}}$, or as of Eq. (3.9b), in $\mathbf{F}_y +$ 64 K, and their decodable $2n$-input characters in a maximum static space as translation rows and columns in terms of $\{\mathbf{F}_{xx'}, \mathbf{F}_r, \mathbf{F}_\varphi, \mathbf{F}_y\} = \boxed{\mathbf{TT}}$. Therefore, the compressed characters y_i are decoded by flag vectors φ as carried in their address of the finite field $\mathbf{F}_y \times \mathbf{F}_\varphi = \boxed{\mathbf{G}}$.

Thus

Definition 3.10. The number of coefficients in which they may differ in $\mathbf{F}_y$ to return original characters, is equal to the number of bit-pair manipulations $\mathfrak{V}(\varphi A_y)$ elicited from Eqs. (3.5) using **znip** operators based on Eqs. (3.10)-(3.12).

The last two definitions deduce the following

Definition 3.11. Distance d conserves the finiteness of character I/Os via the code comparator, comparing characters between $\{\mathbf{F}_{xx'}, \mathbf{F}_y\} \subset$ $\boxed{\mathbf{TT}}$ and $\{\mathbf{F}_y\} \subset$ $\boxed{\mathbf{G}}$, and will not recover any randomness between two or more compressed characters located in 1 or > 1 rows of $\mathbf{F}_y$ in $\boxed{\mathbf{G}}$.

In addition,

Definition 3.12. d is 0 if at least 2 compressed y's are stored in the same row address of $\mathbf{F}_y$ in $\boxed{\mathbf{G}}$, which represent 4 original redundant x's decoded from $\boxed{\mathbf{TT}}$,

---

and

Definition 3.13. Distance d is > 0 if compressed characters are located in 2 up to $k = 65536$ grid rows, representing some original characters redundant, otherwise, all as different in $\boxed{\mathbf{TT}}$.

Upon Definitions 3.9-3.13, the following solution is emerged to satisfy a lossless $\mathcal{C}$-and-$\mathcal{C}'$ scenario of at least two compressed y's stored in the $\boxed{\mathbf{G}}$ file component:

Solution 3.5. According to Eqs. (3.8) and (3.9), and by Definitions 3.2-3.8, to decompress data losslessly, the 4 bit-flag operators **znip**, operate on 2 y's in the finite field $\mathbf{F}_y$ in $\boxed{\mathbf{G}}$, manipulating the byte in form of bit-pairs, which return 2 xx''s as their original. The original 4-character string is located in a $\boxed{\mathbf{TT}}$ file with all distances

prefixed for each pair of xx'. This transforms the grid to a distance of 0 when original characters are returned at the $\mathcal{C}'$ phase for each read row r. The code comparator compares the unique **znip** row-by-column address from the table with the stored character in one of the grid file rows, from end-of-file to the file's header.

Hence, benefiting from the Hamming distance propositions followed by their proofs in Symonds (2007) [48], and Eqs. (3.4)-(3.6), the usage of **znip** operators gives a maximum distance d between vectors $\varphi_{\{y_1,y_2\}}$ and $\varphi_{xx'_1xx'_2}$, as $d(\varphi_{\{y_1,y_2\}}, \varphi_{xx'_1xx'_2}) = 16$ for the number of required bit-pair manipulations (*prefix dual-coding* $\mho$). Therefore, as a bonus result, we immediately deduce the following two corollaries:

Corollary 3.3. For two compressed characters y_1 and y_2, giving d with respect to time t, as Hamming rate $R_{\mathbb{H}} = d/t$ to search for a string match, results in a distance $d(\{y_1, y_2\}, xx'_1xx'_2) = 0$ during decompression $\mathcal{C}'$. Let this distance be d'.

Corollary 3.3 for the compressed characters y_1 and y_2, further gives

---

Corollary 3.4. Part 1: The $\varphi_{y_1y_2}$ and $\varphi_{xx'_1xx'_2}$ vectors in output do not differ in the number of coefficients when time t allows $\boldsymbol{\mho}(\{y_1, y_2\}, xx'_1xx'_2) = 16 \xrightarrow{\varphi} 0$ grid transformations, such that from Eqs. (3.8)-(3.9), we firstly establish the integral on distance d at time t

$$\forall R_{\mathbb{H}} \in \mathcal{C} | R_{\mathbb{H}} = \int_{\Delta t} \frac{d}{t}\,\mathrm{d}\,t = \frac{\|r\|}{\mathbf{t}} \int_{\min \|r\|_{\mathbf{G}}}^{d(\mathbf{G},\mathbf{TT})} d(\{y_1, y_2\}, xx'_1xx'_2)\,\mathrm{d}\,d = \frac{\Delta d}{\Delta t}$$

$$\text{where } \exists\ \{y_1, y_2\} \in \mathbf{F}_y \,|\, \ell\,(\mathbf{F}_y) = 2\mathrm{B} + 64\mathrm{K} = \ell\left(\boxed{\mathbf{G}}\right)\ ,\ \mathbf{t} = 1\mathrm{s}\ ,\ \text{and}$$

$$\exists\ \{xx'_1, xx'_2\} \in \mathbf{F}_{xx'} \,|\, \ell\,(\mathbf{F}_{xx'}) = 4\mathrm{B} + 64\mathrm{K} = \ell\left(\mathbf{F} \in \boxed{\mathbf{TT}}\right)\ . \quad (3.14a)$$

Given the occupied $\varphi_{\{y_1,y_2\}}$ and $\varphi_{xx'_1xx'_2}$ values in components set $\{\mathbf{G}, \mathbf{TT}\}$ from a maximum number of rows r, between 4 original characters in $\boxed{\mathbf{TT}}$ and 2 compressed y's in $\boxed{\mathbf{G}}$, as distance d at time t, in virtue of Eq. (3.4) and Lemma 2.1, we deduce

$$\therefore R_{\mathbb{H}} = \int_{\Delta t} \frac{d}{t}\,\mathrm{d}\,t = \iint_{\frac{y_1}{2^{(8+8)}}}^{\max d} \dot{d}\,\mathrm{d}\,R = \frac{\|r\|}{\mathbf{t}} \int_{\frac{1}{k}\approx 0}^{r' \times \boldsymbol{\mho}(\varphi A_y)} \mathrm{d}\,\boldsymbol{\mho}\ ,$$

$$\text{where } \begin{cases} r' \in [1,k]_{\mathbf{TT}} \\ t \in [0,\mathbf{t}] \end{cases} \tag{3.14b}$$

Thus, the maximum value of r' rows holding the translation of the 2 compressed characters to their 4 original characters, is 2 rows in $\boxed{\mathbf{TT}}$, giving a Hamming rate

$$\therefore R_{\mathbb{H}} = \frac{2.23_{\mathbf{G}}}{k_{\mathbf{TT}}} \int_{\approx 0}^{2_{\mathbf{TT}}(4+4)} \frac{r \times \mathfrak{V}}{t} \,\mathrm{d}\, t = \frac{\Delta d}{\Delta t} \geq 0.0039 \text{ Bps} \,, \tag{3.14c}$$

whereby Definition 3.13, constant k is 65,536 grid rows. Rate $R_{\mathbb{H}}$ measured in Bps, is equal to the change of number of $\mathfrak{V}$ bit-pair manipulations on the compressed data in an array of r rows, relative to their original characters' rows r', at time t.

Part 2: After applying **znip** operators in the Eq. (3.14c) upper integral limit, denoting a future $\mathcal{C}'$ as stored addresses by y, we then compute its future rate returning xx'

$$\begin{aligned} \exists \mathcal{C}' \ni R'_{\mathbb{H}} = \frac{\|r'\|}{\mathbf{t}} \int_{\max r'_{\mathbf{TT}}}^{\|r\|_{\mathbf{G}}} \mathrm{d}\,\mathfrak{V} &= \frac{\sqrt{\sum_{i=1}^{k} i^2}_{\mathbf{G}}}{\sqrt{k^2}_{\mathbf{TT}}} \int_{2}^{\frac{2.23}{k}} \frac{\mathfrak{V} \times r'}{t} \,\mathrm{d}\, t \\ &= \frac{\Delta d}{\Delta t} \geq 295.6 \,\mathrm{Bps} \end{aligned} \tag{3.14d}$$

Thus, the total Hamming rate performing $\mathcal{C}$ and $\mathcal{C}'$ is $\sum R_{\mathbb{H}} = R'_{\mathbb{H}} + R_{\mathbb{H}} \in \mathcal{C}\mathcal{C}'$. To evaluate Eq. (3.14d), we measure the total distance d between $\mathcal{C}$ and $\mathcal{C}'$ points via **znip** as their $\mathfrak{V}$ string-match relation. Employing the Pythagoras' theorem gives an imaginary part $\imath = \sqrt{-1}$ for $\mathcal{C}'$, added to its conversed real part from $\mathcal{C}$ as follows

$$\begin{aligned} \sum d(\mathcal{C},\mathcal{C}') = \pm 2\pi \int_{\sqrt{\mathcal{C}}}^{\sqrt{\mathcal{C}'}} \sqrt{\mathcal{C}} \;\mathrm{d}\,\mathcal{C}' &= 2\pi\sqrt{\mathcal{C}^2 - \mathcal{C}\mathcal{C}'} = |8.88|\,\mho \,; \\ \vdash \forall \mathfrak{V} \in \mathbb{C}\ell_4 \left(\mathbb{R}^{2^{\|\mathbf{e}\|}}\right) \Big| \|\mathbf{e}\| \sum d(\mathcal{C},\mathcal{C}') \\ = \left(\mathcal{C}\left(\sum \int_{\mathrm{in}}\right)\right) \mathfrak{V} \left(\mathcal{C}'\left(\sum \int_{\mathrm{out}}\right)\right) &= \|dd'\| = \sqrt{4}\mathcal{C}(d) \\ - \sqrt{4}\mathcal{C}'(d) = 2\mathcal{C}(d) \,+ 2\mathcal{C}(d') &= 8.88\|\mathbf{e}\| \approx 17.7\,\mho \,, \end{aligned}$$

therefore

$$\begin{aligned}\|dd'\| \vdash & \left(\mathcal{C}\left(x_1x_1'x_2x_2'\right) \xrightarrow{d} \{y_1, y_2\}\right) \mathbf{znip} \left(\mathcal{C}'\left(\{y_1, y_2\}\right) \xrightarrow{d'} \{xx_1', xx_2'\}\right) \\ & = \mathfrak{V}\ \mathbf{znip}\ r' = \mathfrak{V}(\{y_1, y_2\}, xx'_1xx'_2) \overset{+}{\bigcap} \mathfrak{V}(xx'_1, xx'_2)\ ,\ \text{ideally} \\ & = \mathfrak{V}\left(xx_1' + xx_2'\right) = \mathfrak{V}(x_1x_1'x_2x_2') = \mathfrak{V}\left(\sum \int_{\text{out}}\right) = 16 + \imath\imath_{1234}^{\|\mathbf{e}\|} \\ & = 17\,\mho\ ,\end{aligned}$$

and for the latterly-deduced result, we finally complement

$$\therefore \forall d\, \exists d' \in \Delta d \,|\, \{\Delta d = (17.7 - 17) < 1\,\mho\} \Leftrightarrow \{d\,\mathfrak{V}\,d' = 16 \xrightarrow{\varphi} 0\,\mho\}\ . \tag{3.14e}$$

Operator $\overset{+}{\bigcap}$ denotes a combinatorial string catenation and intersection of y_i and xx_i' elements, emitted into a (discrete) concrete sequence $\int$. Equations (3.14e), show that there is no **znip** manipulation $\mathfrak{V}$ to be made on $r' \in \boxed{\mathbf{TT}}$ to obtain xx' during $\mathcal{C}'$, except a $d\,\mathfrak{V}\,d'$ string-search, match and catenate relationship, thus giving $d' = 0\,\mho$.

