SALZA: Practical Algorithmic Information Theory and Sharper Universal String Similarity

François Cayre, Marion Revolle, Isabelle Sivignon, Nicolas Le Bihan

To cite this version:

François Cayre, Marion Revolle, Isabelle Sivignon, Nicolas Le Bihan. SALZA: Practical Algorithmic Information Theory and Sharper Universal String Similarity. 2020. ⟨hal-03010529⟩

HAL Id: hal-03010529
https://hal.science/hal-03010529v1

Preprint submitted on 17 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

SALZA: Practical Algorithmic Information Theory and Sharper Universal String Similarity

François Cayre, Marion Revolle, Isabelle Sivignon and Nicolas Le Bihan

Abstract

Practical algorithmic information theory often relies on using off-the-shelf compressors, which both limits the range of target applications and introduces inaccuracies caused by the compressors implementation constraints and the fact that they are only able to handle multiple strings through concatenation.

We describe SALZA, a practical implementation of algorithmic information theory based on Lempel-Ziv routines, that is largely immune to the above issues. We focus on providing relationships that hold strictly in practice for strings of arbitrary length. We hope this work contributes to making clearer the links between Lempel-Ziv complexity, string similarity, and the use of compressors for computing information distances.

The capabilities of SALZA are highlighted by computing a universal semi-distance on strings, the NSD, and by performing causal inference using the PC algorithm, an application for which off-the-shelf compressors are hardly usable. SALZA was designed to take advantage of multi-core machines.

Index Terms

Algorithmic Information Theory, Causal Inference, String Similarity, Universal Classification.

I. Introduction

IN the discrete finite case, information theory in the probabilistic setting relies on the definition of a set $\mathcal{A} = \{a_k\}$ and a discrete random variable X taking values in $\mathcal{A}$ equipped with a probability mass function (p.m.f.) $p(a_k) = \Pr\{X = a_k\}$ to express the entropy H of X, the

An early version of this work appeared at IEEE ISIT'19, Paris, France.

Marion Revolle was supported by a French Ministry of Research PhD grant.

Authors are with the GAIA team at GIPSA-Lab, CNRS UMR 5216.

Emails: first.last@grenoble-inp.fr.

Corresponding author: François Cayre.

expected quantity of information that a string drawn from X conveys. Using base-2 logarithm and ignoring zero probability events, the Shannon entropy in bits is defined as:

$$H(X) = \mathbb{E}_p\left[-\log p(X)\right].$$

Defining another p.m.f. q on $\mathcal{A}$, the relative entropy, or Kullback-Leibler (KL) divergence between p and q is defined as:

$$D_{KL}(p\|q) = \mathbb{E}_p\left[\log\frac{p(X)}{q(X)}\right].$$

From there, one defines the mutual information $I(X;Y)$ between random variables X and Y on $\mathcal{A}$ as the KL-divergence between their joint p.m.f. and the product of X and Y p.m.f.'s (marginals). And the conditional entropy $H(X|Y)$ is defined as $H(X) - I(X;Y)$.

Yet, as powerful as it is, probabilistic information theory says nothing about a particular realization x of X. This was the motivation for algorithmic information theory, independently developed by Solomonoff, Kolmogorov and Chaitin. Given a universal computer $\mathcal{U}$ accepting programs c of length $|c|$, the Kolmogorov (prefix[1]) complexity $K(x)$ of a string x is defined as:

$$K(x) = \min_{c:\,\mathcal{U}(c)=x} |c|.$$

One similarly defines the conditional Kolmogorov complexity $K(x|y)$ of a string x knowing another string y and eventually obtains the same kind of information calculus as in the probabilistic setting, albeit generally up to some additive constants.

With some simplification in the notations and assuming several i.i.d. $x^n \sim \prod_{i=1}^{n} p(a_i)$, then (Theorem 7.3.1 in [2]):

$$\mathbb{E}\left[\frac{1}{n}K(X^n)\right] \to H(X).$$

This relationship between Kolmogorov complexity and Shannon entropy shows the strong connection they share. Yet, the Kolmogorov complexity is uncomputable on a universal Turing computer and its use, following Solomonoff's advice, should be that of a golden standard against which practical solutions should be evaluated. In the same research line, the Normalized Information Distance (NID) has been proposed [3], that serves as an ultimate distance between

[1]The Kolmogorov prefix complexity has become the standard version of algorithmic complexity (see [1], Chap. 3) because, relying on a Turing machine accepting programs that are not the prefix of others, it is able to circumvent issues that the original, plain Kolmogorov complexity did not consider at the time (subadditivity and non-monotonicity over prefixes), see *infra*.

strings:

$$\mathrm{NID}(x,y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}. \tag{1}$$

One of the most influential proposal for a computable complexity of an individual sequence was proposed by Lempel and Ziv [4]. This work soon after gave birth to the popular LZ family of source coding algorithms [5], [6], leading to optimal compressors for a range of source distributions. In turn, those compressors would eventually be used as practical proxies for the approximation of Kolmogorov complexity. In [7], Ziv and Merhav used the cross-parsing proposed in [8] to derive an estimate of the KL-divergence between p.m.f.'s of which only one realization is known.

Using an off-the-shelf compressor C, the Normalized Compression Distance [9], [10] is a practical embodiment of the NID that computes a universal distance between sequences x and y. If xy is the concatenation of x and y and $|x|$ is the length of x, then:

$$\mathrm{NCD}(x,y) = \frac{|C(xy)| - \min\{|C(x)|, |C(y)|\}}{\max\{|C(x)|, |C(y)|\}}. \tag{2}$$

The expression of the NCD implicitly assumes $|C(xy)| = |C(yx)|$, which is an approximation that depends on the particular compressor used in practice. In effect, a compressor has to deal with a few constraints, and being aware of them may help selecting the right compressor for the data at hand [11].

But another limitation in using compressors stems from the way they enforce the concatenation-based approximation to compute an estimate of conditional information $C(y|x) \approx C(xy) - C(x)$: while this is both practical and theoretically satisfying when two strings are involved in the computation, approximations can only get worse when more strings are involved — for example, performing causal inference would require to compute something like $C(y|x_1,\ldots,x_n)$ efficiently and reliably, for a number of varying $y, x_1, \ldots, x_n$. It is not clear how one could guarantee to do that in practice. At the very least, this would certainly imply managing concatenations in a somewhat cumbersome and inefficient fashion.

We argue that using off-the-shelf compressors hinders what can be actually implemented to leverage the power of algorithmic information estimates for applications where conditioning must be computed with much greater flexibility. Although using algorithmic information has been justified for causal inference [12], practical realizations on this side have been scarce.

Hence, we believe that an *implementation* of algorithmic information theory [13] is both possible and needed, ensuring that *(i)* the required flexibility in conditioning is easily achieved and *(ii)* the computed quantities verify usual information-theoretic relationships for strings of arbitrary length.

A. Notations

The empty string, set or multiset is denoted by $\emptyset$. The set of non-empty finite strings on alphabet $\mathcal{A}$ is denoted by $\mathcal{A}^+$ and $\mathcal{A}^\star = \mathcal{A}^+ \cup \emptyset$. The length of alphabets, strings, sets and multisets are denoted with $|.|$. The symbol $+$ is also used to denote addition of multisets. The first k strings in a set are denoted with $x_{\leq k}$ and by convention $x_{\leq 0} = \emptyset$. The concatenation of two strings x and y is denoted with xy. Logarithms are in base 2. Terms related to implementation or simulations are typeset using `monospaced` font.

B. Lempel-Ziv symbols, between theory and practice

We shall make extensive use of Lempel-Ziv-type factorization [5] and cross-factorization [8]. Hence, we start by making clear some basic implementation choices.

We first remark that popular compressors from the Lempel-Ziv family (most notably `gzip` [14], [15] and `lzma` [16]) nowadays are based on LZ77 [5] instead of LZ78 [6]. The reasons for this are that computer memory has become cheaper and it is easier to manage a sliding window (from 32KiB in [14] to 1MiB by default and maximum 4GiB in [16]) than an evolving dictionary of substrings not seen so far.

We will also use algorithms similar to those based on a sliding window, except that its size will be in practice limited by the host memory (pointers are 32-bit wide in SALZA).

Further, since the inception of the Lempel-Ziv source coding techniques [5], [6], [8], where a symbol is composed of a reference to the next substring and the next following character, most expositions now use only references or characters — with characters being either described as length-1 references (most notably in the computer science literature, see [17] and related works) or transmitted as such (as in off-the-shelf compressors, *e.g.* as in [14]). The same discrepancy also appears between the original exposition of [8] and some of its later uses [7], [18].

In what follows, this distinction will be irrelevant, to the exception of Sec. II-D.

II. SALZA AS A MEASURE OF STRING SIMILARITY

The first step in this work is to explain how a generic factorization routine relates to string similarity and to previous works [4] on computable complexity. Also, we shall discuss why compressors may not always be the best tools for similarity assessment.

A. SALZA *factorizations*

SALZA is built around a routine for sequential factorization (parsing) of a string y when given as prior knowledge a set of strings $x_1,\ldots,x_n$. Following the Lempel-Ziv basic idea, SALZA will look for the next longest substring that matches the beginning of the remaining part of y to be factorized. Each string is associated to its own search structure (see Appendix A-B for details) so that the next longest substring cannot overlap between strings.

Two kinds of factorization were implemented, depending on whether the part of y that was already factorized (the prefix of y) is included in the search space or not.

The first kind of factorization will be denoted with $y|x_1,\ldots,x_n$: at each factorization step, the next longest substring is to be found either in one of the $x_1,\ldots,x_n$, or in the prefix of y.

The second kind of factorization will be denoted with $y|^{+}x_1,\ldots,x_n$: at each factorization step, the next longest substring is to be found only in one of the $x_1,\ldots,x_n$. This can be seen as a generalization of the procedure described in [8] allowing for several "databases".

When the kind of factorization is left unspecified, it will be denoted with $y\wr x_1,\ldots,x_n$.

Hence, at each factorization step, we look for the longest substring in all prior knowledge strings $x_1,\ldots,x_n$ (also including the prefix of y for the factorization $|$) and we keep the longest of them as the next factor.

*Definition 2.1 (*SALZA *symbols and lengths):* Given strings $y,x_1,\ldots,x_n$ in $\mathcal{A}^{\star}$, SALZA will factorize y into m symbols:

$$y\wr x_1,\ldots,x_n = (s_1,l_1,z_1)\ldots(s_k,l_k,z_k)\ldots(s_m,l_m,z_m),$$

where $s_k \in \{y,x_1,\ldots,x_n\}$ is (a pointer to) the string in which the next longest substring was found, $1 \le l_k \le |s_k|$ is the length of the next longest substring, and $1 \le z_k \le |s_k|$ is a pointer in string s_k where the next longest substring starts.

All SALZA symbol lengths are collected in the multiset $\mathcal{L}_{y\wr x_1,\ldots,x_n} = \{l_k\}$, $1 \le k \le m$.

Remark 1 (Invariance of $y \wr x_1, \ldots, x_n$): $y \wr x_1, \ldots, x_n$ is invariant with respect to the order of the prior knowledge strings $x_1, \ldots, x_n$ because the sequential factorization of y only evolves after all strings have been searched for the longest substring.

Remark 2 (Integer composition of $|y|$): $\mathcal{L}_{y \wr x_1, \ldots, x_n}$ is an integer composition of $|y|$ because $\sum_{l \in \mathcal{L}_{y \wr x_1, \ldots, x_n}} l = |y|$.

