First Steps Toward Compression-aware Algorithms for Efficient Multi-party Computation

Technical Research Paper
Presented to the Faculty of the
School of Engineering and Applied Science
University of Virginia

By

Yonathan Fisseha

May 08, 2020

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed: _____

Approved: _____ Date _____
Nathan J. Brunelle, Assistant Professor of Computer Science, Department of Computer Science

First Steps Toward Compression-aware Algorithms for Efficient Multi-party Computation

Yonathan Fisseha
yf2ey@virginia.edu
Department of Computer Science
University of Virginia
Charlottesville, Virgina

ABSTRACT

The world is seeing an unprecedented growth in data with the proliferation of user-centered applications. Unfortunately, this data is also privacy sensitive and often shared across individual and organizational boundaries. One solution that respects the privacy of the users while enabling a form of data sharing is multi-party computation (MPC). More specifically, it enables users to share the result of computations performed on their aggregated data without exposing the aggregated data. However, the performance of MPC had made the theoretically sound solution impossible in practice until recently. This work is the first step in a larger project that aims to contribute to the ongoing effort of making MPC more practical. The project proposes to apply compression and compression-aware algorithms to decrease the size of the input and output of the MPC protocol without incurring the cost of decompression. This work's contribution is framing the design of the algorithms in terms of promise problems, and proposing a partitioning for the language.

KEYWORDS

Compression, Compression-aware algorithms, Multi-party computation, Privacy

1 INTRODUCTION

As computers continue to play an integral part in our society, digital data also continues to grow in size and variety. A particular example for this relationship is the start and growth of Big Data. Big Data is large, unstructured, dynamic data mostly generated over the Internet, especially due to the proliferation of customer-data driven applications in Web 2.0, such as social media services [10]. The global data is expected to grow to 40,000 exabytes between 2005 and 2020 by doubling every two years [9]. Moreover, around 44% of this data is expected to go through cloud providers, and as much as 33% of the data will have analytic value [9]. As the data is generated by individuals, a majority of it carries private information, thus there is a growing need to not only efficiently compute on this large and unstructured data but to do so securely. The technical solution must recognize the dilemma in the need to aggregate data and the need for privacy. This dilemma can be resolved for at least a subset of use cases, such as data sharing between organizations, by enabling them to privately compute on the aggregated data without exposing the data of each data contributor, and then sharing the result of the computation with all parties.

Classical algorithm and system designs fail to meet the challenge of providing privacy-aware services based on this massive data. They lack the computational efficiency that scales to the data size, while providing strong security guarantees to protect the users' privacy. This report introduces the first steps towards the design of systems that are efficiently scalable to the scale of Big Data (and more) while offering strong security guarantees that allow arbitrary computation on the private data. It relies on two foundational priori concepts: compression-aware algorithms and multi-party computation.

2 OUTLINE

Section 3 provides an overview of data compression and multi-party computation, which form the basis of this project. Section 4 introduces the technical contribution of this work via promise problems. Section 5 introduces the experimental work this project contributes. The subsections within cover the experiment design, experiment data, and analysis of the data. Sections 7 and 6 conclude the report and suggest further work towards the goals introduced earlier.

3 BACKGROUND

Data Compression

Informally, compression is the process of representing information using fewer number of computer bits. The implicit assumption, which was not formally stated for many years, is that information typically has redundancy which can be removed. Morse code offers a great intuition of compression without the formality and complexity of modern day compression schemes. Morse code was introduced in 1837 by Samuel Morse as a way to transfer information through electric pulses, with no clear intention of compressing information [19]. Morse code works by signalling longer and shorter electrical currents, where the various combinations

of long and short pluses map to the English alphabet [19]. For example, the letter `e` is represented by a single short pulse while the letter `y` is represented by a long-short-long-long pulse. The length of pulses for each letter is associated with how frequently the letter is used in the English language, hence why `e`, which is the most frequently used letter, has a shorter pulse representation than `y`, which is not that frequently used. For comparison, consider a hypothetical version of Morse code where every letter has a unique but equally long signal. Intuitively, one can observe that Morse code will transfer any sufficiently large English text with fewer pulses than the hypothetical version, since it represents letters based on frequency.

Shannon formalized this notion of representing information using the concept of entropy [22]. A given message is said to have high entropy if it holds more information and inversely said to have low entropy if the information is not unique. Naturally, this formalizes the idea that not all information is equally important; for example, redundant information within a message does not decrease the total information of the message. Compression is then the act of maximizing entropy. Mathematically, the entropy $\mathbb{H}$ of a message M is defined as,

$$\mathbb{H}(m) = \sum_{m \in M} \frac{p(m)}{log(p(m))} \tag{1}$$

where $p(m)$ is the probability of $m \in M$. It follows then that the entropy will always be above zero since $\mathbb{H}$ is a positive function. The expression properly captures the intuition that Morse code is more succinct than a random assignment of pulse lengths since it takes the probability (or frequency) of a letter into account, thus it represents the information more succinctly.