Corollaries 3.3 and 3.4 are possible, if and only if, component $\boxed{\mathbf{TT}}$ is accessed, thereby char addresses compared by code and their data decoded. Such φ vectors agree in all coordinates of the grid's $i \times j \times k \times l$ dimensions standing for an address during FBAR I/O operations. To keep matters simple, here is an example on a single compressed character y, spatially returning 2-original characters xx'

Example 3.1. If xx' = `01000000 00100100` = @$, then by default `11111111` for y is decoded, only when y occupies $i \times j \times k \times l$ dimensions with a unique combination of operators. This combination is $\mathbf{ippp} \times \mathbf{niin}(\texttt{11111111}) = \texttt{01111111 00010100}$ for $i \times j$, and this output intersected with the combination $\mathbf{znnn} \times \mathbf{znzz}(\texttt{01111111 00010100}) = \texttt{01000000 00100100}$ for $k \times l$. The y is stored in one of the rows out of 65,536 possible **z**, **n**, **i**, **p** combinations, which now returns characters @$ = xx' by code.

Therefore, we reconstruct the pair xx' as our output. The output content is now equal to its original. □

Corollary 3.4 assumes continuity on the whole interval quantified as a spatial-temporal type. It further proves Corollary 3.3 via Eqs. (3.14d)-(3.14e), with an output sequence $\int_{\text{out}}$ from Eqs. (3.6) and (3.9). The spatial intervals in Eqs. (3.14a)-(3.14c) are delimited by the rows magnitude $\|r\|$ employed from Eqs. (3.9) in $\boxed{\mathbf{G}}$, and its translation type $\|r'\|$ in $\boxed{\mathbf{TT}}$, where r builds the maximum distance d as the upper limit of integration. The temporal interval is given by t, and as conditioned in Eq. (3.14a), goes with a maximum ideal time $\mathbf{t} = 1\,\text{s}$, i.e., processor(s) and memory being able to handle an occupied space $\geq 64\,\text{K}$ I/O cases. The *fluxion* $\dot{d}$ in Eq. (3.14b), covers both temporal and spatial conditions from Eq. (3.14a), and absorbs any covariance of a great time and distance change (rate R) into a small one, resulting their Δ forms in Eq. (3.14c). The double integral, in this case, is absorbed into one succeeding integral as the future rate of $\mathcal{C}'$ in Eq. (3.14d) proportional to the rate given by $\mathcal{C}$. Either rate is measured in bytes per second (Bps). It conveys to the implementation of prefix code and processing for prefuzzy bit-pair manipulations, which involves comparing addresses and rows for each set of compressed characters. For example, the minimum 295.6 Bps in Eq. (3.14d), corresponds to a minimum ASCII table-read requirement as 2×128 translatable characters or 256 Bps to decode compressed data in FBAR. Thus, an extra 39.6 Bps memory allocation is needed to conduct a full $\mathcal{C}'$. The Hamming distance used in Solutions 3.4 and 3.5 relative to components $\boxed{\mathbf{G}}$ and $\boxed{\mathbf{TT}}$, is now subject to model construction for an I/O FBAR operation.

3.2.2 Algorithm Components and Model Construction

Using the string sample "`resolved`" from Eq. (3.9c), and considering the **znip** operators used in Example 3.1, suppose we establish a *translation table* $\boxed{\mathbf{TT}}$ like Table 3.1 to read $i \times j \times k \times l$ addresses (rows) containing $4\,y$'s (32 bits) from $\boxed{\mathbf{G}}$. The $\boxed{\mathbf{TT}}$ file is fixed in size = 65,536 rows, and at least requires two key columns to translate $i \times j \times k \times l$, $1\,y$ binary content to xx' binary content and vice-versa.

At the $\mathcal{C}$ phase, the program writes $\boxed{\mathbf{G}}$ with ceratin characters, known as *occupant chars* as output y in a specific row number. This number must correspond to an address that returns original characters

when the occupant char is decompressed.

At the $\mathcal{C}'$ phase, the program reads the grid file contents. From the occupant chars and the row address columns in **TT** or Table 3.1, the program returns *original chars* according to the 'original char' column. Occupant chars are those characters residing in the **G** file. Once the program identifies the occupant char in a particular row number, outputs xx' for that character according to the original char column from the **TT** file. This file in size is always 8 MB for any reference point as a bit-flag for an occupant char corresponding to the original file. The matrix vectors and I/O process layout for the example looks like this

$$\overbrace{\{\texttt{resolved}\}}^{\text{original text}} \xrightarrow{\text{read}} \begin{array}{c}\mathbf{P}\\ \updownarrow \\ \mathbf{TT}\end{array} \xrightarrow{\text{write}} \mathbf{G} = \{\texttt{a},\texttt{b},\texttt{c},\texttt{d}\} \in \left\langle \mathbb{R}^4, \varphi \right\rangle \tag{3.15}$$

and the decomposition of the **G** component after being constructed and written by program **P**, is

$$\begin{bmatrix} \texttt{a} \\ \texttt{b} \\ \texttt{c} \\ \texttt{d} \end{bmatrix} \otimes \overbrace{\left\langle \begin{pmatrix} \textbf{zizp} \\ \textbf{zizp} \\ \textbf{zini} \\ \textbf{zini} \end{pmatrix} \times \begin{pmatrix} \textbf{npni} \\ \textbf{npzp} \\ \textbf{zpzp} \\ \textbf{zizi} \end{pmatrix} \times \begin{pmatrix} \textbf{zini} \\ \textbf{zini} \\ \textbf{zizp} \\ \textbf{zini} \end{pmatrix} \times \begin{pmatrix} \textbf{zizi} \\ \textbf{zpnp} \\ \textbf{zini} \\ \textbf{zinp} \end{pmatrix} \right\rangle}^{\langle \mathbb{R}^4, \varphi \rangle(\beta)} = \begin{bmatrix} \emptyset & \cdots & \emptyset \\ \vdots & \ddots & \\ \texttt{d} & & \emptyset \\ \emptyset & \cdots & \emptyset \\ \vdots & \ddots & \\ \texttt{c} & & \emptyset \\ \emptyset & \cdots & \emptyset \\ \vdots & \ddots & \\ \texttt{a} & & \emptyset \\ \emptyset & \cdots & \emptyset \\ \vdots & \ddots & \\ \texttt{b} & & \emptyset \end{bmatrix}_{\ell}$$

where $\ell = 64\texttt{K} + 4\texttt{B}$, (3.16)

and is a measured output denoting original data, such that

$$\{\texttt{re},\texttt{so},\texttt{lv},\texttt{ed}\} \longleftarrow \boxed{\textbf{P}} \overset{\beta}{\longleftarrow} \overbrace{\left\langle \begin{pmatrix}7\\12\\6\\1\end{pmatrix}, \begin{pmatrix}11\\14\\6\\13\end{pmatrix}, \begin{pmatrix}1\\6\\4\\2\end{pmatrix}, \begin{pmatrix}13\\13\\15\\7\end{pmatrix} \right\rangle}^{\text{address}} = \langle \texttt{a},\texttt{b},\texttt{c},\texttt{d} \rangle$$
$$\boxed{\textbf{P}} \updownarrow \boxed{\textbf{TT}} \tag{3.17}$$

In Eq. (3.16), the constructed **G** by empty values $\emptyset$ with 65,536 rows (64 K), has now an extra 4 bytes (chars) written to it by **P**. Program **P** before writing to **G**, accesses **TT** to write the 4 chars in specific locations denoting original data, given by (3.15) and (3.16). Later, for a decompression, the 4D function φ in the **P** code, manipulates β to obtain original data (its binary). This manipulation occurs when **P** refers to **TT**. The addresses of these 4 chars are identified in the **TT** file by **P** to reconstruct original data according to (3.17).

The left half of the input string '`resolved`' in Eq. (3.15), is illustrated by a hypercube in Fig. 3.1. The grid file is constructed according to Eqs. (3.9), as well as **TT** for the code to access bit flags. Program **P** accesses the occupant chars `a`, `b`, `c` and `d` (known as y in **G**) to return the original chars at the $\mathcal{C}'$ phase. This phase is recognizable between components **TT** and **P** relationship '$\updownarrow$' in Eqs. (3.15)-(3.17).

Table 3.1 The FBAR Translation Table

Row #	Bit-flag address	95 ASCII characters as "occupant chars" representing the "original char" column via the "bit-flag address" column	Original char
1	1x1x1x1	abcde...zABCDE ...Z123...0'~!@$...>,<	aa
2	1x2x1x1	abcde...zABCDE ...Z123...0'~!@$...>,<	¥[a]
3	1x3x1x1	abcde...zABCDE ...Z123...0'~!@$...>,<	•[a]
4	1x4x1x1	abcde...zABCDE ...Z123...0'~!@$...>,<	©[a]
⋮	⋮	⋮	⋮
65534	16x16x16x14	abcde...zABCDE ...Z123...0'~!@$...>,<	ÿó
65535	16x16x16x15	abcde...zABCDE ...Z123...0'~!@$...>,<	ÿü
65536	16x16x16x16	abcde...zABCDE ...Z123...0'~!@$...>,<	ÿÿ

[a] The actual translation table contents or **TT** file for an LDC/LDD access and management
[b] The size of this component is approximately 8 MB.

Once the bit-flag addresses are identified by program **P** subroutines,

thereby compared and interpreted in code, the original data is returned. This is done by bit manipulation $\mathfrak{V}$ from Eqs. (3.5) on β based on addresses to obtain original data (Example 3.1). The intersected addresses occupy in total 4 bytes for the 8-byte sample, since the number of stored chars in the $\boxed{\mathbf{G}}$ file is 4, or 4 bytes. Since an empty $\boxed{\mathbf{G}}$ is static in size, 64 K of rows, all the chars stored with addresses also denote a static allocation. The addresses in this sample respectively are `7x11x1x13`, `12x14x6x13`, `6x6x4x15` and `1x13x2x7`. This is clearly specified in the four-dimensional vector space or storage subspace of bit-flags and addresses by Eq. (3.17). It is denoted by dimensional contents between the angle brackets $\langle\rangle$ notation. In this example, the occupant chars occupying the specific addresses are shown as $\langle \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d} \rangle$ in Eq. (3.17).