Proposition 1 ($|\mathcal{L}_{y \wr x_1, \ldots, x_n}|$ is non-increasing by conditioning): For any three strings $x, y, z \in \mathcal{A}^\star$, $|\mathcal{L}_{y \wr x, z}| \leq |\mathcal{L}_{y \wr x}|$.

Proof: If $y = \emptyset$, $\mathcal{L}_{y \wr x} = \mathcal{L}_{y \wr x, z} = \emptyset$. The following assumes $y \in \mathcal{A}^+$. Let $\mathcal{L}_{y \wr x}^{\leq i} = \left\{ l_{y \wr x}^1, \ldots, l_{y \wr x}^i \right\}$ be the first i lengths in the factorization $y \wr x$. To each i, we associate $v(i)$ the smallest integer such that $\sum_{k=1}^{i} l_{y \wr x}^k \leq \sum_{k=1}^{v(i)} l_{y \wr x, z}^k$. We shall show that $\forall i$, $v(i) \leq i$ (which proves the claim by setting $i = |\mathcal{L}_{y \wr x}|$). The proof is inductive:

For the base case ($i = 1$), $v(1) = 1$ because z could only bring a longer reference (so that $l_{y \wr x}^1 \leq l_{y \wr x, z}^1$). Hence, $v(1) = 1$;

For the inductive step, we shall show that: $v(i) \leq i \implies v(i+1) \leq i+1$.

By definition of $v(i)$, only two cases occur:

- (Case 1) $\sum_{k=1}^{i} l_{y \wr x}^k = \sum_{k=1}^{v(i)} l_{y \wr x, z}^k$. In this case, the two factorizations are in sync and the exact next same longest substring is to be searched with or without knowledge of z. This next substring may only be longer in z so $v(i+1) = v(i) + 1$, and (by the premise):

$$v(i+1) = v(i) + 1 \leq i + 1;$$

- (Case 2) $\sum_{k=1}^{i} l_{y \wr x}^k < \sum_{k=1}^{v(i)} l_{y \wr x, z}^k$. In this case, the factorization $y \wr x, z$ is in advance over $y \wr x$, and only two cases occur depending whether step $i+1$ in $y \wr x$ will overtake step $v(i)$ of $y \wr x, z$ or not:

  (Case 2.1) $\sum_{k=1}^{i+1} l_{y \wr x}^k \leq \sum_{k=1}^{v(i)} l_{y \wr x, z}^k$. Then $v(i+1) = v(i)$ and (also by the premise):

$$v(i+1) = v(i) \leq i < i + 1;$$

  (Case 2.2) $\sum_{k=1}^{i+1} l_{y \wr x}^k > \sum_{k=1}^{v(i)} l_{y \wr x, z}^k$. In the worst case, the factorization step $v(i) + 1$ will see $y \wr x$ back in sync. Otherwise, $y \wr x, z$ keeps ahead. Hence, $l_{y \wr x, z}^{v(i)+1} \geq \sum_{n}^{i+1} l_{y \wr x}^n - \sum_{n}^{v(i)} l_{y \wr x, z}^n$ and $v(i+1) = v(i) + 1$, which finally shows the claim similarly to Case 1 above.

■

Definition 2.2 (Product of SALZA *factorizations):* Let y_1 and y_2 two strings being factorized, each with respective prior knowledge strings $x_{1,1},\dots,x_{1,m_1}$ and $x_{2,1},\dots,x_{2,m_2}$. Let also:

$$y_1 \wr x_{1,1},\dots,x_{1,n_1} = (s_{1,1},l_{1,1},z_{1,1})\dots(s_{1,m_1},l_{1,m_1},z_{1,m_1}), \quad \text{and}$$
$$y_2 \wr x_{2,1},\dots,x_{2,n_2} = (s_{2,1},l_{2,1},z_{2,1})\dots(s_{2,m_2},l_{2,m_2},z_{2,m_2}).$$

We define their factorization product as the concatenation of their SALZA symbols:

$$y_1 \wr x_{1,1},\dots,x_{1,n_1} \;\times\; y_2 \wr x_{2,1},\dots,x_{2,n_2} =$$
$$(s_{1,1},l_{1,1},z_{1,1})\dots(s_{1,m_1},l_{1,m_1},z_{1,m_1})\,(s_{2,1},l_{2,1},z_{2,1})$$
$$\dots(s_{2,m_2},l_{2,m_2},z_{2,m_2}).$$

*Definition 2.3 (*SALZA *joint factorization):* The joint factorization of $x_1,\dots,x_n \in \mathcal{A}^\star$ is defined as the following product of factorizations:

$$x_1 \cdot \ldots \cdot x_n = \prod_{i=1}^{n} x_i \wr x_{\le i-1}.$$

Hence, $x|\emptyset$ may therefore be used to denote the usual LZ77 factorization of x and $|x| = |\mathcal{L}_{x|^{+}\emptyset}|$.

Remark 3 (On concatenation and symmetry): Because the factorization is a sequential operation:

1) $xy|\emptyset = x|\emptyset \times y|x$ and therefore $\mathcal{L}_{xy|\emptyset} = \mathcal{L}_{x|\emptyset} + \mathcal{L}_{y|x}$;
2) $xy|^{+}\emptyset = x|^{+}\emptyset \times y|^{+}\emptyset$ and therefore $\mathcal{L}_{xy|^{+}\emptyset} = \mathcal{L}_{x|^{+}\emptyset} + \mathcal{L}_{y|^{+}\emptyset}$;
3) $x \cdot y \neq y \cdot x$ in general.

B. Relative string similarity

If the multiset $\mathcal{L}_{y \wr x_1,\dots,x_n}$ is seen as an integer composition of $|y|$, we can associate it to the integer *partition* of $|y|$ obtained by sorting the lengths in $\mathcal{L}_{y \wr x_1,\dots,x_n}$ in decreasing order. In the following, we assume $\mathcal{L}_{y \wr x_1,\dots,x_n}$ is sorted in decreasing order and it is therefore a partition of $|y|$.

Definition 2.4 (Relative similarity measure): Let $[|y|]$ the set of partitions of integer $|y|$ and $\mathcal{L}_{y \wr x_1,\dots,x_n} \in [|y|]$. A relative similarity measure between y and prior knowledge strings $x_1,\dots,x_n$ is defined as a score function $S : [|y|] \to \mathbb{R}^+ \cup \{0\}$.

When the context makes it clear, we either write $S(\mathcal{L})$ or $S(y \wr x_1,\dots,x_n)$ instead of $S(\mathcal{L}_{y \wr x_1,\dots,x_n})$.

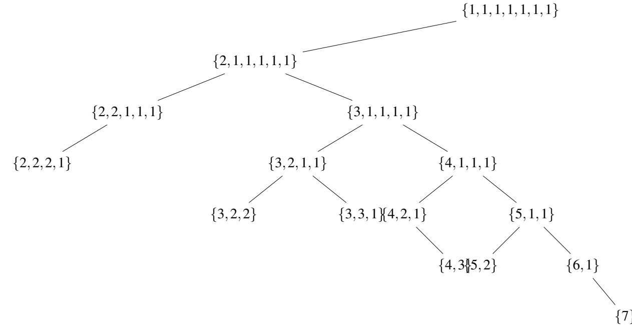


Fig. 1: Binary tree of the integer partitions of 7. To each level corresponds the same number of summands. Whenever possible, growing downwards is done either by replacing two 1's with one 2 (left child), or by removing a 1 and increasing the smallest summand greater than 1 if it has multiplicity 1 (right child).

Examples of score functions S include counting the elements of the multiset and the length of the compressed string y according to the factors $y \wr x_1, \ldots, x_n$. We are interested in finding a score function that is as much precise as possible, even for short strings, so as to cover the full range of possible applications.

Even though no closed-form expression exists for the number of partitions of an integer[2], they can be constructed along a binary tree [19], see Fig. 1, that can be used to induce an order of interest (Proposition 2 and Table I). Such a binary tree is called the *partition tree* of the corresponding integer.

We slightly adapt notations from [19] to our setting. A partition of a positive integer $|y|$ is a m-tuple $\mathcal{L} = \{l_1, l_2, \ldots, l_m\}$ of positive integers (the summands of the partition) such that:

$$l_1 + l_2 + \cdots + l_m = |y|,$$

$$l_1 \geq l_2 \geq \cdots \geq l_m \geq 1.$$

When it exists, the number of summands strictly greater than 1 is denoted by h, $1 \leq h \leq m$, *i.e.*:

$$\mathcal{L} = \{l_1, \ldots, l_h > 1, l_{h+1} = 1, \ldots, l_m\}.$$

[2]The partition function $P(n)$ gives the number of partitions of an integer n. In 1918, Hardy and Ramanujan proposed the following asymptotic approximation:

$$P(n) \approx \frac{1}{4n\sqrt{3}} \exp\left(\pi\sqrt{\frac{2n}{3}}\right) \text{ as } n \to \infty.$$

$\{1,1,1,1,1,1,1\}$
$\{2,1,1,1,1,1\}$
$\{2,2,1,1,1\}$
$\{3,1,1,1,1\}$
$\{2,2,2,1\}$
$\{3,2,1,1\}$
$\{4,1,1,1\}$
$\{3,2,2\}$
$\{3,3,1\}$
$\{4,2,1\}$
$\{5,1,1\}$
$\{4,3\}$
$\{5,2\}$
$\{6,1\}$
$\{7\}$

TABLE I: List of integer partitions of 7 obtained when traversing the tree of Fig. 1 in breadth-first order.

Given a partition $\mathcal{L}$, one can define two operations acting on $\mathcal{L}$, namely A and B, such as:

- A: Add one summand of value 2 and remove two of value 1;
- B: If the smallest summand strictly greater than 1 has multiplicity 1 ($l_{h-1} > l_h$), increase it by 1 and remove a summand of value 1.

When operation A or B can[3] be applied to a partition $\mathcal{L}$, the resulting partition is denoted by $\mathcal{L}A$ or $\mathcal{L}B$. The binary tree enumerating all integer partitions of n is constructed starting from the root $\perp = \{l_1 = 1, \ldots, l_m\}$ by applying A (*resp.* B) to get the left (*resp.* right) child of a partition whenever possible.

In the sequel, necessary conditions are assumed to hold true when the result of an operation on a partition is stated.

In [19], Corollary 13 states that if the nodes of the partition tree are listed in top-down left-to-right level (breadth-first) order, the resulting list is in decreasing order of the number of summands and in lexicographic order for partitions having the same number of summands (see Table I).

Proposition 2 (Score compliant with breadth-first order): Let $[|y|]$ the set of integer partitions of $|y|$, $\perp = \{1, \ldots, 1\}$ ($|\perp| = |y|$) and $S : [|y|] \to \mathbb{R}^+ \cup \{0\}$ a score function that verifies the

[3]The number of partitions of an integer is finite, therefore operation A or B cannot always be applied. This is made explicitly clear in the definition of operation B, but it is more implicit in that of operation A (which requires two summands of value 1 to be applied). Thus, the partition tree is not a full binary tree, see Fig. 1.

following conditions:

1) $\forall\ 0 \le i \le |y|$, $S(\bot B^i) \ge S(\bot A^{i+1})$;
2) $\forall\ 0 \le i \le |y|$, $S(\mathcal{L}AB^i) \ge S(\mathcal{L}BA^i)$.

Then S complies with the breadth-first order.

Proof: Consider the binary alphabet $\mathcal{B} = \{A, B\}$. We label any node x of the partition tree by the word on $\mathcal{B}^\star$ composed of the sequence of operations performed from the root $\bot$ to x and denote it by label(x). By convention, label$(\bot) = \emptyset$. Then, following the notations introduced before, the integer partition of $|y|$ represented by the node x is denoted by $r(x) = \bot\text{label}(x)$.