How well a given message is compressed under some compression scheme, such as Morse code, can then be quantified as the compression ratio **CR**,

$$\mathbf{CR} = \frac{uncompressed\ size}{compressed\ size} \tag{2}$$

Modern compression schemes are typically divided into lossless and lossy. Lossless compression schemes aim to preserve all information in the message while lossy schemes relax this requirement in order to increase the compression ratio. The nuance comes from what one considers redundant. For example, the human eye does not register most of the information in a digital image thus image compression techniques are often lossy, where they sacrifice as much information as possible from the image without losing image quality noticeable to the human eye. The JPEG image format does this using discrete cosine transformation, followed by quantization and an application of the baseline lossy compression algorithm [24]. However, lossy compression is not acceptable for many applications, such as text files. A commonly used lossless compression scheme is Lempel-Ziv Compression.

Lempel-Ziv Compression

The Lempel-Ziv compression scheme was first proposed in 1977 by Ziv and Lempel as LZ'77 and refined again in 1978 into LZ'78 [27, 28]. The scheme is built around a simple concept: for each repeated group of sequences, save one instance of the sequence, and all other occurrences can be replaced by a pointer to the single saved instance. Since the pointers to the dictionary items can be represented more compactly than the sequences itself, the representation achieves compression. This technique is now used widely, and has produced various similar algorithms referenced as the LZ families. Other works have made various improvements to the LZ-family related to compaction, parallelization, hardware acceleration, and reliability [13, 15, 21]. This work depends on the original LZ'78 compression scheme. The compression and decompression algorithms are presented below.

Algorithm 1: LZ'78 Compression Algorithm

```
Result: LZ'78 compression dictionary
dictionary = Null;
char = string[0];
prefix = Null;
while char ≠ end of string do
    if char ∈ dictionary then
        prefix.append(char);
    else
        dictionary.add(prefix,char);
        prefix = Null;
    end
    char = string[next]
end
```

Retrieving the dictionary content in the order of insertion produces the LZ'78 compressed version of the input string. The compressed string would thus be a set of tuples of the form `(index to prefix, new char)`.

Note that traversing the dictionary in the insertion order of the compression enforces the property that the code at index j `(i, char)` where index i is code `(m, char)` and if $i < j$, j will point to an already decompressed code $(m, char)$. The compression and decompression algorithms run in time $O(m)$ where m is the size of the message being compressed and decompressed. The linear bounds follow from basic analysis of the algorithms, however Ferragina et al. have shown tighter bit-complexities [7]. Ziv and Lempel have given a constructive proof, based on finite-state encoders which theoretically parallels Shannon's entropy, that LZ is the lower bound on the compressibility of a string [28].

Algorithm 2: LZ'78 Decompression Algorithm

```
Result: Decompressed string
compressed = LZ'78 compressed string input;
dictionary = compression dictionary generated by
 LZ'78;
while compressed ≠ empty do
    index, char = compressed.pop();
    code = dictionary.get(index).append(char);
    dictionary.replace(index, code);
end
```

Compression-aware Algorithms

Data compression is classically seen as a long-term storage optimization mechanism [18]. Consequently, many of the classical designs and implementations are focused on optimizing for long-term storage. However, as the growth of data continue to increase, the performance bottleneck in the von Neumann computer architecture becomes the in-memory space and speed as well. Compression-aware algorithms aim to introduce the benefits of compression to data residing in memory and being processed by algorithms.

The classical processing of using compression follows three basic steps: compress the data, store the data in long-term storage, decompress the data when an algorithm needs it. However, both the compression and decompression steps are typically time consuming as they willingly trade-off their time complexities for the compression quality. Therefore, if compression is to be used in short-term storage scenarios, performance cost of compression and decompression must be negligible. Compression-aware algorithms ofter a path towards this goal by avoiding the decompression step altogether. A compression-aware algorithm is an algorithm that is designed with the intention of computing on compressed data. Typically, the algorithm is designed to work with a specific compression scheme. Recent works have successfully applied this technique to improve performance in DNA alignment, streaming data, and query processing [4, 5, 12]. Previous work, related to this project, has shown searching and sorting to be possible on LZ'77 compressed string with asymptotically better runtime [2].