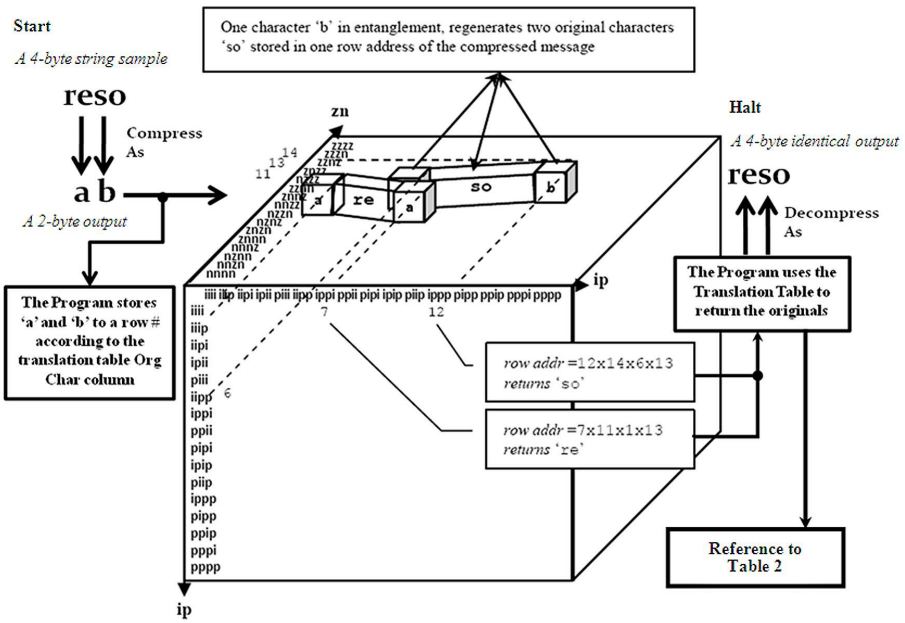


Fig. 3.1 An I/O $\mathcal{CC}'$ process on a '`reso`' string is given in a 4D grid constructor. This constructor shows a 50% LDC with a DE state: the smaller inner-cubes in two places at the same time or "characters in $\pm\pi$-entanglement." This model is a radix to higher DE-LDCs.

The motive for choosing this hypercube (Fig. 3.1) is anchored within the implementation of chars, being converted to binary as modeled back

in Section 2.4, thereby generating self-contained flags within an input char of the $\boxed{\mathbf{G}}$ grid. This results in 50% pure compression, covering 2 chars per entry. From Axioms 3.1 and 3.2, and Definitions 3.4 to 3.12, emitting Eqs. (3.15)-(3.17), we put all of the emerging 1-bit **znip** flags into unique combinations to obtain double-efficiency. We intersect them with other **znip**'s representing a second char input. Therefore

---

Lemma 3.1. Each character output is shared between 1**ip** and 1**zn** dimension, as a stored character. Containing 2 chars $\leftrightarrow$ 2**ip** + 2**zn** = 4 dimensions is done in 65,536 rows or addresses. A minimally 2 original chars from 1 stored char is decoded.

The analogy of Lemma 3.1 is mappable to Moore's Law and Knowledge Management by Gilheany [19], stating: "each time a bit is added to the address bus width, the amount of memory that can be addressed is doubled." Four-bit addresses allow the addressing of 16 bytes of memory, and in Lemma 3.1, are the 4 dimensions containing the 2 chars or xx'. Eight bits allow the addressing of 256 bytes of memory, whereas 16 bits can address 65,536 bytes of memory (and extra work is necessary to address 640 kilobytes of memory, as was the case on the early IBM PCs). In the FBAR case, an 8-bit y can address a 16-bit xx' in one of the 65,536 $\boxed{\mathbf{G}}$ file (portable memory) rows. Therefore, in terms of "an address information sent immediately following the *control byte* as a 16-bit word (65,536 possible addresses)" [47], here, is compressed as a 16-bit xx' to an 8-bit y character in $\boxed{\mathbf{G}}$ rows. So, y in addition to a **znip** flag, plays the role of an 8-bit control byte for 65,536 possible addresses. Thus, we further deduce another lemma:

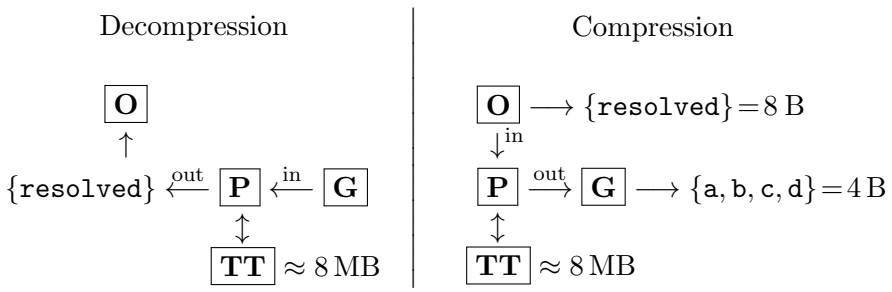
Lemma 3.2. Lemma 3.1 gives a control double-byte $>$ a standard control byte for all intersecting addresses in $\boxed{\mathbf{G}}$. Since an input data is doubly compressed as $\ell(xx') = \ell(y)$, the static access of information in $\boxed{\mathbf{G}}$ is minimally, doubly faster than any other memory access when the compressed data is decoded.

The knowhow of these hypercube processes i.e., data access, compare, interpret after storage, is summarized in Table 4.1, and imple-

mented in our practical section, Section 4, with performance results on the expected Hypothesis 4.1 in Section 5.2.

3.2.3 Summary of Model and Theory

The 4D bit-flag model (hypercube) contains data for I/O transmissions. It maps contents in binary by intersecting their values in four dimensions using FBAR operators, suitable for any data type. The logic incorporated in this model, is of a combinatorial type, i.e. fuzzy, binary AND/OR logic. The rationale to the construction of this hypercube was to observe input characters, each pair of characters to be in two places at the same time. For imagery data types, an integer value is assigned instead, to satisfy an address representing two colors in two places simultaneously, out of the RGB color model for a 50% LDC (recall Section 2.5). Therefore, constructing 2^n memory addresses, in form of a grid file, gives a novel solution of how to compress data in doubles and pairs, losslessly. The fuzzy component of this logic is the middle point connecting binary with more possible states of logic. This connection of minimum to maximum number of states is defined in terms of an interrelated equation for all states of logic, and universal in all codeword representations. This was earlier introduced in Section 2.4. By combining Eqs. (3.15)-(3.17) layout on the sample, we deduce the following components' paradigm. We later use this paradigm for the practical application of the algorithm to execute the operations held by components **TT**, **P**, **G** and original file **O** as follows:



Component **O** as original file, is where the original text or string is located. The practical process and structure of all LDC/LDD components are given in Section 4.

4

FBAR Compression Practice

We implement the 4D model as the algorithm's prototype based on the theoretical aspects of FBAR logic on I/O data transmissions. This prototype should perform DE predictable values. To do so, DE values are enclosed as bits of information, from a $\mathcal{C}$ form to its decoded $\mathcal{C}'$ form in a lossless manner. Finally, we highlight certain details on the definiteness of future entropies supporting a growing negentropy, like Hyvärinen *et al.* [25], proving a universal predictability, contrasting the popular Shannon's method of 1st order to 4th, inclusive of its general orders indeed.

4.1 FBAR Components, Process and Test

To fully implement an algorithm, one must understand how it works in terms of its testable structure and model representation. This is illustrated in Fig. 4.1. Furthermore, the algorithmic components must be introduced in terms of size, their process relationships, executables and data types. The $\mathcal{C}$ and $\mathcal{C}'$ phases of the algorithm, iteratively use the following components: the **G** as the grid file, **TT** as the translation table file, and **P** as the program source code for I/O

executions. The **G** file contains all compressed data representing the original characters. We call this the final compressed FBAR product or compressed file. We now introduce these components, their roles and functions for the $\mathcal{C}$ and $\mathcal{C}'$ implementation as follows:

As proven in theory, from Eq. (3.16), the grid **G** component consists of 8-bit blank entries or $\emptyset$ in 65,536 rows, providing a possible ASCII (256×256)= 64 K of static space for I/O data. The I/O data are processed by the **P** component. This component deals with original contents **O** component as original data which comprises of information built on one or more data types, given by the user. The **O** component, is our input sample and should be tested for a lossless compression $\mathcal{C}$, as well as decompression $\mathcal{C}'$. The **TT** component consists of combinatorial details of any data as a table on bit-flags, row number and occupant chars, available to **P**.

The size of this table is static $\approx$ 8 MB for its self-contained information. Program **P** consists of lines of code to execute $\mathcal{C}\mathcal{C}'$ procedures. It accesses **O** at the $\mathcal{C}$ phase, thereby constructs **G** and puts occupant chars in a specific bit-fag and row number (a prefix address) as compressed data, using **TT** information. At the $\mathcal{C}'$ phase, the same program accesses **G**, and by reading both its contents and addresses identified by **TT**, reconstructs **O**. This has been illustrated in Section 3.2.3.

It is evident for each sample, at least one task T is executed to perform compression parallel to decompression operations. Each conducted task allows one to evaluate the algorithm I/O's in terms of temporal measurement, here bitrate, as well as spatial measurement as bpc or entropy. Once implementation is resolved on this small scale (1 **O** file input), test cases are maximized or extended to the large, in number, and in scale for I/O data integration. This scalability of I/O's would guarantee the correctness of the code on FBAR logic requirements. For example, constructing an abstract release of a character reference column in the prefix **TT** component, based on standard keyboard characters, including whitespace " ", would not exceed 96 entries: 95 printable ASCII characters (decimal # 32-127) as shown in

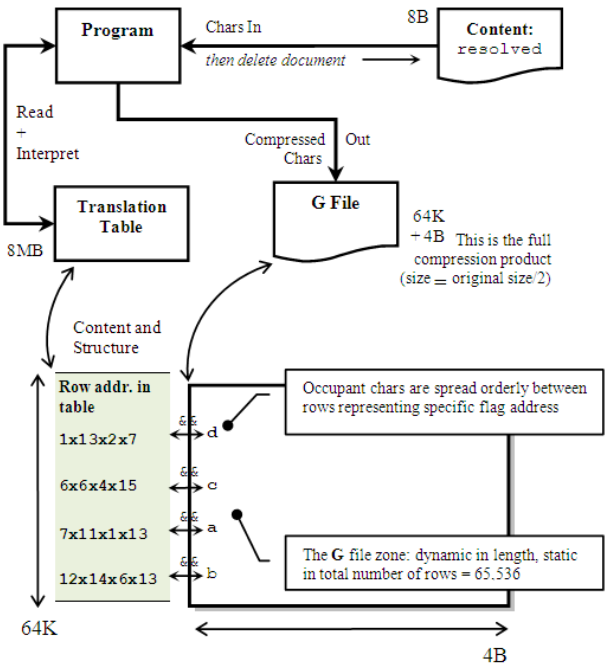


Fig. 4.1 The 4D logic constructor files with an 8 B to 4 B compression.

Table 3.1, plus 1 control character. The latter is used to create a block or a jump character, indicated as {/a, /b,...}, between every {1st, 2nd,...} 95-occupant char entry (or 95 y's). At the $\mathcal{C}$ phase, this in total gives $95 \times 2 = 190$ original char entries per block, and is denoted by the '$\mathcal{C}$(char)' column in Table 4.1. Note that, at the $\mathcal{C}'$ phase, the program uses block chars to return {1st, 2nd, ..., 95th} pair of the original chars, hence forming words and sentences in the right order.