Consider the ordered sequence $\{x_k\}_k$ of nodes given by a breadth-first traversal of the integer partition tree of $|y|$. If $A < B$, then the nodes are ordered according to the radix order[4] (also called shortlex order) of their labels (see [20], Vol. 1, solution of Exercise 2.3-(15) p. 562 and Sec. 2.3.3 p. 351). Assuming Conditions (1) and (2) and given two consecutive nodes x_k and x_{k+1}, we have to show that $S(r(x_k)) \ge S(r(x_{k+1}))$. Two cases occur:

1) if x_k and x_{k+1} are not on the same level of the tree, then there exist $0 \le i \le |y|$ such that label$(x_k) = B^i$ and label$(x_{k+1}) = A^{i+1}$. We have $S(r(x_k)) = S(\bot B^i)$ and $S(r(x_{k+1})) = S(\bot A^{i+1})$, and the result follows from Condition (1);
2) otherwise x_k and x_{k+1} are on the same tree level and there exists $0 \le i \le |y|$ and a word M on $\mathcal{B}^\star$ such that label$(x_k) = MAB^i$ and label$(x_{k+1}) = MBA^i$. $S(r(x_k)) = S(\bot MAB^i)$ and $S(r(x_{k+1})) = S(\bot MBA^i)$, and the result follows from Condition (2) with $\mathcal{L} = \bot M$.

■

C. Order compliant score functions

One immediately verifies that $S_{\mathbb{1}}(\mathcal{L}) = |\mathcal{L}| - 1$ complies with the breadth-first order. However, it is unable to make a difference between partitions located on the same tree level.

*Definition 2.5 (*SALZA *score function S_p):* The SALZA score function of a partition $\mathcal{L} = \mathcal{L}_{y \wr x_1,\ldots,x_n} = \{l_1, \ldots, l_m\}$ is defined as:

$$S_p(\mathcal{L}) = \begin{cases} 0 & \text{if } \mathcal{L} = \emptyset \\ |\mathcal{L}| - \frac{1}{|y|+1} - \sum_{k=1}^{p} (|y|+1)^{-k} l_k & \text{otherwise,} \end{cases} \tag{3}$$

where $p \ge 1$ is a parameter allowing to tune the precision and $l_k = 0$ if $k > m$.

[4]The radix order is defined by $x < y$ if $(|x| < |y|)$ or $(|x| = |y|$ and $x = uAx'$ and $y = uBy')$.

In other words, the fractional part of S_p is built by writing the l_k's in radix $(|y|+1)$ (if not for the corrective term $1/(|y|+1)$, see below). And S_p may be seen as an arbitrary-precision version of $S_{\mathbb{1}}$. The value of p is set according to Appendix A-A.

*Proposition 3 (*SALZA *score is compliant with breadth-first order):*

1) $\forall\ 0 \le i < |y|,\ S_p(\perp B^i) \ge S_p(\perp A^{i+1})$;
2) $\forall\ 0 \le i < |y|,\ S_p(\mathcal{L}AB^i) \ge S_p(\mathcal{L}BA^i)$.

Proof: Let $\mathcal{L} = \{l_1, \ldots, l_h > 1, l_{h+1} = 1, \ldots, l_m\}$ a partition of $|y|$. In the sequel, we shall repeatedly apply operations A and/or B to a partition, leading to different values of h (and l_h) for the resulting partitions. However, we want to express what happens to $\mathcal{L}$ after operations A and/or B have been applied, so we require that h and l_h refer to that of $\mathcal{L}$ — and the actual h and l_h of the resulting partitions can be easily deduced from the following expressions:

$$
\begin{aligned}
\perp A^i &= \left\{l_1 = 2, \ldots, l_i = 2, l_{i+1} = 1, \ldots, l_{|y|-i}\right\}, \\
\perp B^i &= \left\{l_1 = 1+i, l_2 = 1, \ldots, l_{|y|-i}\right\}, \\
\mathcal{L}AB^i &= \{l_1, \ldots, l_h, l_{h+1} = 2+i, l_{h+2} = 1, \ldots, l_{m-i-1}\}, \\
\mathcal{L}BA^i &= \{l_1, \ldots, l_h + 1, l_{h+1} = 2, \ldots, l_{h+i} = 2, \\
&\qquad l_{h+i+1} = 1, \ldots, l_{m-i-1}\}.
\end{aligned}
$$

Let $\Delta_1(i) = S_p(\perp B^i) - S_p(\perp A^{i+1})$ and $p = |y| - i$:

$$
\Delta_1(i) = 1 - \frac{i-1}{|y|+1} + \sum_{k=2}^{i+1} (|y|+1)^{-k} - (|y|+1)^{-(|y|-i)}.
$$

The expression of $\Delta_1(i)$ is written such that decreasing p removes terms starting from the right. For instance, when $p < |y| < i$, the last term $(|y|+1)^{|y|-i}$ disappears. Lowest values of $\Delta_1(i)$ are obtained for $p = |y| - i$ or $p = 1$ (in which case all positive terms of the sum disappear.)

For $p = |y| - i$, since $0 \le i < |y|$, $(|y|+1)^{-(|y|-i)} \le (|y|+1)^{-1}$, so that:

$$
\frac{i-1}{|y|+1} + \frac{1}{(|y|+1)^{|y|-i}} \le \frac{i}{|y|+1} < 1.
$$

For $p = 1$, using the fact that $i < |y|$, we get $i - 1 < |y| + 1$ and $1 - (i-1)/(|y|+1) > 0$. Hence:

$$
\forall\ |y| \ge 0, \forall\ p \ge 1, \forall\ 0 \le i < |y|, \Delta_1(i) \ge 0.
$$

Let $\Delta_2(i) = S_p(\mathcal{L}AB^i) - S_p(\mathcal{L}BA^i)$ and $p = m - i$:

$$\Delta_2(i) = (|y|+1)^{-h} - i(|y|+1)^{-(h+1)} + \sum_{k=h+2}^{h+i} (|y|+1)^{-k}.$$

One has that:

$$\frac{1}{(|y|+1)^h} - \frac{i}{(|y|+1)^{h+1}} = \frac{|y|+1-i}{(|y|+1)^{h+1}}.$$

Since $i < |y|$:

$$\forall\ |y| \geq 0, \forall\ p \geq 1, \forall\ 0 \leq i < m \leq |y|, \Delta_2(i) \geq 0.$$

■

Definition 2.5 provides an insight into the way SALZA implements string similarity: strings y and $x_1, \ldots, x_n$ will be all the more similar when *(i)* the associated factorization is shorter and *(ii)* when it contains more longer factors.

When the set of prior knowledge strings is empty, two cases arise depending on the kind of factorization used:

- Case $y|^{+}\emptyset$: no substring can ever be found, so we end up enumerating the characters of y and $S_p(y|^{+}\emptyset)$ will merely reflect the length of y;
- Case $y|\emptyset$: substrings can only be found in the part of y that has already been parsed, and $S_p(y|\emptyset)$ can therefore be seen as a measure of self-similarity.

Compared to [4] and [7] where one *counts* the number of factors in a factorization (thus amounting to selecting a level on the partition tree with $S_{\mathbb{1}}$), SALZA will provide a more fine-grained measure because it is able to make a difference between partitions (and associated factorizations) located on the same tree level. Further, this difference is associated to a rather intuitive measure of similarity based on the length of the factors — which is backed by empirical evidence elsewhere [21], that longer matches help in building more discriminative features for string similarity (the Noisy Stemming Hypothesis in [21]).

Proposition 4 ($S_p(y \wr x_1, \ldots, x_n)$ is positive and bounded):

$$0 \leq S_p(y \wr x_1, \ldots, x_n) < |y|. \tag{4}$$

Proof: When $y = \emptyset$, $\mathcal{L}_{y \wr x_1,\ldots,x_n} = \emptyset$ and $S_p(y \wr x1, \ldots, x_n) = 0$.

When $|y| > 0$, Proposition 3 states that the minimum is reached when $|\mathcal{L}_{y \wr x_1,\ldots,x_n}| = 1$ (y is

a substring of any of the $x_1,\ldots,x_n$), and the maximum is reached when $|\mathcal{L}_{y\wr x_1,\ldots,x_n}| = |y|$ (no substring of any of the $x_1,\ldots,x_n$ matches any substring of y — this is the root of the partition tree).

Hence, for $|y| > 1$, the minimum is:

$$1 - \frac{1}{|y|+1} - \frac{|y|}{|y|+1} = 0 \leq S_p(y \wr x_1,\ldots,x_n),$$

and the maximum is:

$$S_p(y \wr x_1,\ldots,x_n) \leq |y| - \frac{1}{|y|+1} - \sum_{k=1}^{p} (|y|+1)^{-k} < |y|.$$

■

Remark 4: For strings y of size 1, both $S_{\mathbb{1}}$ and S_p equal zero.

D. Comparing with compression

Most practical compressors may be seen as two-stage pieces of software engineering. The first stage will extract some regularity information from the string (LZ coding, Burrows-Wheeler transform (BWT) [22]), while the second stage will use entropy coding in order to converge faster towards entropy (Huffman [23] or arithmetic coding, possibly intertwined with Move-to-Front strategy). The intricacies between these two stages make modeling of the performances of real compressors a tedious task (most notably, LZ77-based compressors will further quantize reference lengths and distances prior to encoding them). In the literature, tractable expressions are obtained by implementing the second stage with universal coding of integers [24].

Here, we use an estimation of the compressed string length (*à la* LZ77) for a score function on an integer partition of $|y|$. For actual compression, of course, we could not encode the factors sorted by decreasing length, but the estimated compressed string length would be the same. Since real LZ77-based compressors tend to transmit characters as such (and not as length-1 references), we shall spend $\log|\mathcal{A}|$ bits for each of them. Let $d = \max\{|y|, |x_1|, \ldots, |x_n|\}$. Using δ-codes [24] both for the length and offset of a reference, we shall spend $\log l_k + 2\log(1+\log l_k) + \log d + 2\log(1+\log d)$ bits for a reference[5].

[5]The z_k's in Def. 2.1 start at 1, so we do not extend the δ-code for offsets in s_k's to handle zeros.

In this setting, the compressed string length based on the $\mathcal{L} = \mathcal{L}_{y \wr x_1,\ldots,x_n}$ partition of $|y|$ reads:

$$S_c(\mathcal{L}) = \sum_{k=1}^{h} \log\left(c(1+\log l_k)^2 l_k\right) + \sum_{k=h+1}^{|L|} \log|\mathcal{A}|,$$

where $c = \log\left(d(1+\log d)^2\right)$. Let now $\Delta_{c,1}(i) = S_c(\mathcal{L}B^i) - S_c(\mathcal{L}A^{i+1})$:

$$\begin{aligned}
\Delta_{c,1}(i) = {} & \log(c(l_h+i)(1+\log(l_h+i))^2) + \sum_{k=h+1}^{h+i+1} \log|\mathcal{A}| \\
& + \sum_{k=h+i+2}^{m-i-1} \log|\mathcal{A}| + \log|\mathcal{A}| - \log(c l_h(1+\log l_h)^2) \\
& - \sum_{k=h+1}^{h+i+1} \log 8c - \sum_{k=h+i+2}^{m-i-1} \log|\mathcal{A}| \\
\Delta_{c,1}(i) = {} & \log\left(|\mathcal{A}| \frac{(l_h+i)(1+\log(l_h+i))^2}{l_h(1+\log l_h)^2}\right) \\
& + (i+1)\log\left(\frac{|\mathcal{A}|}{8c}\right).
\end{aligned}$$

Let $\Delta_{c,2}(i) = S_c(\mathcal{L}AB^i) - S_c(\mathcal{L}BA^i)$. Similarly:

$$\begin{aligned}
\Delta_{c,2}(i) = {} & \log\left(\frac{l_h(1+\log l_h)^2}{(l_h+1)(1+\log(l_h+1))^2}\right) \\
& + \log\left(\frac{(2+i)(1+\log(2+i))^2}{8}\right) + (i-1)\log\left(\frac{|\mathcal{A}|}{8c}\right).
\end{aligned}$$

Neither $\Delta_{c,1}$ nor $\Delta_{c,2}$ is positive for all values of $i \geq 0$ and $|y| > 1$, only when i is large enough — meaning that compressed string length will be a meaningful score only when the differences between the two factorizations will be large enough (the partitions corresponding to the two factorizations are on two sufficiently distant levels of the partition tree). Even using a sliding window of size $w \leq d$ would only marginally help (replacing d by w in c, although the actual coding scheme would still have to be defined).