This project is focused on designing a compression-aware equivalent of the longest common subsequence algorithm. LCS is often used to measure the similarity of strings and, in genomics to extract common sections of genes. It is classically solved using dynamic programming techniques and runs in time $O(n^2)$, in the length of the strings. As mentioned in the previous section, the compression scheme under consideration is LZ'78.

The classic dynamic programming solution produces a matrix of size $m \times n$ where m and n are the sizes of the two strings. `matrix[m-1, n-1]` gives the length of the longest common subsequence. Simple backtracking techniques can recover the string itself. The challenge is then to perform the same operations on LZ'78 compressed strings instead of the uncompressed strings and yield the same table.

Multi-party Computation

Multi-party computation is a process that enables a group of independent parties to contribute private input to a computation and collaboratively compute on the aggregated data without disclosing any of the private individual inputs [6]. MPC is particularly effective in cases where the parties cannot trust each other nor any third-parties since it fully protects the private inputs of the parties. The general concept of MPC was introduced by Andrew Yao in the 1980s [26]. He later designed the Garbled Circuit protocol as a realization of the concept, which forms the basis of most MPC protocols used today [6].

The following example was popularized by Yao: two millionaires would like to find who is richer but would like to keep their respective net-worth a secrete. More formally, we are interested in computing the function $\mathcal{F}(n_1, n_2) = n_1 > n_2$ where n_i is the network of the millionaires without disclosing n_i to any party. The MPC protocol Π must enable the computation of the function $\mathcal{F}(x_1 ... x_n)$ with n parties providing the n_i inputs. $\mathcal{F}$ must be a discrete function, which the is true in this case. Furthermore, for brevity, lets restrict the domain of n for the running example to $\{100, 1000\}$. Then Yao's Garbled Circuit is a valid protocol Π.

A Garbled Circuit (GC) is an MPC protocol defined as a quadruple (Gb, En, Ev, De), where:

(1) $Gb(1^\lambda, \mathcal{F}) \rightarrow (F, e, d)$: a garbling algorithm that takes a security parameter λ, and discrete function $\mathcal{F}$ and produces a garbled circuit F, an information encoding key e, and an information decoding key d
(2) $En(e_1, e_2, x) \rightarrow X$: an encoding algorithm which takes in keys e_1 and e_2 and information x and produces X, a garbled output under the two keys. Typically En is a double-key encryption algorithm
(3) $Ev(F, X) \rightarrow Y$: an evaluator which takes a garbled circuit F and a garbled input X, and produces a garbled output Y
(4) $De(d_1, d_2, Y) \rightarrow y$: a decoding algorithm that takes in decoding keys d_1, d_2 and garbled input Y and produces a plaintext y. Typically a double-key decryption algorithm related to En

Then we say the garbled circuit $\mathcal{G}$ is correct if,

$$Pr[De(d, Ev(F, En(e, x))) = \mathcal{F}(x)] = 1 \tag{3}$$

where $Gb(1^\lambda, \mathcal{F}) = (F, e, d)$ for a sufficiently large λ [25]. The full correctness argument as well as the constructive

security proof was given by Yao [26]. Continuing the running example, the function $\mathcal{F}$ can be seen as a table T of size $|D_x||D_y|$ where D is the domain of the variables. T enumerates the mapping between inputs X, Y to outputs. The function $\mathcal{F}(x, y)$ can then be easily evaluated as a lookup $T[x, y]$. Since the function at hand has a small domain and is discrete, it can be exhaustively enumerated here.

n_1	n_2	Output
100	1000	0
1000	100	1
100	100	0
1000	1000	0

Note that every party has access to $\mathcal{F}$, thus can easily construct the table above. Then let the first millionaire be m_1 and the second m_2. Arbitrarily choose one of them to initiate the protocol; the example continue with m_1. Then m_1 generates secure keys k_i, k_j corresponding to each unique x_i, x_j in the input domains D. It then encrypts or encode each output in T using the corresponding keys for the row, $En(k_i, k_j, table[i, j])$. The result of this encryption procedure is the garbled circuit F; that is, m_1 has executed Gb. To avoid leaking data based on the fixed order elements in T, m_1 randomly permutes the rows of T.