The process design and development of the algorithm is illustrated in Fig. 4.1, with results listed in Table 4.1. The process begins with encoding input data using a dictionary coder, after which a high and low-state prefix fuzzy-binary conversions occur for compression. Recalling Eqs. (2.2)-(3.1), each level of planar projection, from a lower 2D-layer to its upper, forms a 4D *quaternions plane* [4] or hypercube, as a 1-bit flag *bi-vectors group* [32]. This group in the hypercube has

Table 4.1 The FBAR I/O Character Process and Occupation Table

Row address	$\mathcal{C}$(char)#; $\mathcal{C}_r$		Original chars; total		Occupant char	Size (bits)
7x11x1x13	1	2:1=50%	re	2	a	8
12x14x6x13	2	2:1=50%	so	4	b	8
6x6x4x15	3	2:1=50%	lv	6	c	8
1x13x2x7	4	2:1=50%	ed	8	d	8
13x1x1x6	5	2:1=50%	f	10	e	8
6x13x7x11	6	2:1=50%	or	12	f	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮
the same as last	**96**	**1:1=0%**	∞	**191**	**/a**	**16**
8x12x8x12	97	2:1=50%	55	193	a	8
8x12x11x2	98	2:1=50%	5$	195	b	8
⋮	⋮	⋮	⋮	⋮	⋮	⋮

[a] The **TT** file is used for each **G** file-read on the compressed chars as 'occupant chars' to return 'original chars' in the process.

[b] The table portrays the FBAR I/O products as original and compressed data per $\mathcal{C}\mathcal{C}'$ operation. The program compares values in the highlighted cells to return a $\mathcal{C}$ or $\mathcal{C}'$ product.

its own augment in identifying impure 01, 10, and pure states of 11 and 00 for each converted data byte. In return, for an LDD, the converted binary data are recalled via a translation table (Table 3.1) as part of the dictionary or database represented by a set of occupying characters as the compressed version, denoting original data. We recall the original values from the **TT** file for each compressed occupying char, via a "grid file" as a portable memory grid or **G** on single bit-flags to decompress data. The FBAR dictionary consists of data references parsed into the translation table, building a static size of flag information, later used by the program for string value comparisons (the highlighted cells in Table 4.1).

4.2 Methods of Double-Efficiency

We implement the algorithm in form of a prototype. The prototype presents the FBAR model and its encoding/decoding components for DE compressions.

As shown in Fig. 4.2, the prototype representing program **P**, compresses data by loading a document sample. The program uses a mem-

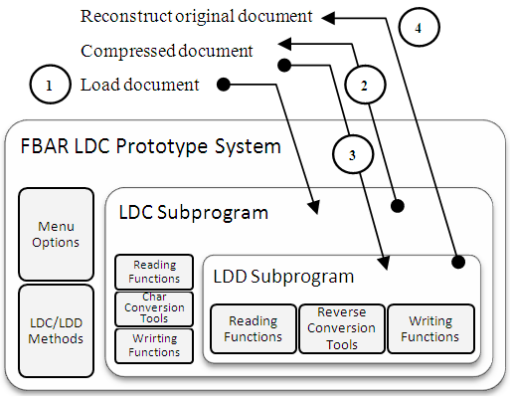


Fig. 4.2 The structural components of the FBAR prototype.

ory grid file **G**, which is a portable file containing single bit-flags in 65,536 rows or addresses. The translation of addresses for original characters, is given in a **TT** file rows with a static size of 8 MB, for any amount of input data manipulated by prefix code. The code interpreter decompresses data, once the flags are compared with the compression result. The decompression uses these prefix flags as compressed data, reconstructing the original document. All of these components, their processes and size are already proven in our theory, Section 3.2.2. In the following sections, we implement the algorithm components with results and evaluate its DE claim on I/O samples.

4.2.1 Algorithm Sample and Test

Assumption 3.1 holds good for the following algorithm:

Proposition 4.1. From Assumption 3.1, suppose for every x character input we have a righthand character x', its sequence appears as $\int_{\text{in}} = xx'$. For a long sequence $\int$, we suppose a sumset $\sum \int = (x_1x'_1 + x_2x'_2 + \ldots + x_mx'_m)$ to be our information input. Our objective in the program is to compress $\int$ to single-byte characters or $\mathbf{F}_y = \{y_1, y_2, \ldots, y_n\}$ or recall the Proof on Proposition 3.1.

Algorithm 4.1. Let program **P** get 2 Characters from left-to-right of sequence $\int$. If **P** continues in taking 2 more Characters with respect to time t, it instantiates a series of tasks $T_1, T_2, T_3, \ldots, T_n$. These LDC tasks for each *information processing cycle* on the sequence appear as

$$\sum T \times \sum \int = (\underbrace{x_1 x_1'}_{T_1+} + \underbrace{x_2 x_2'}_{T_2+} + \ldots + \underbrace{x_m x_m'}_{+T_n}) \xrightarrow{\text{get}} \boxed{\mathbf{P}} \xrightarrow{\text{store}} \boxed{\mathbf{G}} \xrightarrow{\text{out}} \mathbf{F}_y$$

such that, $\mathbf{F}_y = \{y_1, y_2, \ldots, y_n\}$.

The processing cycles in our algorithm should follow

(1) **Input**: entering data into the program
(2) **Processing**: performing operations on the data according to the **TT** file
(3) **Output**: presenting the conversion results, in this case, the **G** file
(4) **Storage**: saving data, or output for future use, in this case, the **G** file.

Now by applying the **znip** operators (as 1-bit flags) on Binary Sequence Character $\beta(\text{`1'}) = \texttt{11111111}$ as our default value in program **P**, to obtain the actual binary on each xx' per task T, we then code our algorithm:

Algorithm 1 comprises of LDC tasks, storing results in the **G** file. From the user, the program gets 2 chars, and inputs it from left-to-right of the file. The program by default contains a character '1' assigning the two concatenated input chars to the '1' (the customized β). Now, the program in line # 6 generates a character representing the 2 chars in the correct row (corresponding row) according to ASCII standard for the same characters. This is further instructed in line # 7 of the code, where the occupying row also represents an address of the compressed chars in the **G** file. The static translation table **TT** file is then used containing prefix addresses for every row out of 65,536 rows to translate, replacing one char with its original two chars

for a 50% compression. This is expressed in line # 8, which gets the original 2 chars from **TT** per compressed char in **G** at the $\mathcal{C}'$ phase of the algorithm. The remaining lines of the algorithm just denote the opposite condition where the new string is again requested from the user to input from the start.

Algorithm 1: A lossless data compression sample

Input: A set of LDC tasks and data conversions
Output: Storing LDC output to **G** file as a compressed string y

```
1  begin
2      while There are still input characters in ∫ do
3          foreach 2 Characters from left-to-right of ∫ do
4              Get 2 input characters x_i x'_i ;
5              Pack 1-bit flags on Binary Sequence Character '1' = x_i x'_i ;
6              Generate an occupant character y_i according to the order of
               x_i x'_i ;
7              Store 1 Occupant Character y_i in the 1-bit flags row # in G
               file;
8              if y_i and row # is in the Translation Table then
9                  Continue getting the next 2 characters from left-to-right of
                   ∫ ;
10             else
11                 Output the code for Pack as New String ;
12                 Restart Packing as New String in G file ;
13                 New String = y ;
14
```

So, we can now initiate the $\mathcal{C}'$ phase of the algorithm in terms of Algorithm 2. This algorithm comprises of LDD tasks, reconstructing results in a new file after reading from the **G** file relative to the **TT** file. From the **G** file, the program reads 1 character from right-to-left and reads the row number in line # 4, comparing it with the 65,536 available translations in the **TT** file (dictionary) in line # 5. The program reconstructs a string of translated characters and adds up newcomer characters to its string to build a full word, or a sentence of the original information, in line # 6-12, where # 12 denotes that, Old Code = New Code. Then the program cleans up the memory at line # 13.

Algorithm 2: A lossless data decompression sample

Input: A set of LDD tasks and data conversions
Output: Decompressing **G** file as an LDD output xx'

```
1  begin
2      while Reading characters row-by-row from end-of-file G do
3          foreach 1 character from right-to-left of string y do
4              Read row # ;
5              if Character y_i is not in (row # and Occupant Character)
                   columns of TT file then
6                  New String z = Get translation of Old Code ;
7                  New String z = String z + Character ;
8              else
9                  Get translation of Old Code ;
10                 Character z_i = 1st or 2nd or ... or nth 2 characters in
                       String ;
11             Replace Character with 2 new characters from the TT file ;
12             New String z = xx' ;
13             Delete temporary row # and row characters ;
```

Relevant to the example provided in Algorithm 2, we further particularize an LDD in Algorithm 3, which is equivalent to Algorithm 2.

Algorithm 3 comprises of LDD tasks sampled from Algorithm 2, practicing a 16-byte (2-char) concatenation of the compressed chars y_i = `d` then `c` then `b` then `a` (in lines # 8, 10, 12, 14), reconstructing a 64-byte result after the concatenation operation is done. This reconstruction of original chars occurs in a new file after reading from the **G** file relative to the **TT** file.

From the **G** file, the program reads 1 char from right-to-left pre-positioned to a block char and reads the row number in line # 4-6, comparing it with the 65,536 available translations in the **TT** file, in line # 7. Finally, The program reconstructs a string of translated characters from line # 8 up to line # 14, and adds up (concatenate) newcomer characters to its string to reconstruct the full word as the output given in line # 15, in this case '`resolved`'.

Algorithm 3: An LDD sample that returns the '`resolved`' string

Input: A set of LDD tasks and data conversions
Output: Decompressing **G** file as an LDD output '`resolved`'

```
1  begin
2      while Reading characters row-by-row from end-of-file G do
3          foreach Last Block Character y_i do
4              if Character y_i is a Block Character then
5                  Read Character pre-positioned to Block Character ;
6                  Read row # ;
7                  Get row address from TT file ;
8                  if Character y_i ='d' and row address = '1x13x2x7' then
9                      Output String ='ed' ;
10                 else if Character y_i ='c' and row address = '6x6x4x15'
                   then
11                     Output String ='lv'+'ed' = 'lved' ;
12                 else if Character y_i ='b' and row address = '12x14x6x13'
                   then
13                     Output String ='so'+'lved' = 'solved' ;
14                 else if Character y_i ='a' and row address = '7x11x1x13'
                   then
15                     Output String ='re'+'solved' = 'resolved' ;
16                 else
17                     Print no data or null compressed ;
18
19             else
20                 Print no block character in range ;
21
```

4.2.2 Maximum LDC/LDDs

Maximum LDCs must respectively satisfy Hypotheses 4.1 and 4.2 from below, as *midpoint* and *maximum* LDCs for the 4D model implementation. These are the updated versions of the hypotheses H.4 and H.5 by Alipour and Ali (2010) [2], which cover discrete interval values of Eqs. (5.2) and (5.3), later introduced in Section 5. One of which as the most radical to our model implementation is Hypothesis 4.1. This hypothesis subsists on the algorithm's predecessors, which deal with minimum and middle-point LDCs. The minimum LDCs mainly project onto the dynamic memory allocation points which are of interest when

optimization of the algorithm is concerned. For example, during **G** and **TT** I/O operations, the dynamic size of **G** must be managed by dynamic *read*-and-*write* of y's as minimum compression, maintaining a maximum 50% compression on all characters when only 1 **TT** file is being read. We implement maximum LDCs by self-containing the static memory allocation points as mid-points in a **TT** file bit-flag addresses, specified back in Sections 3.1 and 3.1.2, as follows:

The following represents "midpoint LDCs" on the version-to-version 4D model

---

Hypothesis 4.1. A sequence of bit-flags representing double-efficient compressed data in FBAR, once reused by its translation table adjacent to other purely compressed data, results in a decompressed message.

---

whereas its null hypothesis would be

Hypothesis 4.1_0 The sequential recall and reuse of bit-flags from memory/grid, is firstly minimum-compression dependent, and secondly, unachievable for an identical data reconstruction.