The above should be mitigated given the following two observations:

1) A compressor is not designed to specifically comply with the two conditions of Proposition 2 — but even if this comparison may seem not so fair, it nevertheless suggests that compressors are likely to be less precise tools for assessing similarity in general;
2) Real compressors use several tricks that we are unable to model but that allow to converge faster towards entropy (they are, however, unable to handle conditioning with as much

flexibility as SALZA).

III. SALZA AS AN IMPLEMENTATION OF INFORMATION THEORY

Our notion of string similarity is not defined in the asymptotic regime, which is how performances of compressors are evaluated. Now, even if information theory is primarily concerned with very long strings, we are nevertheless in position to list various information-theoretic properties for SALZA that hold strictly in practice for arbitrary string lengths.

The biggest departure from the probabilistic setting is that, because factorizations are *sequential* processes, we will be limited to *asymmetric* relationships. For a number of relevant applications, however, this is not an issue (see Sec. IV and V).

We shall use the subscript f in the notation S_f to denote either $S_{\mathbb{1}}$ or S_p, since both were shown to comply with our notion of string similarity. When it is omitted, we assume $S_f = S_p$.

A. Basic tools and conditioning

*Definition 3.1 (*SALZA *joint and self measures):* Given strings $x_1,\ldots,x_n \in \mathcal{A}^*$ and following Def. 2.3, the SALZA joint measure is defined as:

$$S_f(x_1 \cdot \ldots \cdot x_n) = \sum_{i=1}^{n} S_f(x_i \wr x_{\leq i-1}).$$

Remark 5 (On joint and self measures):

1) By Def. 2.4, the order of the $x_1,\ldots,x_n$ does matter;
2) When applied to a single string, the notation for joint measure may gracefully degrade into that of the LZ77-based computation of a measure on self (factorization $|$). Further, in the case of a single string, the context will help determine which kind of factorization is meant.

*Definition 3.2 (*SALZA *conditional mutual measure, asymmetric version):* Given strings $x,y,z \in \mathcal{A}^*$, the SALZA conditional mutual measure of x and y given z is defined as:

$$I_f(x : y \wr z) = S_f(z \cdot x) + S_f(z \cdot y) - S_f(z \cdot x \cdot y) - S_f(z). \quad (5)$$

If $I_f(x : y \wr z) = 0$, x and y are said to be independent given z.

B. Properties

Proposition 5 ($S_f(y \wr x_1,\ldots,x_n)$ is non-increasing by conditioning):
Let $y \in \mathcal{A}^\star$ and $\{x_1,\ldots,x_{n+1}\}$ a set of $n+1$ strings in $\mathcal{A}^\star$.

$$\forall\, n \geq 0,\; S_f(y \wr x_{\leq n+1}) \leq S_f(y \wr x_{\leq n}). \tag{6}$$

Proof: Proposition 1 only allows three cases:

- (Case 1) $|\mathcal{L}_{y\wr x_{\leq n+1}}| < |\mathcal{L}_{y\wr x_{\leq n}}|$, in which case the integer partition associated to $\mathcal{L}_{y\wr x_{\leq n+1}}$ is located on a strictly lower level of the partition tree of $|y|$ than that associated to $\mathcal{L}_{y\wr x_{\leq n}}$ and Proposition 3 ensures that $S_f(y \wr x_{\leq n+1}) < S_f(y \wr x_{\leq n})$;
- $|\mathcal{L}_{y\wr x_{\leq n+1}}| = |\mathcal{L}_{y\wr x_{\leq n}}|$ and the integer partition associated to $\mathcal{L}_{y\wr x_{\leq n+1}}$ is
  - (Case 2.1) the same as that associated to $\mathcal{L}_{y\wr x_{\leq n}}$ (string x_{n+1} did not bring new knowledge on y) and $S_f(y \wr x_{\leq n+1}) = S_f(y \wr x_{\leq n})$;
  - (Case 2.2) in strictly increasing lexicographic order than that associated to $\mathcal{L}_{y\wr x_{\leq n}}$ (because at least one longer substring was found during the factorization $y \wr x_{\leq n+1}$ compared to $y \wr x_{\leq n}$) and Proposition 3 ensures that $S_f(y \wr x_{\leq n+1}) \leq S_f(y \wr x_{\leq n})$.

Setting $n = 0$ shows that the maximum is reached for $S_f(y \wr \emptyset) = S_f(y)$. ■

Proposition 6 (S_f is subadditive): Given strings $x, y \in \mathcal{A}^\star$,

$$S_f(x \cdot y) \leq S_f(x) + S_f(y);$$
$$S_f(y \cdot x) \leq S_f(x) + S_f(y).$$

Proof: By Def. 3.1:

$$\begin{aligned} S_f(x \cdot y) &= S_f(x) + S_f(y \wr x) \\ &\leq S_f(x) + S_f(y) \text{ (by Proposition 5).} \end{aligned}$$

The case $S_f(y \cdot x)$ is similar. ■

Proposition 7 (Kolmogorov's formula[6] for SALZA*):* Given strings $x, y, z \in \mathcal{A}^\star$,

$$I_f(z \cdot x : y) = I_f(z : y) + I_f(x : y \wr z).$$

[6]For the origin of this name, see [25].

Proof: By Eq. 5 on all three terms:

$$I_f(z\cdot x : y) = S_f(z\cdot x) + S_f(y) - S_f(z\cdot x\cdot y)$$
$$I_f(z : y) = S_f(z) + S_f(y) - S_f(z\cdot y)$$
$$I_f(x : y \wr z) = S_f(z\cdot x) + S_f(z\cdot y) - S_f(z\cdot x\cdot y) - S_f(z).$$

■

Proposition 8 (Fast computation of $I_f(x : y \wr z)$):

$$I_f(x : y \wr z) = S_f(y \wr z) - S_f(y \wr x, z). \tag{7}$$

Proof: By Eq. 5 and Def. 3.1:

$$\begin{aligned} I_f(x : y \wr z) &= S_f(z\cdot x) + S_f(z\cdot y) - S_f(z\cdot x\cdot y) - S_f(z) \\ &= S_f(z) + S_f(x \wr z) + S_f(z) + S_f(y \wr z) \\ &\quad - \left[S_f(z) + S_f(x \wr z) + S_f(y \wr z, x)\right] - S_f(z) \\ I_f(x : y \wr z) &= S_f(y \wr z) - S_f(y \wr x, z). \end{aligned}$$

We have used Remark 1 relative to the invariance with respect to permutation of the prior knowledge strings ($S_f(y \wr z, x) = S_f(y \wr x, z)$). ■

*Proposition 9 (*SALZA *conditional mutual measure is positive):* Given strings $x, y, z \in \mathcal{A}^\star$,

$$0 \le I_f(x : y \wr z). \tag{8}$$

Proof: By Eq. 7 and Proposition 5. ■

C. SALZA *as an information measure*

Proposition 10 (Chain rule for SALZA *conditional mutual measure, asymmetric version):* Given strings $x, y, z, t \in \mathcal{A}^\star$,

$$I_f(x : y\cdot z \wr t) = I_f(x : y \wr t) + I_f(x : z \wr t, y). \tag{9}$$

Proof: By Eq. 5:

$$\begin{aligned}I_f(x:y\cdot z\wr t) &= S_f(t\cdot x)+S_f(t\cdot y\cdot z)-S_f(t\cdot x\cdot y\cdot z)-S_f(t)\\&= S_f(t\cdot x)-S_f(t)+S_f(y\wr t)-S_f(x\wr t)\\&\quad -S_f(y\wr x,t)+S_f(z\wr t,y)-S_f(z\wr x,y,t).\end{aligned}$$

Also by Eq. 5:

$$\begin{aligned}I_f(x:y\wr t) &= S_f(t\cdot x)-S_f(t)+S_f(t\cdot y\cdot z)-S_f(t\cdot x\cdot y\cdot z)\\&= S_f(t\cdot x)-S_f(t)+S_f(y\wr t)-S_f(x\wr t)\\&\quad -S_f(y\wr x,t).\end{aligned}$$

By Eq. 7:

$$I_f(x:z\wr t,y)=S_f(z\wr t,y)-S_f(z\wr x,y,t).$$

■

Proposition 11 (Data processing inequality for SALZA*, asymmetric version):* Given strings $x,y,z\in\mathcal{A}^\star$:

$$S_f(y\wr z)=0\implies I_f(x:y\wr z)=0\implies I_f(x:y)\le I_f(z:y).$$

Proof: The first implication is clear by the proof of Proposition 4 since $S_f(y\wr z)=0$ either implies that $y=\emptyset$ or y is a substring of z. By Eq. 7, $I_f(x:y\wr z)=0$.

Also by Eq. 7, $I_f(x:y\wr z)=0\iff S_f(y\wr z)=S_f(y\wr x,z)$. The same equation further states that:

$$\begin{aligned}I_f(x:y)&=S_f(y)-S_f(y\wr x),\\I_f(z:y)&=S_f(y)-S_f(y\wr z).\end{aligned}$$

Replacing $S_f(y\wr z)$ with $S_f(y\wr x,z)$ in the latter, one has that $I_f(z:y)=S_f(y)-S_f(y\wr x,z)$. By Proposition 5, $-S_f(y\wr x)\le -S_f(y\wr x,z)$ and the second implication holds. ■

We now show that S_f satisfies the same requirements as in Sec. 7.1 of [13] to qualify as a submodular information measure. Such an information measure is defined on the lattice of strings, using the lexicographical order as $\le$. We shall rephrase here the approach of [13] to approximate submodularity using joint factorizations.

Let $s,t \in \mathcal{A}^\star$ and z the concatenation of the unique substrings shared by s and t. Let x (*resp.* y) the concatenation of the substrings in s (*resp.* t) that are not in z such that $|s| = |zx|$ (*resp.* $|t| = |zy|$). Then, using joint factorizations, the lattice operators are computed like $s \vee t = z \cdot x \cdot y$ and $s \wedge t = z$. Now, the submodularity property ($S_f(s) + S_f(t) \geq S_f(s \wedge t) + S_f(s \vee t)$) translates into positivity of the conditional mutual information $I_f(x : y \wr z)$. Note that this is only approximate submodularity, since we do not ensure commutativity of the lattice operators ($S_f(x \cdot y) \neq S_f(y \cdot x)$ in general, see Fig. 2).

Proposition 12 (S_f is an information measure in the sense of [13]): Given strings $x,y,z \in \mathcal{A}^\star$:

1) Normalization: $S_f(\emptyset) = 0$;
2) Monotonicity (over prefixes [1], p. 197): $x \leq y \implies S_f(x) \leq S_f(y)$;
3) Approximate submodularity: $S_f(z \cdot x) + S_f(z \cdot y) \geq S_f(z \cdot x \cdot y) + S_f(z)$.

Proof:

1) (Normalization) By Def. 2.5;
2) (Monotonicity over prefixes) Using the lexicographic order on strings, $x \leq y$ implies that one can write $y = xz$ and therefore $|y| = |x| + |z|$. Hence, depending on the kind of factorization used:
   - Case $|$: $\mathcal{L}_{y|\emptyset} = \mathcal{L}_{xz|\emptyset} = \mathcal{L}_{x|\emptyset} + \mathcal{L}_{z|x}$,
   - Case $|^+$: $\mathcal{L}_{y|^+\emptyset} = \mathcal{L}_{xz|^+\emptyset} = \mathcal{L}_{x|^+\emptyset} + \mathcal{L}_{z|^+\emptyset}$.

   Monotonicity over prefixes trivially holds for $S_{\mathbb{1}}$ in both cases ($\mathcal{L}_{y\wr\emptyset}$ can only have more elements than $\mathcal{L}_{x\wr\emptyset}$). For S_p, denote by $l_k^{(x)}$ (*resp.* $l_k^{(y)}$) the (sorted) elements of $\mathcal{L}_{x\wr\emptyset}$ (*resp.* $\mathcal{L}_{y\wr\emptyset}$) and let $\Delta(x,y) = S_p(\mathcal{L}_{y\wr\emptyset}) - S_p(\mathcal{L}_{x\wr\emptyset})$:

$$\begin{aligned}
\Delta(x,y) = |\mathcal{L}_{y\wr\emptyset}| - |\mathcal{L}_{x\wr\emptyset}| &+ \frac{|z|}{(|x|+|z|+1)(|x|+1)} \\
&+ \sum_{k=1}^{p} \left((|x|+1)^{-k} l_k^{(x)} - (|x|+|z|+1)^{-k} l_k^{(y)} \right) \\
&\geq |\mathcal{L}_{y\wr\emptyset}| - |\mathcal{L}_{x\wr\emptyset}| + \frac{|z|}{(|x|+|z|+1)^2} \\
&+ \sum_{k=1}^{p} \left((|x|+1)^{-k} l_k^{(x)} - (|x|+|z|+1)^{-k} l_k^{(y)} \right)
\end{aligned}$$

The first difference above is positive, as well as those in the sum for $1 \leq p \leq |\mathcal{L}_{x\wr\emptyset}|$ (if

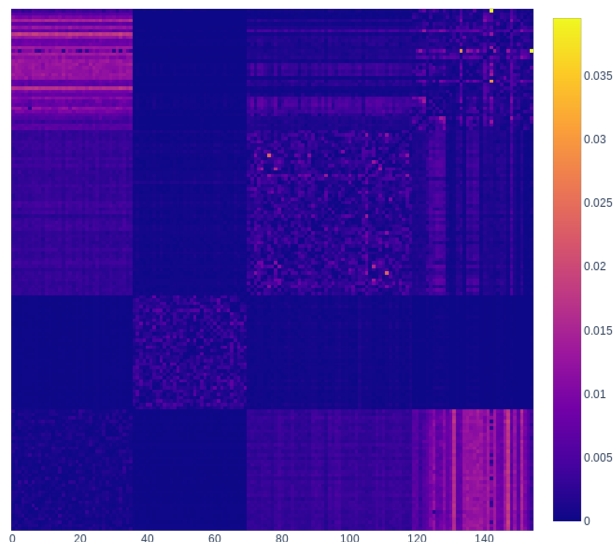


Fig. 2: Heatmap of the departure from symmetry. The mean (*resp.* maximum) departure is 0.24% (*resp.* 3.95%.)

$y = x$, the $l_k^{(y)}$ are the $l_k^{(x)}$). For $p > |\mathcal{L}_{x \wr \emptyset}|$, the sum of the remaining negative terms is always smaller than $|z|/(|x|+|z|+1)^2$ because $|\mathcal{L}_{x \wr \emptyset}| \geq 1$ and the sum is maximal when $|\mathcal{L}_{z \wr \emptyset}| = 1$ (thus $l^{(y)}_{|\mathcal{L}_{x \wr \emptyset}|+1} = |z|$). So for any z and any $p \geq 1$, $\Delta(x,y) \geq 0$;

3) (Approximate submodularity) By Def. 3.2 and Proposition 9.

■

The asymmetry of S_p is tedious to derive analytically. We use four datasets, namely `markov` (see *infra*), and `mammals`, `languages` and `music` from [9], [10] to compute departure from symmetry like:

$$\frac{2|S_p(x \cdot y) - S_p(y \cdot x)|}{S_p(x \cdot y) + S_p(y \cdot x)}.$$

The heatmap of Fig 2 represents departure from symmetry for all pairs of strings in the four datasets we consider. While the mean departure is 0.24%, only some strings (a handful of MIDI music scores) will cause more severe deviation.

From now on, we start specifying explicitly the kind of factorization suited to dedicated applications.

IV. NORMALIZED SALZA SEMI-SISTANCE

As a first application of SALZA, we now devise a universal semi-distance on strings, the Normalized SALZA semi-Distance (NSD). It is universal in the sense that it does not use modeling of the data. We shall compare it to the NCD [9], [10] for classification purposes, since we seek to provide objective measures of performance (which can be hardly derived from the phylogenetic trees produced for clustering applications).

In doing so, we shall describe a straightforward and reproducible way of producing strings from continuously-valued attributes — hence encompassing both discrete and continuous datasets for clustering/classification applications. First, we review the NCD from the point of view of the conditioning.

A. Review of NCD implementation

An information distance between strings x and y is the length of the shortest program for a universal computer to transform x into y and y into x (see [1], p. 642). This ultimately gave birth to the NID (Eq. 1). Using an actual compressor C as a computable proxy for the uncomputable K, one defines the NCD (Eq. 2) as a similarity metric on strings. Note that while the NID is actually a distance (up to negligible errors — see [1], Theorem 8.4.1, p. 662), using an actual compressor may cause deviation from this ideal behaviour.

Let us now consider how the NCD works and focus on the numerator when a LZ77-based compressor is used (*e.g.*, `gzip` or `lzma`). When computing $C(xy)$, the compression is done sequentially. When the sliding window is moving over the boundary of x and y, it is essentially the same as if we were using the factorization $y|x$. But when the sliding window is entirely over y, no regularity information from x can ever be taken into account anymore. The same conclusions may be drawn when considering a block-BWT compressor (*e.g.*, `bzip2`). SALZA, because it does not work by blocks or features a sliding window limited by the host memory, is largely immune to this issue. And it enforces a clean use of the $x|^{+}y$ conditioning without those side-effects.

Yet, the NCD (that assesses string similarity) is not the only approach to universal clustering/classification. Ziv and Merhav [7] have proposed an approach that relies on estimating the KL-divergence (which is akin to assessing dissimilarity between strings) by using $x|^{+}y$ factorizations [8]. We shall see that assessing either similarity or dissimilarity can lead to two semi-distances.

B. Definition of the NSD

Definition 4.1 (Normalized SALZA *Semi-Distance):* Given two strings x,y defined on $\mathcal{A}$ such that $|x|,|y|>1$, the normalized SALZA semi-distance between them is defined as:

$$\mathrm{NSD}_f(x,y)=\frac{\max\left\{S_f(x|^{+}y),S_f(y|^{+}x)\right\}}{\max\left\{S_f(x),S_f(y)\right\}}. \tag{10}$$

Proposition 13 (NSD_f is a normalized semi-distance):

1) $\mathrm{NSD}_f(x,y)=\mathrm{NSD}_f(y,x)$;
2) $0\leq\mathrm{NSD}_f(x,y)\leq 1$;
3) $\mathrm{NSD}_f(x,y)=0 \iff x=y$.

Proof:

1) This is clear by Def. 10;
2) By Proposition 4, $\mathrm{NSD}_f(x,y)\geq 0$. By Proposition 5, $\mathrm{NSD}_f(x,y)\leq 1$;
3) Restricting the identity of indiscernibles to cases $|x|,|y|>1$ avoids the two pathological cases for which NSD_f is indeterminate (after Remark 4). We shall now write the rest of this proof using the generic factorization symbol $\wr$ because either factorization kind would work. Setting $\mathrm{NSD}_f(x,y)=0$ implies that $S_f(x\wr y)=S_f(y\wr x)=0$, in which case x is a substring of y and y is a substring of x, so $x=y$. Now setting $x=y$ implies that $S_f(x\wr y)=S_f(y\wr x)=0$, in which case $\mathrm{NSD}_f(x,y)=0$.

■

Remark 6 (Changing the conditioning): We wrote the above proof using $\wr$ to emphasize that assessing similarity also leads to a semi-distance:

$$\mathrm{NSD}_f^{\mathrm{sim}}(x,y)\frac{\max\left\{S_f(x|y),S_f(y|x)\right\}}{\max\left\{S_f(x),S_f(y)\right\}}. \tag{11}$$

Remark 7 (True distance): It is easy to find counter-examples that violate the triangle inequality for NSD_f (or Eq. 11). If it is an issue, then one can use an external procedure based on Dual-Rooted Prim Trees (DRPT) that will turn a semi-distance matrix into a true distance matrix [26], even if renormalization may be needed afterwards. We have implemented a naive, slow version of the DRPT procedure. For the record, we shall include results using the DRPT procedure in the benchmark of Sec. IV-E.

C. Datasets description

In contrast with [10] where the authors focus on clustering, we would like to assess the performances of universal semi-/distances for classification. Another difference with [10] is that we are seeking a way to naturally extend the application of those metrics to datasets containing continous values describing numeric attributes, not only discrete datasets.

Of course, there should be a conversion from the real domain to discrete values. A lot of works in the literature focus on computing thresholds to binarize each value, but this approach seems to be very data-dependent and therefore hard to reproduce — not to mention that the length of the strings produced is the number of attributes. Instead, we put forward the following simple idea that for each attribute we:

1) Select a different set of two characters ;
2) Normalize each attribute value;
3) Use N characters to actually write down an attribute value.

For example, using character set $\{\texttt{a},\texttt{b}\}$ and $N = 10$, a value of 0.3 will produce the string `aaabbbbbbb`. With a high number of attributes, character sets will ultimately repeat but this approach can already handle a few of them. Table II summarizes the characteristics of the datasets we consider. Obviously, the value of N should be set according to a tradeoff between the length of the strings produced and the precision of the numerical values for the attributes.

The `markov` dataset is the only purely discrete dataset. It was constructed using 6 probability transition matrices between 256 characters. For each matrix, 6 realizations where obtained to produce strings of 15001 characters. This length was chosen to fit the internal 32KiB buffer of `gzip` so that it will handle the concatenation of two strings correctly (for the NCD/`gzip`). The goal for classification is to group realizations by transition matrices. This dataset is the closest to those used in [10] and compressors are expected to perform well.

The `gaussian` dataset was constructed in a more involved fashion, in order to assess the classification robustness against noise. We have chosen respectively $\{22, 20, 25, 26, 27, 24, 24\}$ samples of 7 classes. Each sample contains values for 1000 attributes and 1000 noisy, non-informative values. Each informative attribute value is drawn according to $\mathcal{N}(\mu_k, \sigma^2)$, where $\mu_k \in \{20, \ldots, 26\}$ is the mean value of each class and $\sigma^2 = 23.5$ is their common variance. Non-informative values are drawn according to $\mathcal{N}(19, \sigma^2)$. In total, each column of the sample matrix contains 2000 values and the rows are randomly permuted.

Dataset	String length	# Strings	# Classes	# Attributes
markov	15001	36	6	—
gaussian	102000	168	7	1000
iris	204	150	3	4
wine	663	178	3	13

TABLE II: Datasets for classification performance assessment ($N = 51$).