The following represents "maximum LDCs" on the version-to-version 4D model

---

Hypothesis 4.2. A sequence of compressed data in form of four-dimensional 1-bit flags, when partitioned into memory or confined in information space/grid, results in a maximum LDC possible $\geq 87.5\%$ with optimal bitrates.

---

whereas its null hypothesis would be

Hypothesis 4.2_0 The compression of any data length into one single-byte is firstly minimum-compression dependent, and secondly, unmanageable and irreversible for data reconstruction like Hypothesis 4.1.

According to Sections 3-4.2.1, Hypothesis 4.1 is by now achieved, which addresses the 4D model implementation, independent of its null hypothesis limitations due to the 4D model characteristics i.e., the **TT** and **G** components and their relationships. These dimensional relationships on I/O $\mathcal{CC}'$ data are discussed as follows:

For an 87.5%, obviously, the column with 96 characters will not change, however, the '$i \times j \times k \times l$' column in its configuration becomes '$i \times j \times k \times l \quad i \times j \times k \times l$', and the last column with 2 characters, becomes 8 characters, since the cubic representation of the '1st $i \times j \times k \times l$' with the '2nd $i \times j \times k \times l$' has a second *non-commutative symmetric* format: '2nd $i \times j \times k \times l$' with the '1st $i \times j \times k \times l$', giving four distinct addresses simultaneously. So, for the former, this means, 2 original chars result in 1 char in compression (2:1 or 50%), and for the latter, 8 original chars result in 1 compressed char ($100\% - 12.5\% = 87.5\%$ or 8:1 bytes) as an 'occupant char' (see, Table 3.1), occupying a row in the compressed file **G** in Fig. 4.1. The symmetry '2nd $i \times j \times k \times l$' with the '1st $i \times j \times k \times l$', altogether, gives four distinct double-char addresses simultaneously, i.e., an 8:1 LDC. This satisfies 65536^4 **TT**ables $= 1.84 \times 10^{19}$ unique combinations, or, 16 exabytes (EB) of grid rows. In case of columnar symmetry in two translation tables, $65,536^2 = 4.1$ GB, handles the 16 EBs when column values are co-intersected by a comparator matrix in our code, residing in the LDD subprogram comparator (Sections 3.2.1 and 3.2.2). So, four 64 K grid row combinations, handle the same EB values in four parallel tables. This requires a complex compression matrix coding as symmetric and antisymmetric access of **TT** data on current machines equipped with dual CPUs. The reason is having an optimized version by creating multi-threads on the four parallel **TT** files (*prefix data*) per LDC operation. To this account, we pose a formulation:

4.2.3 Complex Matrix Coding

Let $\mathcal{C}_{\text{matrix}}$ be a variable for a compression matrix with decision nodes in FBAR code on either spatial or temporal measurements made by

Alipour and Ali (2010) [2], where its values to process **TT** data for **G** read/write operations would establish

$$\mathcal{C}_{\text{matrix}} \propto \mathcal{C}_{\max}\left(\beta, t_L\right) \ , \tag{4.1}$$

where $\mathcal{C}_{\max}$ is a data function for the highest possible layer of lossless compression (HLLC) by FBAR, and t_L is the time taken to process **TT** and **G** files for an I/O binary sequence β. This deduces

$$\therefore \mathcal{C}_{\text{matrix}} = M \times \mathcal{C}_{\max} \ . \tag{4.2}$$

In fact, time length t_L corresponds to the current time where present pseudocode decision points would not exceed the limit of 'if-else statements', even in case of extending them into the 16 EB scenarios. In other words, a 64-bit microprocessor, in principle, handles at most, 18 EBs of space [33], if based solely on 1 **TT**able. So, we program 4 **TT**'s to just have a 32 MB table with our FBAR package. In our later results in Section 5.2, Eq. (4.1) becomes evident in terms of *cyclomatic complexity* M [37], with a conjecture of just including the concatenation operator '+' in its stateful extension. This makes FBAR as efficient as possible in its $\mathcal{CC}'$ product results. The more **TT**s included in Eq. (4.1), the more complexity or decision nodes of code loops. For the 'if-else' statements satisfying a $\mathcal{CC}'$ (Algorithms 1 and 3), we deduce that an efficient complex matrix code dedicated to the 4 **TT**-read per **G**-write content, requires $M = M_{\text{previous}} + 3$ (as sampled below in Algorithm 4). The reason compared to the previous pseudocodes is that, the concatenation '+' operator, triples on decision points in the new complex version in terms of Algorithm 4. Apart from each short-circuit 'AND' operator adding a 1 to the M [50], the number of if-statements shall remain the same for the LDC/LDD codes (recall Algorithm 2). The simulated results on Eq. (4.1) are listed in Table 5.1.

Algorithm 4, below, comprises of LDD tasks and is analogous to Algorithms 2 and 3, but with the ability to manage large amounts of reconstructable data through complex coding ($\mathcal{C}_{\text{matrix}}$ code). The algorithm hypothetically returns 64 bytes (8 original characters) represented by 1 single byte (1 compressed character), as standardized in the **TT** file, after reading all translatable bit-flag combinations in 4×65536 conjoint rows (Fig. 4.3). The character reconstruction method,

from line # 8 and 9, results in a new file after reading from the **G** file relative to its **TT** file. From the **G** file, in line # 2-5, the program reads $n \geq 1$ character from left-to-right until it reaches a block character y_n, and then reads its row number in line # 6. It compares the character(s) located between two block characters y_n and y_{n-1}, with the 65,536 available translations in the **TT** file, in line # 5-7. The program reconstructs a string of translated characters and adds up newcomer characters to its string to reconstruct the full word, sentence or original information, from line # 8-10. The program cleans up the memory, being well-aware that if any input character is not given according to line # 11 to the algorithm, null ($\emptyset$) is returned, specifying no code block in range, in line # 14.

Algorithm 4: An 87.5% LDD sample: 1 compressed character returning 8 original characters

```
   Input: A set of LDD tasks and data conversions
   Output: Decompressing [G] file as an LDD output 'resolved'
 1 begin
 2     while Reading characters row-by-row from end-of-file [G] do
 3         foreach Last Block Character y_i = y_n do
 4             if Character y_i is a Block Character then
 5                 Read Character(s) pre-positioned to Block Character ;
 6                 Read row # ;
 7                 Get row address from [TT] file ;
 8                 if Character y_i = 1 Occupant Character and row address =
                   'i × j × k × l'+'i × j × k × l'+'i × j × k × l'+'i × j × k × l'
                   then
 9                     Output String = '1st 2 characters' + '2nd 2 characters'
                       + '3rd 2 characters' + '4th 2 characters' = '8 original
                       characters' ;
10                 else
11                     Print no data or ∅ compressed ;
12
13             else
14                 Print no block character in range ;
15
```

4.2.4 Component Relationships

To perform double, or even quadruple efficiencies, the program must refer to a **TT** file comprised of bit-flag addresses from the 4D model, with their corresponding original chars as well as occupant chars in code representing their original char positions. These I/O references have been shown in Table 4.1. For quadruple efficiencies, the comparator's if-else statements are expanded in terms of reading multiple **TT**s in parallel. This returns for each unique address, 4 original chars denoting a 75%, and 8 original chars denoting an 87.5% compression. The 4 original char version, requires the complex matrix to read data from 2**TT**s correspondingly, since each **TT**, according to the 50% LDC, returns 2 original chars per 1 compressed char as an occupant char. Therefore, 2**TT**s return 4 original chars at the $\mathcal{C}'$ phase. So, for performing the $\mathcal{C}_r$ = 8:1, or the 8 original char version, we require 4**TT**'s for each data-read by the comparator to succeed 4 address translations, or

$$4\,\mathbf{TT} \times 4\ \textit{occupant char translations} = 8\ \textit{original chars}. \tag{4.3}$$

In Eq. (4.3), the number of translations is n in n**TT**, and applies to an n-hypercube, $2^n\,n!$, by Coxeter *et al.* (2006) [16], as $2^{16} \times n = 16^{4\mathrm{D}} \times n$**TT** flag combinations.

TTable 1 ⟷	TTable 2 ⟷	TTable 3 ⟷	TTable 4
1x1x1x1	1x1x1x1	1x1x1x1	1x1x1x1
…	…	…	…
16x16x16x16	16x16x16x16	16x16x16x16	16x16x16x16

Fig. 4.3 Translation tables in parallel intersections for an 8:1 LDC. The $\mathcal{C}_{\text{matrix}}$ code accesses data for read + write operations from all **TT**s with a configuration of addresses only, building a (16×16×16×16)×4 or a $16^{4\mathrm{D}}$×4 matrix for a **G** file's $\mathcal{CC}'$ write operation.

In Fig. 4.3, Eq. (4.3) is illustrated in form of a flag-char relationship diagram, corresponding to bit-flag values (char address) of the 4D model. As a result, we return the original chars exactly as expected at the $\mathcal{C}'$ phase. For highest DEs, we therefore extend the number of **znip** columnar combinations from the previous **TT** in terms of row-by-row

intersections. This is called a 4 table-based algorithm. It delivers double DEs, and thereby, quadrupled efficiencies as well. We described this in terms of fulfilling 4.1 GB and 16 EB combinations in the above paragraphs, respectively. In other words, on all occasions, the program's interpreter/comparator matrix must be able to handle 1, 2 and 4 **TT**s for all intersections between them, needing just 8, 16 and 32 MB static size on an x86 machine, instead of the EB barrier denoting no columnar interactions whatsoever. For example, an intersection of `1x1x1x1` with `1x2x1x1` with `16x16x16x15` with `1x4x1x1`, from **TT**s 1-to-4 in Fig. 4.3, returns $^{aa}¥^{a}•^{a}©^{a}$ original chars. Hence, the length of 64 bits is thus *self-contained* and *fixed* by the program's comparator efficiently, using just 8 bits out of the 32 MB of the traversed tabular space, denoting an 87.5% compression.