The standard iris and wine datasets were downloaded from the UCI Machine Learning Repository[7] [27]. Each column contains respectively 4 and 13 attributes. The iris dataset is often advertised as being easily classified. Yet, it is anticipated that the small number of attributes it contains may represent a challenge for universal semi-/distances and the value of N may eventually be the most important parameter. For the sake of uniformity and after taking into account the various numerical precision of the attributes, we set $N = 51$ for all relevant datasets.

D. Methodology

Classification is performed on the semi-/distances matrices using spectral clustering. In order to compute accuracy figures (see below), the true and predicted labels are matched iteratively by selecting the most probable label each time.

Remark 8 (On the NCD): It turns out that the matrices produced by the NCD[8] are neither symmetric nor perfectly normalized. This does not come as a surprise (at least for the lack of normalization), as it was reported in [10].

Yet, this prevents direct use of spectral clustering. Therefore, the NCD matrices are symmetrized by taking the maximum value of $\mathrm{NCD}(i, j)$ and $\mathrm{NCD}(j, i)$ and they are further properly normalized by using the largest value in the matrix.

For each dataset, we have computed the NCD using four standard compressors: gzip [14], [15], lzma [16], bzip2 [28] and ppmd [29]. gzip and lzma are both based on LZ77: gzip uses a 32KiB sliding window with Huffman coding of the characters, the quantized lengths and distances, while lzma uses a default 1MiB sliding window with arithmetic coding of the characters, the quantized lengths and distances, intertwined with Move-To-Front of the distances. bzip2 is based on the BWT of 256KiB-blocks of the input string, encoded using Huffman coding of the characters, processed with Move-To-Front. ppmd uses prediction by partial matching to build a model of

[7] http://archive.ics.uci.edu/ml/datasets

[8] Software downloaded from https://complearn.org.

the next character given the previous ones, producing socalled *context* and *transition* structures that are encoded using range coding. Both `bzip2` and `ppmd` are noted for performing best on natural language or source code inputs, while `gzip` and `lzma` offer more uniform performances — with `lzma` being the current state of the art in general-purpose compression.

E. Results

We shall evaluate the NSD's and the NCD's against the following criterions:

1) accuracy: the ratio of correctly classified samples ;
2) silhouette coefficient [30]: a measure of cohesion (the mean distance of a sample to others in the same cluster) *vs.* separation (the minimum distance between a sample and those in other clusters) — the highest the silhouette coefficient value, the cleanest the clustering ;
3) wall-clock time: the NCD relies on the potential parallelism implemented at the compression routine level (to the best of our knowledge, only `lzma` implements some form of parallelism), while the NSD natively breaks down all factorizations so they can be distributed on the available CPU cores. In this regard, since we did not try to rewrite the NCD to take advantage of parallelism, it should be noted that most running times of the NCD are actually worst-case figures.

Table III reports the overall results obtained for the four datasets and the ten semi-/distances we have benchmarked.

As for the accuracy, the NSD's gives more consistent results than the NCD: the NCD may outperform the NSD (only for the `gaussian` dataset, NCD/`gzip` outperforms NSD^{sim} — yet not the NSD), but at the price of selecting the compressor best suited to the data at hand [11] and with the additional issue that a given compressor may produce catastrophic results (`gzip` for `markov` or `wine`, `lzma` for `gaussian` or `ppmd` for `iris`). For the NSD, the following observations are in order:

1) the DRPT routine surprinsingly *decreases* the classification performances. Actually, the DRPT routine works by inferring the manifold on which the data lives — and we likely have not enough samples for this strategy to work ;
2) comparing NSD vs. NSD^{sim}, the NSD consistently produces clusters with higher silhouette coefficients, yet the accuracy is sometimes better using NSD^{sim} ;
3) comparing $\text{NSD}_{\mathbb{1}}$ vs. NSD, the results are nearly identical — the strings are already long enough so that the added precision of S_p does not help here.

Dataset	Semi-/Distance	Accuracy (%)	Silhouette	Time
markov	$\mathrm{NSD}_{\mathbb{1}}$	**100**	**0.048**	**1.022s**
	NSD	**100**	**0.048**	1.080s
	$\mathrm{NSD}_{\mathbb{1}}^{\mathrm{sim}}$	**100**	0.036	1.045s
	$\mathrm{NSD}^{\mathrm{sim}}$	**100**	0.036	1.219s
	NSD/DRPT	**100**	0.045	+0.045s
	$\mathrm{NSD}^{\mathrm{sim}}$/DRPT	**100**	0.032	+0.047s
	NCD/`gzip`	*36.11*	*0.000*	1.083s
	NCD/`lzma`	97.22	*0.000*	*1m34.347s*
	NCD/`bzip2`	**100**	0.001	9.496s
	NCD/`ppmd`	**100**	0.001	1m7.051s
gaussian	$\mathrm{NSD}_{\mathbb{1}}$	**98.81**	**0.039**	**25.652s**
	NSD	**98.81**	**0.039**	25.946s
	$\mathrm{NSD}_{\mathbb{1}}^{\mathrm{sim}}$	97.02	0.018	26.221s
	$\mathrm{NSD}^{\mathrm{sim}}$	92.26	0.018	26.447s
	NSD/DRPT	91.67	*0.017*	+7.032s
	$\mathrm{NSD}^{\mathrm{sim}}$/DRPT	69.05	*0.005*	+8.942s
	NCD/`gzip`	76.79	0.002	5m21.363s
	NCD/`lzma`	*42.86*	*0.001*	*43m28.433s*
	NCD/`bzip2`	94.64	0.023	2m23.850s
	NCD/`ppmd`	61.90	*0.001*	10m40.040s
wine	$\mathrm{NSD}_{\mathbb{1}}$	91.01	**0.255**	1.575s
	NSD	91.01	0.251	1.486s
	$\mathrm{NSD}_{\mathbb{1}}^{\mathrm{sim}}$	**93.82**	0.046	1.658s
	$\mathrm{NSD}^{\mathrm{sim}}$	93.26	0.045	1.556s
	NSD/DRPT	58.43	0.071	+11.616s
	$\mathrm{NSD}^{\mathrm{sim}}$/DRPT	*46.63*	0.015	+11.091s
	NCD/`gzip`	44.38	*0.011*	**0.977s**
	NCD/`lzma`	67.98	0.025	*4m48.918s*
	NCD/`bzip2`	77.53	0.021	3.013s
	NCD/`ppmd`	60.11	0.019	3m25.158s
iris	$\mathrm{NSD}_{\mathbb{1}}$	86.67	0.430	1.075s
	NSD	87.33	**0.435**	1.101s
	$\mathrm{NSD}_{\mathbb{1}}^{\mathrm{sim}}$	**88**	0.132	1.073s
	$\mathrm{NSD}^{\mathrm{sim}}$	**88**	0.125	1.064s
	NSD/DRPT	70.67	0.296	+5.680s
	$\mathrm{NSD}^{\mathrm{sim}}$/DRPT	67.33	0.231	+6.066s
	NCD/`gzip`	64	*0.097*	1.312s
	NCD/`lzma`	82	0.123	*2m30.652s*
	NCD/`bzip2`	82.67	0.100	**0.886s**
	NCD/`ppmd`	*34.00*	0.113	2m16.631s

TABLE III: Classification performance of universal semi-/distances. The best results are in bold face and the worst are in italic face. The wall-clock time for DRPT variants is reported as the time the naive DRPT routine took in addition to that of the corresponding NSD semi-distance matrix computation. The NSD used 19 CPU cores.

Also for the silhouette coefficient, the NSD's gives more consistent and better results than the NCD, meaning that the NSD clusters are more compact and distant from one another. Again, results vary greatly for the different compressors, with `lzma` producing better clusters more often

than other compressors.

Up to the reservation explained above, the running time is better for the NCD on small datasets (with the notable exceptions of `lzma` and `ppmd` being consistently the slowest performers because of their entropy coding stage based on arithmetic coding). When the size of the dataset increases (`markov` and `gaussian`), the built-in parallelism of the NSD starts to outperform the various NCD's, often by orders of magnitude. Using NSD instead of $\mathrm{NSD}_{\mathbb{1}}$ is hardly noticeable from the running time point of view.

For the record, we include in Fig. 3 the phylogenetic trees obtained from computing both $\mathrm{NSD}^{\mathrm{sim}}$ and NSD on the `language` dataset. This figure reflects the general conclusion of Table III that measuring dissimilarity will often produce more compact and distant clusters than measuring similarity.

V. Causal inference

Named after the initials of its inventors, the PC algorithm [31] is the standard method to perform causal inference. It was first devised for the probabilistic setting using Markovian independence on nodes of the Directed Acyclic Graph (DAG) underlying the observations. Later, it was shown that algorithmic information is also a faithful measure to assess Markovian independence on a graph [12], [32].

Starting from a complete, undirected graph, the greedy PC algorithm will collapse edges based on a local, Markovian measure of independence to produce a *skeleton*. By far, this is the most computational-intensive part since, for each node y, one has to compute independence with each connected nodes x, conditioned on an increasing number of other connected nodes. This process is repeated until either no more edges can be collapsed, or a hard limit is reached (see [33] for more details on computing the skeleton).

The historical, *population*, version of the PC algorithm is noted for its sensitivity to the order in which Markovian independence is tested for edge collapse. Recently, a *stable* version has been proposed [33], that requires *(i)* the edges to be repeatedly tested in the same order and *(ii)* edges to be collapsed only after all candidates have been tested. This makes the *stable* version more robust but slower compared to the *population* version. We have implemented both versions.

In the first step of computing the skeleton, further information is saved for the second, final step, that will produce a Completed Partially DAG from the skeleton — *i.e.*, a graph with as

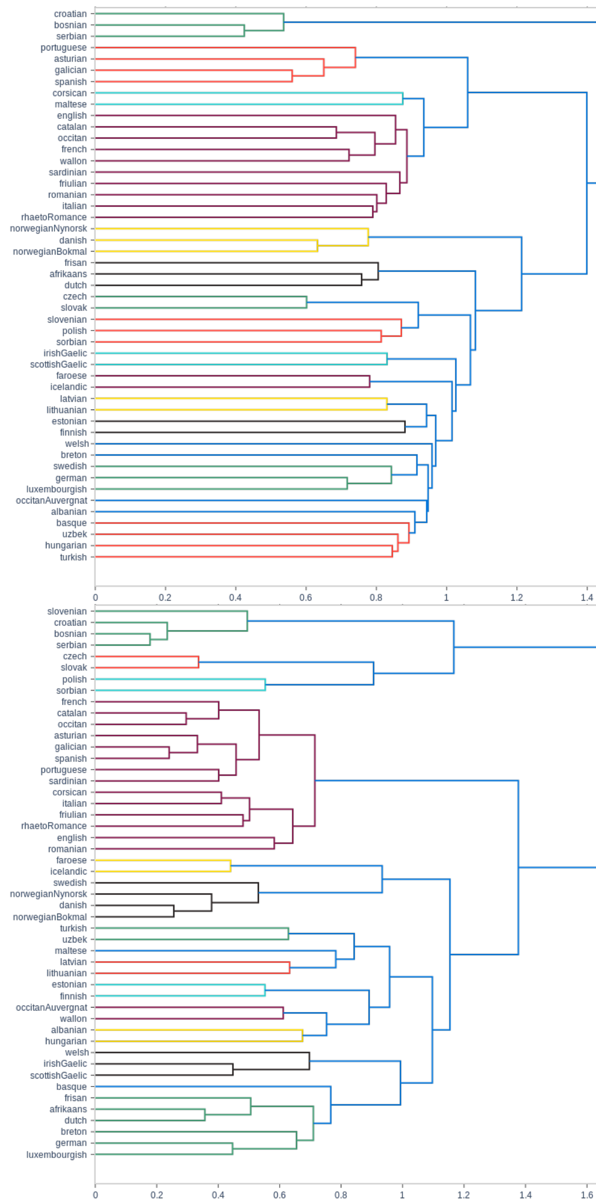


Fig. 3: Phylogenetic trees computed on the `languages` datasets [9], [10] using the Ward distance (top: $\mathrm{NSD}^{\mathrm{sim}}$, bottom: NSD).

much oriented edges as possible [34] that is the representative of the class of DAGs the true DAG underlying the data belongs to.

A. SALZA *normalized independence criterion*

Hence, all that is needed to use the PC algorithm with strings is an efficient routine to compute string independence in order to build the skeleton. The routine should be efficient because, although it is greedy, the PC algorithm has a worst-case complexity of $O(n^3)$ (where n

is the number of strings) — the true complexity depending on the DAG underlying the observed strings. Repeatedly testing for string independence using a compressor would inevitably lead to severe inaccuracies and would require a cumbersome management of the concatenation of strings (not to mention that the search structures would have to be built from scratch all too often). This is where the flexible conditioning of SALZA really shines.

In the probabilistic setting, Markovian independence is typically assessed within 95% confidence intervals. Such a notion of confidence is tedious to translate in the algorithmic setting and we therefore choose to assess independence up to some normalized value η (see Def. 5.1). The default value for η is set to 10^{-2}.

Definition 5.1 (Normalized independence criterion): For strings $x, y, z_1, \dots, z_n \in \mathcal{A}^\star$, $|y| > 1$, x and y are said to be independent of $z_1, \dots, z_n$ up to η, $0 \leq \eta \leq 1$, iff:

$$\frac{I_f(x : y | z_1, \dots, z_n)}{S_f(y)} \leq \eta. \tag{12}$$

Proposition 14 (The left-hand side of Eq. 12 is normalized):

$$0 \leq \frac{I_f(x : y | z_1, \dots, z_n)}{S_f(y)} \leq 1.$$

Proof: By Proposition 5, $S_f(y|x, z_1, \dots, z_n) \leq S_f(y|z_1, \dots, z_n) \leq S_f(y)$. Propositions 4 and 8 conclude. ■

Remark 9 (I_f^+): As a by-product of our implementation, our code also natively implements I_f^+ similarly to Remark 6. Proposition 14 obviously also holds for I_f^+. However, the meaning of I_f^+ is less clear.

B. Causal relationships on languages

From the point of view of causality, representing languages with phylogenetic trees like in Fig. 3 implicitly assumes a tree-like causal structure. Such an assumption may well be regarded as too restrictive, hiding the influences several languages may have had on another.

We include on Fig. 4 the resulting DAG after applying the PC algorithm on the `languages` dataset. Here, it is worth mentioning that the texts that are included in this dataset are translations of the Universal Declaration of Humain Rights. This may explain that some languages are unexpectedly connected: the lexical field of this text is mostly legal, so that some languages may have borrowed terms from another. For example, the English legal lexicon contains a few words from the French.

Lowering η will produce less connected components but will generate more information to orient the edges. We have observed that a few causal structures are preserved while varying η. However, the very idea of using a global value for η may be seen as a limitation of the PC algorithm: further investigation should concentrate on selecting an adequate threshold locally.

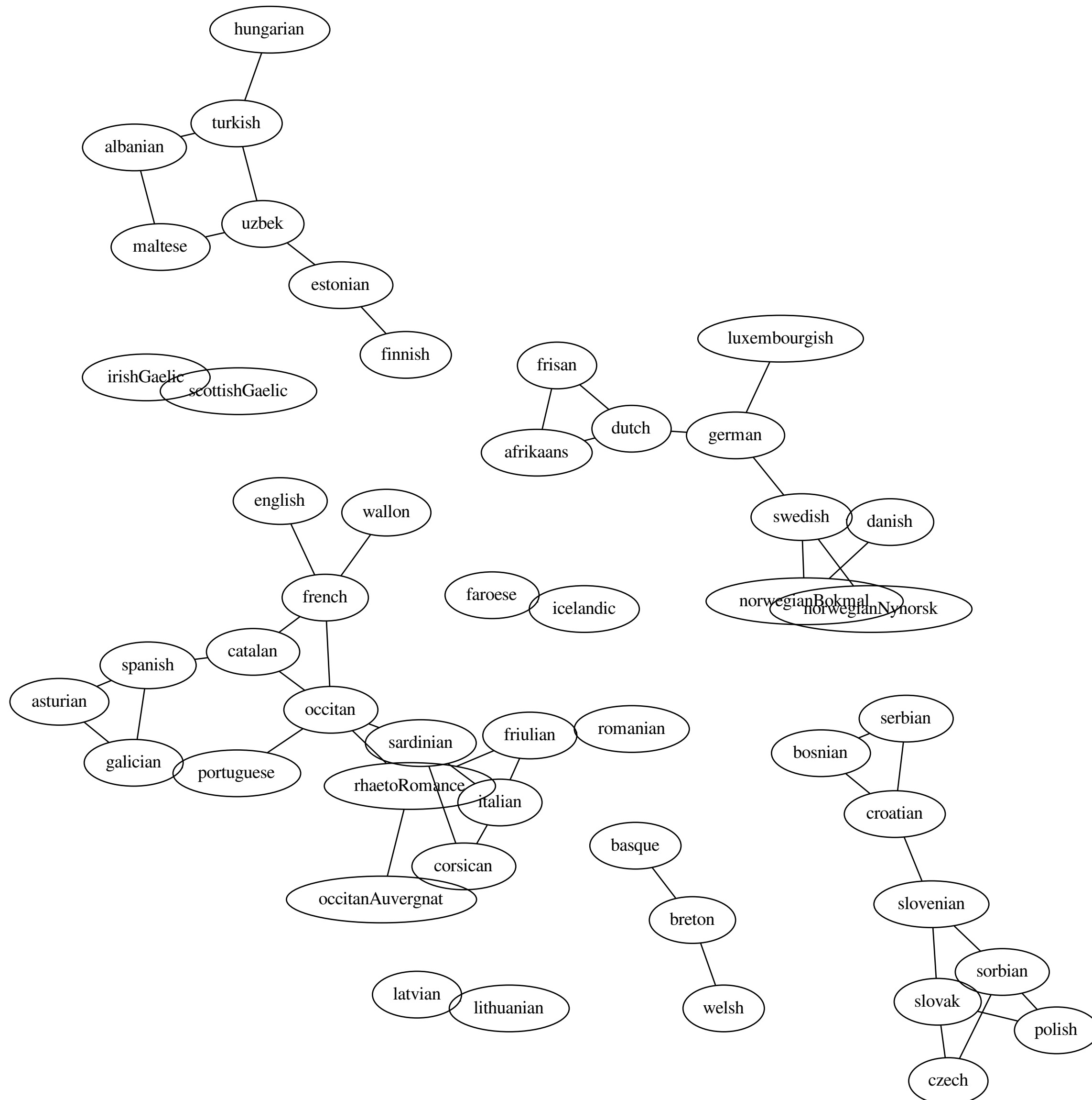


Fig. 4: *Stable* skeleton computed from the `languages` dataset. $\eta = 1\%$ (time: 9m44s.) We did not represent the CPDAG here because only three edges were actually oriented.

C. An experiment in literature

Jean-Philippe Toussaint is a Belgian author who works by typesetting each paragraph, which then gets printed, manually annotated for improvements, typeset again and so on until the final version is reached. We had access to the transcripts of eight successive drafts of one single paragraph in Toussaint's book *La Réticence*. The transcripts each contain approximately 1600 characters, with great changes between the first and only tiny improvements in the last ones.

We depict in Fig 5 the CPDAGs produced by the PC algorithm for its two variants of the computation of the skeleton (*population* and *stable*), as well as for various values of η[9]. The PC algorithm occasionally gets the edges wrong (*e.g.*, transcript 3 is sometimes seen as causing transcript 1, which is impossible), yet it generally finds the correct causal dependencies when the changes are smaller. Following these results, one may formulate the hypothesis that the second transcript is an abandoned trail in the early steps of the writing process followed by Toussaint.

SOFTWARE

The SALZA reference implementation (C code for GNU/Linux and Debian `amd64` binaries repository) and the datasets used in this work are available from:

`https://forge.uvolante.org/code/salza/wikis`

ACKNOWLEDGEMENTS

We are indebted to the anonymous reviewers for providing many helpful comments and pointing us to [13].

The Toussaint drafts are courtesy Prof. T. Lebarbé (Univ. Grenoble Alpes and TGIR Huma-Num).

[9]In Fig. 5, most nodes are connected with two arrows, giving the wrong impression of short cycles. This is indeed a limitation of GraphViz wich allows either for directed or undirected graphs, having no provision for graphs containing both types of edges. When two nodes are connected to each other with two arrows, it only means the algorithm left the edge undirected.

APPENDIX A
IMPLEMENTATION NOTES

A. Adjusting the precision p

It is clear from Eq. 3 that the maximum useful value for p is a function of the machine epsilon ε and $|y|$. Hence, p is selected such that:

$$p = \left\lfloor \frac{-\log \varepsilon}{\log(|y|+1)} \right\rfloor .$$

With standard IEEE-754 `binary64` floating point `double`'s, $\varepsilon = 2.22 \times 10^{-16}$. Therefore, SALZA will take into account at least the first three l_k's for strings of size $|y| \leq 1.3 \times 10^5$, and it will use only l_1 starting from strings of size $|y| \geq 6.7 \times 10^7$. Of course, $|\mathcal{L}|$ is the dominant part in $S_p(\mathcal{L})$. Yet, SALZA will automatically allocate as much precision as possible for shorter strings, where it is needed more.

B. Basic algorithms and search structures

Because SALZA looks for the next longest substring in a set of strings, it requires to be able to efficiently do so from an arbitrary position in the string y being factorized.

In the vast literature pertaining to computing the Lempel-Ziv factorization, much of the works take advantage of incrementally factorizing a string (meaning that the next Lempel-Ziv factors are known only for positions at which they start in the string being factorized). However, one such algorithm exists [35] that computes the next Lempel-Ziv factor from an arbitrary position: `CPS1`[10] (in linear time if the entire string were to be factorized).

This algorithm relies on the suffix array of the string being factorized, which can be constructed in linear time and uses 8 bytes of memory per string character [36]. It also makes use of the longest common prefix array, the construction and storage of which share roughly the same time and space performances [37] as that of the suffix array.

The suffix array is the only data structure required to efficiently perform cross-factorization of y given x, also from an arbitrary position in y [18]. Hence, upon start-up, SALZA builds the suffix array of all strings, saving the construction of the longest common prefix array in cases it is not needed.

[10]This mandatory feature we need is even called an "undesirable aspect" of `CPS1` (see [35], Sec. 2.2).

C. Parallel computations

Various methods have been developed to compute the Lempel-Ziv factorization in parallel (*e.g.* [38]), but they often lack the flexibility we need and their granularity sometimes is too fine-grained for CPU threads.

Instead, we have devised the following strategy. Each string to be factorized is divided in blocks. Each block factorization parameters (block start, strings involved, factorization kind for each string, *etc.*) are prepared and stored into a work queue. Once the work queue is ready, threads keep picking up factorization parameters for the next block and the work queue eventually gets empty. In order not to freeze the machine, we leave one core free for the other processes.