5

Simulation Results, Contribution and Analysis

5.1 Contribution

The main contribution in this paper is presenting a new model on self-embedded flags from Section 3.2.2. It allows an LDC algorithm to conduct a new coding technique i.e., doubling the efficiency between two points of data transmission (DE). The key component of an FBAR algorithm is the **TT** file or translation table whereby double-efficiency is conducted. Moreover, the **G** file, as another component, holds sizes denoting double compression ratios after each full I/O write of contents. This established a key difference in techniques, observed between the FBAR algorithm and other LDC algorithms. According to the "theory of data compression" [12, 44], we conclude that almost every LDC uses Shannon entropy as its 'logic base' in conducting a lossless compression. In fact, repetition of characters in a certain frequency based on the theory of probability is embedded in such LDCs. In layman's terms, information entropy is the same as "randomness." A string of random letters and numbers along the lines of "5f78HJ2Z2Xp4V7Vb6" can be said to have high information entropy, or, large amounts of entropy, while the complete works of Shakespeare can be said to have low infor-

mation entropy. Their LDC products are quite variant, which depend on content pattern probability or character rate of recurrence. FBAR LDC, however, deals with the computation of binary logic regardless of content size and type, whereas other techniques are not bothered about. Binary logic in FBAR deals with individual bits, their combination, repetition, cubic conservation, regardless of character repetition or content type. This means, based on a fixed size character reference table, Table 3.1, we derive a more certain equation (least zero order H values), which is logarithmically the least probabilistic with discrete entropy (in bpc), compared to Shannon's entropy rate H on English source alphabet $\mathbb{A} = \{a, b, c, d, \ldots, z, \texttt{space}\}$ given by

$$H_{\mathbb{A}} = \log_2 m = 4.75\,\text{bpc}\ ,\ \text{where}\ m = 27\,, \tag{5.1}$$

and for higher orders of H, for a given text source made up of English alphabet letters, becomes 4.07, 3.36, 2.77 and 2.3 bpc, respectively. In FBAR, however, fixed values of $\mathcal{C}$ for every double-efficient order remain

$$H_{\wedge\vee(b)} = \log_b |\beta| = [0, 2]\ \text{bpB} \tag{5.2}$$

and for a binary sequence β, the binary probability of two states, $b = 2$, constructing 1 char, entropy H becomes 2, 1 and 0 bits per byte (bpB), regardless of source for a given fixed size binary reference code (compare this with Mackay (2005) [34]). This makes the algorithm to compute information reliably based on fuzzy-binary, rather than string characters. The DE process in Eq. (5.2), evaluates every character by using and-or, pure and impure logic, and from there, further LDCs between bits of information. Equation (5.1), however, deals with the random process to evaluate the whole sequence of characters using probability theory for an LDC result. Equation (5.2), by comparison, improves less dependency on symbolic representations, and has a firm dependency on binary logic, thereby, fuzzy, and finally, DE logic. The latter, however, remains quite intact with higher orders of probability equations promoting Shannon zero-order through third-order and general models, in simplistic sizes of LDC. Reasoning that, DE logic by itself is based on probability behavior over bit states. We define the relationships between logical events, "bit states" of the FBAR algorithm, as LDC

causality in form of supreme states of compression. For any data type at a DE level, the current model (Fig. 3.1) providing DE compressions holds good for superdense coding operators by Bennett [7].

In our next report, we improve our model design, reconfiguring **znip** flags in an extended translation table in aim of super-compressing an encoded message, thereby decode and decompress. The FBAR logic would then be called a DE-negentropic and-or logic (DENAR) in its ultimate performance of LDCs. Hence, a *negentropy* < 0 bpB of Eq. (5.2), denoting DE's above 87.5% compression for a universal predictability, is not farfetched in reality.

5.2 The FBAR Entropic Comparisons

The following bar chart, Fig. 5.1, gives a compression ratio comparison for our chosen algorithms. In this case, we chose WinZip, GZip, WinRK algorithms based on their respective ranks (see, e.g., Bergmans (1995) [8]). The detailed empirical and statistical analyses of these algorithms compared to FBAR, based on the non-parametric Friedman test, have been initially reported by Alipour and Ali (2010) [2].

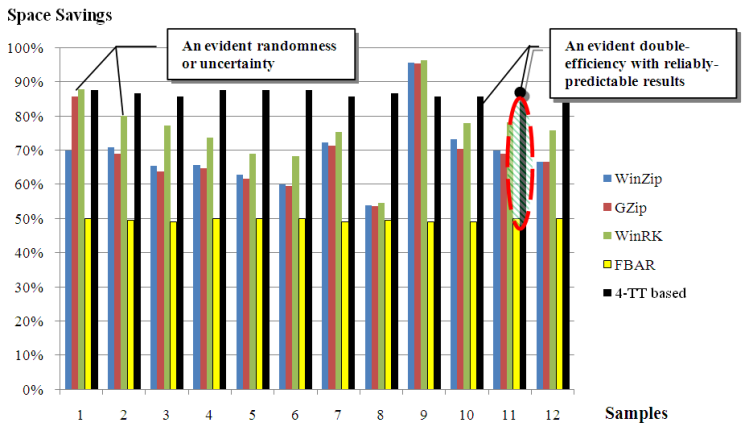


Fig. 5.1 LDC ratio comparisons between FBAR/4-**TT** based and other algorithms on the 12 char-based documents selected by random. The ranking of the algorithms and their non-parametric Friedman comparisons were motivated and defended in the initial thesis of Alipour and Ali (2010) [2].

Figure 5.1, further shows the difference between uniformity of LDC values on FBAR, compared to the random performance (or uncertainty) of others with a highly-ranked algorithm, WinRK. Bitrate and memory usage comparisons between FBAR and WinRK algorithms are given in Fig. 5.2.

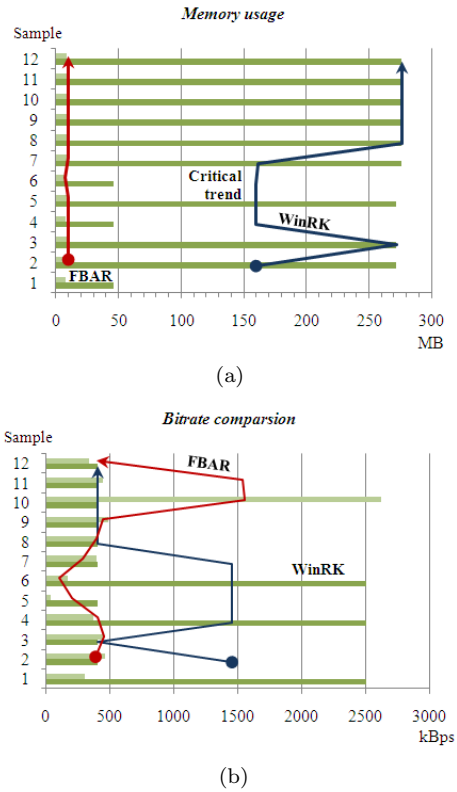


Fig. 5.2 Algorithmic performances of FBAR and WinRK. (a) Memory usage; (b) bitrate.

The selection of an LDC algorithm depends on the following criteria as applicable characteristics to all LDC algorithms:

(1) The ability to compress input data losslessly regardless of type, size and complexity. If data type matters, e.g., being of textual type, must compress textual data losslessly, i.e., the

$\mathcal{C}'$ data after $\mathcal{C}$ must be identical to the original.
(2) Use memory for data access and management issues efficiently, e.g., data rate and spatial occupation of bits during $\mathcal{C}$, i.e., when encoded and referenced upon.
(3) Must have a dictionary coder for validating data, referencing and de-referencing them during the reconstruction phase of data i.e. $\mathcal{C}'$.

Our test samples were char-based, suitable for any ASCII string conversion, e.g., *.txt, *.tex and *.htm, and were selected by random, meaning in terms of content size, content characters i.e. quantity relative to the supported char data types, which were also different, no matter how limited our choice for the current version. The data type is random on the FBAR's evolutionary grade (Fig. 5.2) due to its explicit behavior in converting chars to binary and vice versa, via the 4D translation table.

The current version of FBAR supports char-based data types, and in the future, the conversions of different binary forms are achievable due to the universality of the ASCII table associated with our translation table, which is a midpoint $\mathcal{C}$ converter of both, giving a new data type standard for flags. Meaning that, the random selection of samples is evident, since the documents were all ASCII-based as char-to-binary by our algorithm. The selection of packages or LDC algorithms, however, was not random and was based on the three criteria given above relative to their ranks.

Figure 5.1 shows a distinctive alignment and correlation of $\mathcal{C}_r$'s of the FBAR and DE versions to others. Comparatively, the new algorithm is more reliable in LDC results with consistence in spatial efficiency values performed on compression, which is due to having fixed-size components like the **TT** file, and contrasts other algorithms that create a new dictionary code for each I/O load.

Based on the three characteristics criteria, Fig. 5.3 portrays the selection of FBAR type as oriented to DE-negentropic type during implementation. Its simulation grade on x86 machines, reaches 87.5% fixed LDC scenarios. The 87.5% LDC indicates the lower-bound interval of Eq. (5.2). The zone indicating x86 limits for the hybrid version, shown

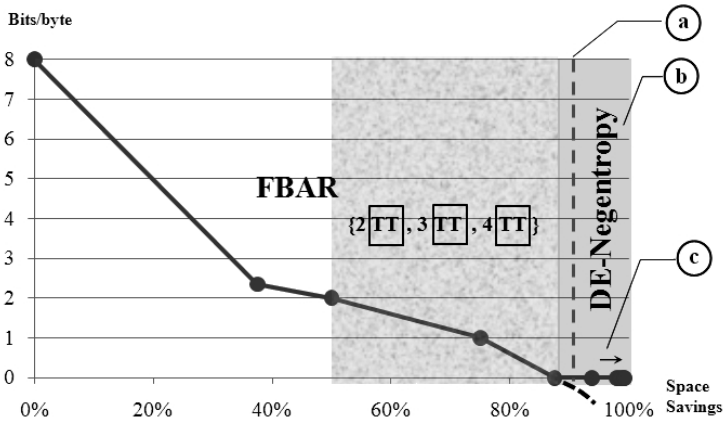


Fig. 5.3 The pure FBAR in its version-to-version evolution, mutates to a DE-Negentropic and-or (DENAR) logic via its hybrid version on x86 machines; a) the EB barrier; b) quantum machines; c) negentropy leading to a universal predictability.

as FBAR∼DENAR in Fig. 5.3, inclusive of the regular FBAR versions (1 **TT** to 4 **TT** usage), continues to expand within the DENAR territory. This means, the structural integrity of the FBAR dictionary (or the 4D grid model) at $H = 0$ bpB final version on x86, is significantly changed in favor of superdense coding or a quantum machine territory. In one word, *FBAR mutates from version-to-version with uniformly-fixed values on space savings.*

The negative entropy of Eq. (5.2), as Eq. (5.3), denotes universal predictability, giving values $\geq 93.75\%$ compression, as estimated. This model could be considered as a solution to *complex negentropy problems* [25] in signal processing and information theory, making the current model universal for negative and positive ranges of Eq. (5.2). Alternatively, we constrict Eq. (5.2) in terms of

$$-H_{\wedge\vee(b)} = \log_b |\beta| < 0 \,\text{bpB} \;, \;\; \text{where } b = 2 \,. \tag{5.3}$$

For the positive range of Eq. (5.3), as Eq. (5.2), twelve documents were given to four different LDC compressors (in random order), relative to their bitrate performance for each LDC execution. The spatial and temporal estimates are given in Table 5.1. Process time of a test, and

percentages of compression, were also measured. The resulted data on both spatial and temporal performances are expanded from 1**TT**, to 4**TT** inclusions. From Eqs. (4.1) and (4.2), in Table 5.1, the t_L on 4**TT**s, without parallel processing, is hypothesized for $\mathcal{C}_{\text{matrix}}= 3\mathcal{C}_{\text{max}}= 5.5\,\text{s}$, satisfying the 'extended if-else' nodes in form of a $16^{4\text{D}}\times 4$ address format (Section 4.2.4). The nodes are quite local in the $\mathcal{C}_{\text{matrix}}$ code, and the HLLC merely concentrates on read/write operations, constituting the current LDD×1**TT** vs. LDD×4**TT** scenarios.