At this point, most of the work has been done in parallel. But it may happen that the Lempel-Ziv factors overlap block boundaries (the last factor of a block spans over the beginning of the next block). Hence, we perform a few additional calls to the main factorization routine (starting from the end of the last factor in a block), until the factorizations of the two consecutive blocks are synchronized. In practice, such a "stitching" overhead typically means computing less than 3 additional factors per block. This way, it is guaranteed that the resulting factorization is the same as if it were performed sequentially. Note that it is crucial for this strategy to be able to compute Lempel-Ziv factors from an arbitrary position.

Among the applications listed, computing the NSD_f matrix is done even more efficiently in parallel since all computations can be scheduled right from the start (which is less easier with the PC algorithm for example). In this case, the work queue is constructed only once. Whereas for all other applications, parallelism takes place mostly at the whole string factorization level (as described above). For example, computing one conditional mutual information measure leads to the creation of a work queue containing the parameters for all blocks of each of the two parts of Eq. 7 and we can still benefit from parallel computations in performing causal inference.

References

[1] M. Li and P. M. Vitániy, *An Introduction to Kolmogorov Complexity and Its Applications*, 3rd ed. Springer-Verlag New-York, 2008. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-49820-1

[2] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. Wiley, 1991.

[3] C. H. Bennett, P. Gács, M. Li, P. M. Vitániy, and W. H. Zurek, “Information Distance,” *IEEE Transactions on Information Theory*, vol. 44, pp. 1407–1423, Jul. 1998. [Online]. Available: http://dx.doi.org/10.1109/18.681318

[4] A. Lempel and J. Ziv, “On the Complexity of Finite Sequences,” *IEEE Transactions on Information Theory*, vol. 22, pp. 75–81, January 1976. [Online]. Available: http://dx.doi.org/10.1109/tit.1976.1055501

[5] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337–343, May 1977. [Online]. Available: http://dx.doi.org/10.1109/tit.1977.1055714

[6] ——, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory*, vol. 24, pp. 530–536, September 1978. [Online]. Available: https://doi.org/10.1109/TIT.1978.1055934

[7] J. Ziv and N. Merhav, "A Measure of Relative Entropy Between Individual Sequences with Application to Universal Classification," *IEEE Transactions on Information Theory*, vol. 39, pp. 1270–1279, Jul. 1993. [Online]. Available: http://dx.doi.org/10.1109/isit.1993.748668

[8] A. D. Wyner and J. Ziv, "Fixed Data Base Version of the Lempel-Ziv Compression Algorithm," *IEEE Transactions on Information Theory*, vol. 37, pp. 878–880, May 1991. [Online]. Available: https://doi.org/10.1109/18.79955

[9] M. Li, X. Chen, X. Li, B. Ma, and P. M. Vitányi, "The Similarity Metric," *IEEE Transactions on Information Theory*, vol. 50, pp. 3250–3264, Dec. 2004. [Online]. Available: http://dx.doi.org/10.1109/tit.2004.838101

[10] R. Cilibrasi and P. M. Vitányi, "Clustering by Compression," *IEEE Transactions on Information Theory*, vol. 51, pp. 1523–1545, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1109/tit.2005.844059

[11] M. Cebrián, M. Alfonseca, and A. Ortega, "Common Pitfalls Using the Normalized Compression Distance: What to Watch Out for in a Compressor," *Communications in Information and Systems*, vol. 5, pp. 367–384, 2005. [Online]. Available: http://dx.doi.org/10.4310/cis.2005.v5.n4.a1

[12] D. Janzing and B. Schölkopf, "Causal Inference Using the Algorithmic Markov Condition," *IEEE Transactions on Information Theory*, vol. 56, pp. 5168–5194, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1109/TIT.2010.2060095

[13] B. Steudel, D. Janzing, and B. Schölkopf, "Causal Markov Condition for Submodular Information Measures," in *Proceedings of the 23rd Annual Conference on Learning Theory*. Madison, WI, USA: OmniPress, Jun. 2010, pp. 464–476. [Online]. Available: https://is.tuebingen.mpg.de/publications/6772

[14] L. P. Deutsch, "DEFLATE Compressed Data Format Specification, version 1.3," 1996. [Online]. Available: http://www.gzip.org/zlib/rfc-deflate.html

[15] ——, "GZIP File Format Specification, version 4.3," 1996. [Online]. Available: http://www.gzip.org/zlib/rfc-gzip.html

[16] I. Pavlov, "7z Format."

[17] M. Crochemore, L. Ilie, and W. F. Smyth, "A Simple Algorithm for Computing the Lempel-Ziv Factorization," in *Proceedings of the 18th Data Compression Conference*. Snowbird, UT, USA: IEEE, Mar. 2008, pp. 482–488. [Online]. Available: https://ieeexplore.ieee.org/document/4483326

[18] C. Hoobin, S. J. Puglisi, and J. Zobel, "Relative Lempel-Ziv Factorization for Efficient Storage and Retrieval of Web Collections," *Proceedings of the VLDB Endowment*, vol. 5, pp. 265–273, Nov. 2011. [Online]. Available: https://dl.acm.org/citation.cfm?id=2078331.2078341

[19] T. I. Fenner and G. Loizou, "A Binary Tree Representation and Related Algorithms for Generating Integer Partitions," *The Computer Journal*, vol. 23, pp. 332–337, Jan. 1980.

[20] D. E. Knuth, *The Art of Computer Programming*. Addison Wesley Longman Publishing Co., Inc., 1997.

[21] N. Cancedda, E. Gaussier, C. Goutte, and J.-M. Renders, "Word-Sequence Kernels," *Journal of Machine Learning Research*, vol. 3, pp. 1059–1082, Feb. 2003. [Online]. Available: http://www.jmlr.org/papers/v3/cancedda03a.html

[22] M. Burrows and D. J. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," Digital Equipment Corp. SRC, Tech. Rep., 1994. [Online]. Available: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

[23] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, pp. 1098–1101, 1952. [Online]. Available: http://dx.doi.org/10.1109/jrproc.1952.273898

[24] Peter Elias, "Universal Codeword Sets and Representations of the Integers," *IEEE Transactions on Information Theory*, vol. 21, pp. 194–203, Mar. 1975. [Online]. Available: https://doi.org/10.1109/tit.1975.1055349

[25] A. D. Wyner, “A Definition of Conditional Mutual Information for Arbitrary Ensembles,” *Information and Control*, vol. 38, pp. 51–59, July 1978. [Online]. Available: https://doi.org/10.1016/S0019-9958(78)90026-8

[26] L. Galluccio, O. Michel, P. Comon, M. Kilger, and A. Hero, “Clustering with a New Distance Measure based on a Dual-Rooted Tree,” *Information Sciences*, vol. 251, pp. 96–113, Dec. 2013. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00726005v2

[27] D. Dua and C. Graff, “UCI machine learning repository,” 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[28] J. Seward, “bzip2,” 1996. [Online]. Available: https://sourceware.org/bzip2/

[29] D. Shkarin, “PPM: One Step to Practicality,” in *Proceedings of the 12th Data Compression Conference*. Snowbird, UT, USA: IEEE, Apr. 2002, pp. 202–211. [Online]. Available: http://dx.doi.org/10.1109/DCC.2002.999958

[30] P. J. Rousseeuw, “Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis,” *Journal of Computational and Applied Mathematics*, vol. 20, pp. 53–65, Nov. 1987. [Online]. Available: https://doi.org/10.1016/0377-0427(87)90125-7

[31] P. Spirtes, C. Glymour, and R. Scheines, *Causation, Prediction, and Search*. Springer, 1993. [Online]. Available: http://dx.doi.org/10.1007/978-1-4612-2748-9

[32] J. Lemeire and D. Janzing, “Replacing Causal Faithfulness with Algorithmic Independence of Conditionals,” *Minds and Machines*, vol. 23, pp. 227–249, 2012.

[33] D. Colombo and M. H. Maathuis, “Order-Independent Constraint-Based Causal Structure Learning,” *Journal of Machine Learning Reasearch*, vol. 15, pp. 3921–3962, 2014.

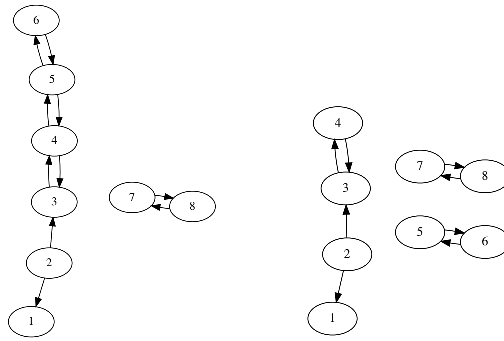
[34] J. Pearl, *Causality, Models, Reasoning, and Inference*. Cambridge University Press, 2009. [Online]. Available: http://dx.doi.org/10.1017/cbo9780511803161

[35] G. Chen, S. J. Puglisi, and W. F. Smyth, “Lempel-Ziv Factorization using Less Time and Space,” *Mathematics in Computer Science*, vol. 1, pp. 605–623, Jun. 2008. [Online]. Available: https://link.springer.com/article/10.1007/s11786-007-0024-4

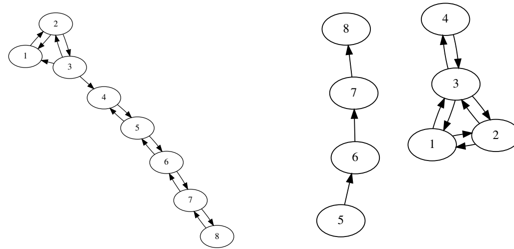
[36] J. Kärkkäinen, P. Sanders, and S. Burckhardt, “Linear Work Suffix Array Construction,” *Journal of the ACM*, vol. 53, pp. 918–936, Nov. 2006.

[37] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, “Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications,” in *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM)*. Jerusalem, Israel: Springer, Jul. 2001, pp. 181–192. [Online]. Available: https://doi.org/10.1007/3-540-48194-X_17

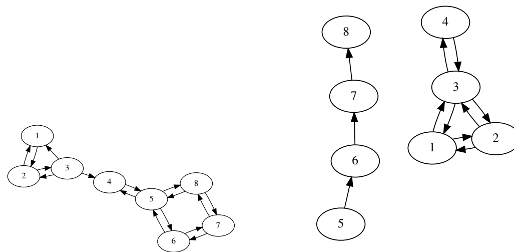
[38] J. Shun and F. Zhao, “Practical Parallel Lempel-Ziv Factorization,” in *Proceedings of the 23rd Data Compression Conference*. Snowbird, UT, USA: IEEE, Mar. 2013, pp. 123–132. [Online]. Available: https://dx.doi.org/10.1109/DCC.2013.20



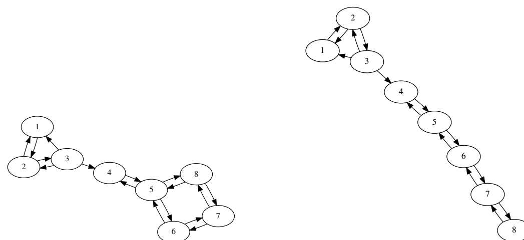
(a) *Population* skeleton, $\eta = 0.02$.
(b) *Stable* skeleton, $\eta = 0.02$.



(c) *Population* skeleton, $\eta = 0.01$.
(d) *Stable* skeleton, $\eta = 0.01$.



(e) *Population* skeleton, $\eta = 0.009$.
(f) *Stable* skeleton, $\eta = 0.009$.



(g) *Population* skeleton, $\eta = 0.008$.
(h) *Stable* skeleton, $\eta = 0.008$.

Fig. 5: Causal inference on the Toussaint drafts: *population* (*resp. stable*) skeleton and CPDAG mean computation time: 0.339s (*resp.* 0.405s).