Table 5.1 Estimates on $\mathcal{CC}'$ phases with rate performance on FBAR via 1**TT** against 4**TT** file-set

No.	File	Size (KB)	t_L = CPU time/s LDC/LDD × 1**TT** vs. 4**TT**			Compressed size (KB)	bpc
			$\mathcal{C}_{\text{matrix}}$	LDC	LDD		
1	text	60.16	0.06	0.2 : 0.26	0.24 : 0.3	30.39 : 7.52	{0,2}
2	book1	662.34	0.42	1.45 : 1.87	1.40 : 1.82	334.62 : 82.79	{0,2}
3	book2	1730.54	2.25	3.95 : 6.2	3.43 : 5.68	874.28 : 216.31	{0,2}
4	paper1	51.28	0.03	0.14 : 0.17	0.20 : 0.23	25.9 : 6.41	{0,2}
5	paper2	114.73	0.345	0.375 : 0.72	0.32 : 0.665	57.96 : 14.34	{0,2}
6	paper3	10.02	0.03	0.06 : 0.09	0.10 : 0.13	5.06 : 1.25	{0,2}
7	web1	730.24	0.66	1.85 : 2.51	1.71 : 2.37	368.92 : 91.28	{0,2}
8	web2	584.1	0.6	1.49 : 2.09	1.36 : 1.96	295.09 : 73.01	{0,2}
9	log	1797.77	0.81	3.7 : 4.51	3.58 : 4.39	908.25 : 224.72	{0,2}
10	cipher	759.42	0.12	0.29 : 0.41	0.32 : 0.44	383.66 : 94.92	{0,2}
11	latex1	204.3	0.09	0.46 : 0.55	0.49 : 0.58	103.21 : 25.53	{0,2}
12	latex2	151.99	0.09	0.45 : 0.54	0.92 : 1.01	76.78 : 18.99	{0,2}
0	**TT** file	≈8 MB	N/A	N/A	N/A	N/A	{0,2} read
	Total	6856.89	5.505	14.41 : 19.92	14.07 : 19.57	3464.12 : 857.07	{0,2}

[a] Estimates on compression with rate performance on FBAR's LDC and LDD using 1**TT** vs. 4**TT** file-set. The main columns representing the LDC × 1**TT** vs. 4**TT**, and LDD × 1**TT** vs. 4**TT** ratios are shown in the three sub-columns of the third column, $\mathcal{C}_{\text{matrix}}$, LDC and LDD.

[b] The base file being accessed by the program for read and write operations is the **TT** file ≈ 8 MB.

The bitrate results, are the least random compared to other LDCs, and thus are in conformity with the spatial results in Fig. 5.2. Figure 5.2 shows the bitrate performance on the 12 test documents, with their critical and optimal trends. The adapted version focusing on HLLC results for a simulated 50% DE-LDC is given in Table 5.1. In it, we further computed the time factor as CPU time in seconds, and the results of both LDD and LDC are reflected in that table. The bitrate, rela-

tive to memory usage, was observed between the algorithms on 'space savings': WinRK vs. FBAR. As we can see, for higher bitrate performances, WinRK has a critical usage of memory per input sample. In some cases, even having 10 kBps for encoding and decoding data, required 800 MB memory on a 2 GHz Athlon CPU. This drawback ranks WinRK's memory performance lower than expected compared to FBAR.

When we associate the left chart values with the right chart in Fig. 5.2, it is evident that the empirical data relative to memory usage on FBAR is optimal, and uniformly correlated except a bitrate jump on sample # 10. This is due to the excessive repetition of characters within the sample. The original input chars were ignored due to their pattern simplicity for storing data. Hence, the algorithm is not forced to take in too much information, thus its computation. The average bitrate was estimated 475 kBps for FBAR, and 925 kBps for WinRK on 12 samples. For the EB barrier in Fig. 5.3, we refer to the explanations provided in Section 4.2.2, which concern $\mathcal{C}$'s $> 87.5\%$ scenarios, addressing fixed $\mathcal{C}_r$ values of DENAR-LDC.

The evolutionary grade of FBAR in Fig. 5.3, further illustrates the elicited $\mathcal{C}_r$ ratios, respectively giving 8:8 for 0%, 8:4 for 50%, 8:2 for 75%, 8:1 for 87.5%, 8:-1 for 93.75%, 8:$\ldots -\infty$ for $\approx$100%. These ratios correspond to Eqs. (5.2) and (5.3) discrete intervals, from present to FBAR future versions, supporting the spatial boundary limits of Hypothesis 4.1, for a universal predictability in Hypothesis 4.2. To avoid ratio confusion on Eq. (5.2), for $b = 2$, we rather state, $2^H = B$ or *bytes* for 8:B, hence Fig. 5.3 on Eq. (5.3), does not scale the vast negative space $0\,\mathrm{B} < 2^{-H} < 1\,\mathrm{B}$ for 8:$-H$ on DE-negentropy. The predictable H-limit is indicated by a dotted curve, descending within the negative space of Fig. 5.3. For example, according to Eq. (5.3), $b = 2$, and for a 93.75% space savings, a $2^{-1} = 0.5\,\mathrm{B}$ is gained. Meaningly, 0.5 *compressed* B is gained against a 100%-read on 8 *original* B, or an 8 B *input* : 0.5 B *output*, as $\mathcal{C}_r = \{100 : 6.25\}\% = 93.75\%$. This is a highly compressed version of data, which ascends to attain an ultimate DENAR-LDC.

5.3 Costs and Future Work

Nowadays, compressors accumulate much more memory space, even more than 250 MBs, e.g., WinRK in Fig. 5.2. This is significant when *overhead information* and *memory caching* issues are studied from the usability aspect of the algorithm. By employing cache memory, the **TT** file is temporarily loaded into memory and accumulate much lesser space. This is imperative for huge data transmissions above TB limits on the network and elsewhere, satisfying the EB limits explained above.

There would be additional cost in terms of the 4**TT** process and management issues discussed in Section 4.2.2, or see, its relevant code complexity relations in the same section. In fact, 32 MB in size, as part of the algorithm's package, is affordable for users. The more the users pay, the greater guaranteed LDCs they get. For example, for 50%, 1**TT** shall suffice, which is 8 MB as part of the package or program, and not something being generated each time we load a document to the program (unlike other compressors). The nature of these **TT**s is being robust in their dimensions, content, and always static in size. In fact, a new standard to ASCII translations could be deemed as an ISO extension, sitting next to the ASCII table.

In the current discussions, the algorithm components were thoroughly discussed to prove the spatial and temporal limits of Hypothesis 4.1. However, the main aim was to prove the algorithm's DE logic and model representation following its applicability and usage in code. Future works shall focus on Hypothesis 4.2 addressing ultimate DENAR-LDCs, its robustness, complexity, reliability, confidence, etc., relative to the universality of the 4D model. We have, however, showed confidence on predictability rated as high as 100% due to having LDC values predicted before compression. This was done by satisfying Hypothesis 1.1 via the **TT** and **G** file components, having any randomness "self-contained" within their code.

The FBAR, based on our current analysis and results, addresses 1 terabyte (TB) and beyond the EB limits (Sections, 2.5 and 4.2.2), hence its components, as a whole, are applicable to databases as VLDB transactions, algorithms and performance.

Our future works also focus on VLDB-related issues, which entail

the qualitative aspect of the algorithm. For instance, suppose within the context of knowledge modeling we employ the translation table with the FBAR interpreter. Since the table is ASCII-based, the code interpreter from Sections 4.2 and 4.2.1, classifies data for a unique set of conversions (a compression or encoding). At the decoding stage, the interpretation of input knowledge by the computer, e.g., the English language, gives an output resultant satisfying the compactness of first-order logic [17, 18]. This information product is too content-based and semantically familiar to the human knowledge. This promotes the usability aspect of FBAR on databases, information retrieval systems, their design and architecture.

From the quantitative viewpoint, Eqs. (5.2) and (5.3) efficiency against Shannon relations like Eq. (5.1), becomes evident in future publications. The current work briefly discussed Shannon, and proves the relatedness of different logic types. The current LDC algorithm is based on FBAR logic and does not use Shannon. Shannon in this work, however, is referred to make algorithmic comparisons, and solely to demonstrate the uniqueness of FBAR compared to randomness, i.e., redundant symbols of information stored and observed in other LDCs, with relatively well-known entropy orders to Shannon codeword. (Recall e.g., Section 2.1).

In future reports, we propose an n4D-superdense model to enhance and promote the current hypercube by combining it with the famous Bloch sphere [13] mapping an N-point probability sphere of data onto its surface for a maximum DEN-LDC. This results in a new hypothesis, yet to be proven as follows:

Hypothesis 5.1. Compressing all of our Universe's data into one single bit or a near-zero byte within an infinitesimal time-frame losslessly, is by combining the 4D FBAR hypercube with a Bloch sphere, giving an n4D-superdense model. Its translation table will contain all logic states of information, emitting a complete $\mathcal{C}'$ product.

Conclusion

In this paper, we have demonstrated that the 4D bit-flag model, in its logic base, is self-contained for lossless data compaction and compression. Thus, it is more reliable for data read and write, compared to probabilistic methods in implementing a lossless compression, due to having predictable values in its LDC results.

By using this model, an LDC program converts any ASCII character to its compressed version, double-efficiently in an exponential manner. Based on our results, the 4D model proves universal predictability from one version to another via a universal translation table for data conversions. Thus, it gives reliably fixed results in every data conversion output per se. Therefore, this makes all algorithmic components of the model, and its self-contained bit-flag values universal as well.

Perceivably, the present FBAR compresses data with fixed compression ratios, where other compressors do not. Almost every lossless compressor uses probabilistic Shannon entropy as its 'logic base' in conducting LDCs. FBAR, however, achieves higher space savings, above 50% as estimated, simulated and discussed in theory from its DE coding technique, as well as a 4D model representation. The FBAR products were studied from an LDC and LDD viewpoint in terms of delivering

DE-$\mathcal{C}$'s $> 87.5\%$, or, a DEN < 0 bpB. It is conclusive that, this algorithm contains predictable values for every double-character input. The predictable fixed value, allows a user to know how much physical space is available within a reasonable time, before and after compression. This confidence in predictability makes FBAR a reliable version compared to the probabilistic LDCs available on the market.

The FBAR algorithm is novel in most aspects such as encryption, binary, fuzzy and information-theoretic methods such as probability. To this account, the fields of interest encompass the newly-born FBAR model useful to information theory mathematicians, electrical and computer engineers as well as computer scientists for its logic, and software engineers for its applications.

Acknowledgements

I gratefully thank my supervisor, Dr. T. A. Gulliver, at the Department of Electrical and Computer Engineering, University of Victoria, Canada, for his constructive remarks in improving the model and theoretical aspects of this work for a better presentation. I also like to thank the anonymous reviewers for their valuable comments and suggestions.

Notations and Acronyms

In this section, we present the main acronyms and notations used in this article with recognition of those notations and definitions formulated in Table 2.1, and elsewhere. We finally outline in a separate table, the key interpretation conceived for these acronyms to avoid any confusion when studying the article.

Table 5.2 Main acronyms and notations that have been used throughout this article.

Acronym	Meaning	Employed notations from Table 2.1 and elsewhere
LDC	Lossless Data Compression	$\mathcal{C}_r, \mathcal{C}, H, \int, \ell$, **O**, **TT**, **P**, **G**
LDD	Lossless Data Decompression	$\mathcal{C}_r, \mathcal{C}', H, \int, \ell$, **O**, **TT**, **P**, **G**
4D	Four-Dimensional	$\mathbf{F}_{xx'}, \mathbf{F}_y, \mathbb{R}^{2^n}, \mathbb{C}\ell_4\mathbb{R}^4$, $h^2\mathbf{e}_{ij}^2, Q_{xx'}$, $Q_y, A_{r\times 4}$
FBAR	Fuzzy Binary AND-OR	$\Phi_\wedge$, $\Phi_\vee$, $\bigcap, \bigcup, \mathcal{A}, \tilde{\mathcal{A}}$
O	Original	xx'
TT	Translation Table [a]	$i \times j \times k \times l$, xx', y
G	Grid [b]	$\int_{\text{out}}$, y
P	Program	$\int_{\text{in}}$, xx', $\int_{\text{out}}$, y, $\boldsymbol{\mho}$, **znip**, $\hat{\boldsymbol{\mu}}$, **O**, **TT**, **G**, $\mathcal{C}_{\text{matrix}}$
I/O	Input/Output	$\int_{\text{in}}$, xx', $\int_{\text{out}}$, y, **O**, **TT**, **G**, **P**
ASCII	American Standard Code for Information Interchange	char $\leftrightarrow \beta$
ISO	International Organization for Standardization	N/A
CPU	Central Processing Unit	$\sum T_i, t_L$
VLDB	Very Large Database [c]	$\sum \mathbf{O}_i \gg \mathbf{G}$
RAM	Random Access Memory	A
B	Byte	$\hat{\boldsymbol{\upsilon}}$
Bps	Bytes per second	$R, R_{\mathbb{H}}$
b	Bit	$\hat{\boldsymbol{\upsilon}}, \beta, b$
bps	Bits per second	R, R
KB	Kilobyte	$\ell(A_{r\times 4})$
MB	Megabyte	$\ell(A_{r\times 4})$
EB	Exabyte [c]	$\sum \ell(A_{r\times 4})_i$, $\sum \mathbf{O}_i$
DE	Double Efficient	$H_{\wedge\vee(b)}$
DENAR	Double Efficient Negentropic AND-OR	$-H_{\wedge\vee(b)}$, $-H_{\wedge\vee(fb)}$

[a] printed in bold to represent a field of data vectors as readable I/O dictionary code.
[b] printed in bold to represent a field of data vectors as I/O storable/compressed data.
[c] briefly discussed in this article, however, mainly considered for future articles covering issues related to VLDB compression, transactions and data management issues.

Table 5.3 Notations and acronyms in Table 5.2 as interpreted relevant to the current topic.

Acronym	As
LDC	an encoding phase, algorithm, code or program
LDD	a retrieving phase, algorithm, code or program
4D	a partitioned field, hypercube, vector or operator
FBAR	logic or algorithm
O	an input file or component when an LDC phase initiated
TT	a dictionary coder, addresses or reference code
G	an output file or component at the LDC phase
P	a source code, file or component
I/O	a data operation
ASCII	a standard table of codes
ISO	a responsible organization for management standards
CPU	the portion of a computer system that carries out the instructions of a computer program
VLDB	a database that contains a high number of tuples (database rows), which occupies large physical filesystem storage space, usually more than 1 terabyte = 10^12 bytes
RAM	a form of computer data storage
B	a unit of digital information mostly consists of eight bits
Bps	Byte-rate as the number of bytes that are conveyed or processed per unit of time, for evaluating algorithm spatial and temporal performance
b	a unit of digital information as a contraction of binary digit with a value of 0 or 1 logic
bps	Bit-rate as the number of bits that are conveyed or processed per unit of time, for evaluating algorithm spatial and temporal performance
KB	a multiple of the unit byte for digital information with a value of either $10^{24}(2^{10})$ bytes or $1000(10^3)$ bytes
MB	a multiple of the unit byte for digital information storage or transmission with two different values, such that 1048576 bytes (2^{20}) generally for computer memory
EB	a unit of information or computer storage equal to one quintillion bytes or 1 EB = 10^{18} bytes = 1073741824 gigabytes = 1048576 terabytes
DE	spatial inclusive of temporal double-efficiency, measured by FBAR entropy rate $H_{\wedge\vee(b)}$ with CPU time t_L, and Hamming rate R for proper compression
DENAR	a negative entropy (NE) measurement of information based on FBAR logic, measured in rate $-H$

References

[1] A. Dembo, T. M. Cover, and J. A. Thomas. Information theoretic inequalities. *IEEE Trans. Info. Theo.*, 37(6):1511–1517, 1991.

[2] P. B. Alipour and M. Ali. An introduction and evaluation of a fuzzy binary and/or compressor. Master's thesis, School of Computing, Blekinge Institute of Technology, Sweden, 2010.

[3] H. Anton. *Calculus: A New Horizon*. Wiley, New York, 6th edition, 1999.

[4] V. I. Arnold. Relatives of the quotient of the complex projective plane by the complex conjugation. *Tr. Mat. Inst. Steklova*, 224:56–67, 1999.

[5] V. K. Balakrishnan. *Theory and Problems of Combinatorics*. McGraw-Hill, New York, 1995.

[6] R. G. Bartle. *The Elements of Integration and Lebesgue Measure*. Wiley Interscience, New York, 1995.

[7] C. Bennett and S. J. Wiesner. Communication via one- and two -particle operators on einstein-podolsky-rosen states. *Phys. Rev. Lett.*, 69:2881–2884, 1992.

[8] W. Bergmans. *Maximum Compression.* software ranks published online in maximum compression benchmark. Phys. Rev. Lett., 2011.

[9] G. Boole. The calculus of logic. *Cambridge and Dublin Math. J.*, 3:183–98, 1848.

[10] J. P. Bowen. Hypercubes. *Pract. Comput. J.*, 5(4):97–99, 1982.

[11] S. F. Bush. *Nanoscale Communication Networks*, chapter 1.6, 1.6.2, pages 19–29. Artech House Publishers, USA, 2010.

[12] C. E. Shannon, A. D. Wyner, and N. J. A. Sloane. *Claude E. Shannon Collected Papers*. Wiley IEEE Press, New York.

[13] D. Chruściński. Geometric aspect of quantum mechanics and quantum entanglement. In *J. Phys. Conf. Ser.*, volume 35, pages 9–16, 2006.
[14] J. P. Cockle. On systems of algebra involving more than one imaginary. *Phil. Magaz.*, 35(3):434–435, 1849.
[15] A. W. Conway. On the application of quaternions to some recent developments in electrical theory. In *Royal Irish Acad. Proc.*, volume 29, pages 1–9, 1911.
[16] H. S. M. Coxeter. *The Coxeter Legacy: Reflections and Projections*. American Mathematical Society, USA, 2006.
[17] J. W. Dawson. The compactness of first-order logic: From gödel to lindström. *Histor. and Phil. of Logic*, 14(1):15–37, 1993.
[18] K. Gödel. *The first proof of the completeness theorem*. PhD thesis, University of Vienna, Austria, 1929.
[19] S. Gilheany. Moore's law and knowledge management. Archive Builders, 2011.
[20] P. R. Girard. The quaternion group and modern physics. *Europhys. J.*, 5:25–32, 1984.
[21] S. Gottwald. *A Treatise on Many-Valued Logics (Studies in Logic and Computation)*. Research Studies Press, Baldock, Hertfordshire, 2001.
[22] G. D. Green and D. Newth. Towards a theory of everything? - grand challenges in complexity and informatics. *Europhys. J.*, 8:1–12, 2001.
[23] W. R. Hamilton. *Lectures on Quaternions*, page 730. Dublin University Press, Utah, 1853.
[24] R. W. Hamming. Error detecting and error correcting codes. *Bell Sys. Tech. J.*, 29(2):147–160, 1950.
[25] A. Hyvärinen. *Measuring non-Gaussianity by negentropy. Independent component analysis*. Cambridge University Press, 2001.
[26] J. Jacas and L. Valverde. *On fuzzy relations, metrics and cluster analysis*, volume I. Approx. Reasoning Tools for AI, 1990.
[27] P. Jackson and D. Sheridan. Clause form conversions for boolean circuits. In *7th Int. Conf. on Theo. and Applic. of Satisfiability Testing, Springer*, pages 183–189, Vancouver, Canada, 2005.
[28] L. A. Zadeh, G. J. Klir, and B. Yuan. *Fuzzy Sets, Fuzzy Logic, Fuzzy Systems*. World Scientific Publishing Co., Singapore.
[29] C. Lanczos. *The Variational Principles of Mechanics*. Toronto Press University, Ontario, 4th edition.
[30] L. S. Leff. *PreCalculus the Easy Way*, page 296. Barron's Educational Series, Cambridge, 7th edition, 2005.
[31] R. Lidl and H. Niederreiter. *Finite Fields*. Cambridge University Press, 2nd edition, 1997.
[32] P. Lounesto. *Clifford algebras and spinors*. Cambridge University Press, 2001.
[33] M. Ebbers, W. O'Brien, and B. Ogden. *Introduction to the New Mainframe: z/OS Basics*, chapter 2.4.8 and 2.4.9, pages 15–19. Vervante, Utah, 2006.
[34] D. J. C. MacKay. *Information Theory, Inference, and Learning Algorithms*, pages 2, 15, 67–74, 91–100. Cambridge University Press, 7th edition, 2005.
[35] A. K. Maini. *Digital Electronics: Principles, Devices and Applications*. Wiley, West Sussex, 2007.

[36] K. Makarychev. A new class of non-shannon-type inequalities for entropies. *Commun. in Info. and Sys.*, 2:147–166, 2002.

[37] T. J. McCabe. A complexity measure. *IEEE Trans. Soft. Eng.*, 2(4):308–320, 1976.

[38] M. J. Murdocca and V. P. Heuring. *Information Theory, Inference, and Learning Algorithms*, pages 50–55, 364–366. Prentice Hall, New Jersey, 2000.

[39] A. Papoulis. *Probability, Random Variables, and Stochastic Processes.* McGraw-Hill, New York.

[40] K. M. Passino and S. Yurkovich. *Fuzzy Control*, chapter 1 and 2, pages 1–117. Addison Wesley Longman, Inc, New Jersey, 1998.

[41] K. Sayood, editor. *Lossless Compression Handbook*, chapter 1, 2, 6, pages 165–243. Academic Press, Elsevier Science, USA, 2003.

[42] C. T. Scarborough and A. H. Stone. Products of nearly compact spaces. *Trans. of Amer. Math. Soc.*, 124(3):131–147, 1966.

[43] C. E. Shannon. A symbolic analysis of relay and switching circuits. Master's thesis, Dept. of Elect. Eng, Massachusetts Inst. of Tech., USA, 1940.

[44] C. E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. J.*, 27:379–423, 623–656, 1948.

[45] S. K. Shukla. Transitive closure. Dictionary of Algorithms and Data Structures, 2004.

[46] N. J. A. Sloane. Bounds for binary codes of length less than 25. *IEEE Trans. Inf. Theo.*, 24:81–93, 1978.

[47] J. R. Smith. *Programming the PIC Microcontroller with Mbasic.* Newnes Publishers, Elsevier, Oxford.

[48] P. Symonds. Part 2: Hamming distance, math 32031: Coding theory. Lecture Paper, 2007.

[49] M. Tribus. *Thermodynamics and Thermostatics: An Introduction to Energy, Information and States of Matter.* Van Nostrand Company Inc., New York.

[50] A. H. Watson and T. J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* National Institute of Standards and Technology, USA.

[51] L. A. Zadeh. Fuzzy sets. *Info. and Control*, 8(3):338–353, 1965.

[52] E. N. Zalta. Terms: i- fuzzy logic, ii- logical consequence. Stanford Encyc. of Phil., 2009.

[53] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Info. Theo.*, 24(5):530–536, 1978.