Output
$En(k_1, k_0, 1)$
$En(k_0, k_1, 0)$
$En(k_1, k_1, 0)$
$En(k_0, k_0, 0)$

The next step requires m_1 and m_2 to exchange keys. The process proceeds with m_1 sending F and the key x' that corresponds to $m_1's$ input– the first millionaire's net worth. Note that x' is a randomly generated key that does not leak information related to the corresponding value. Given x' and F, m_2 needs to find the correct row that corresponds to their input. In other words, m_2 needs y' such that,

$$\exists_{i,j \in T} De(x', y', En(x', y', T[i, j])) = \mathcal{F}(x, y) \tag{4}$$

holds true. However, since T was uniquely generated by m_1, m_2 cannot directly discover the corresponding key to their input. Similarly, if m_2 was to directly ask m_1 for the key that corresponds to their input, they would leak their value. Therefore, m_2 must receive the key y' from m_1 through oblivious transfer. Oblivious transfer is a necessary building block for MPC that allows a party to receive a secret from another party without learning anything extra and without the sender learning anything [6]. Formally, for two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$, where $\mathcal{S}$ holds secrets $\{s_1, ..., s_n\}$ and $\mathcal{R}$ holds selector bit $b \in \{0, 1\}^n$; oblivious transfer enables $\mathcal{R}$ to receive s_b and $\mathcal{S}$ to receive nothing. The example at hand requires a 1-out-of-N oblivious transfer protocol. Implementations and security of oblivious transfers is covered by Evans et al. [6].

Once m_2 securely receives y', they can exhaustively attempt to decrypt/decode every row as $\forall_{i,j \in T} De(x', y', T[i, j])$ until they successfully decrypt. This requires m_2 to know when a De operation is successful. Beaver et al. offer the point and permute technique which enables m_2 to know which row to decrypt without having to attempt decrypting each row [1]. Once m_2 successfully decrypts the correct row, it can share the result of the decryption, which is the result of $\mathcal{F}$ with m_1.

Challenges of Garbled Circuits

Recent works have improved the performance of garbled circuit protocol for arbitrary functions as well as specific functions and algorithms [14, 16, 23]. This work is a first step towards another optimization of garbled circuit protocols' performance. The main cost of garbled circuit protocols, especially as they scale to more than two parties, is the network bandwidth required to transmit T and the computation required to generate and evaluate T [6]. Network bandwidth bottlenecks are common in system design due to the speed difference between local execution of programs and the latency of networks, which in practical settings typically span large geographical distances. The theoretical communication complexity lower bound for MPC has been established to be sublinear, independent of the circuit size. Specifically, Couteau shows the communication complexity is,

$$O\left(n + N\left(m + \frac{s}{\log\log(s)}\right)\right) \tag{5}$$

where s is the size of the layer boolean circuit, N is the number of parties, and m, n are the sizes of the output and input respectively. Moreover, this requires storage polynomial in s,

$$O\left(\frac{s^2}{\log\log(s)}\right) \tag{6}$$

where s is the size of the circuit [3]. Note that Yao's classic garbling scheme does not achieve the optimal communication complexity, but other more recent schemes meet this challenge [6].

Similarly, the generation of T is directly lower-bounded by the expensive cryptographic operations, which in turn are dependent on m and n– the input and output sizes. Therefore, we aim to decrease m and n using compression techniques, and furthermore, design $\mathcal{F}$ to be compression-aware such that the parties do not incur the additional cost of decompression when evaluating $\mathcal{F}'(x', y')$, where x', y' are the strings

x, y compressed under an algorithm-friendly compression scheme, such as the LZ family.

Formally, the Garbling Circuit scheme becomes a sextuple $\mathcal{G}' = (Gb, En, Ev, De, Co, Do)$ where,

(5) $Co(x) \rightarrow (CS, DI)$: is a compression scheme receives a plaintext value x and returns the compression of x and an optional compression dictionary Di
(6) $Do(CS, DI) \rightarrow x$: is a decompression scheme that takes a compressed plaintext value CS and the compression dictionary DI, and returns the decompressed plaintext x

Since Co is a bijective function mapping $D_i \rightarrow D_i'$, where D_i is the domain of party i's input and D_i' is the domain of the compressed plaintexts, all $X \in D_i$ can be replaced by $X' \in D_i' : Co(X)$ in the garbling scheme $\mathcal{G}$ with no other change to produce $\mathcal{G}'$.

4 CONTRIBUTION

This project is a part of a larger project that aims to design a garbling scheme $\mathcal{G}'$ where $\mathcal{F}'$ is a compression-ware string algorithm, such as Edit Distance, Longest Common Subsequence, and Hamming Distance. The compression scheme is the LZ family, and more specifically LZ'78. Previous work from the group has attempted to augment the LCS algorithm with compression-awareness for LZ'78.

However, our initial attempt at solving this problem showed that the problem is rather difficult, so we now consider a promise problem formulation instead. A promise problem is one where the algorithm designer is promised the input to their algorithm comes from a subset of all strings [11]. More formally, a promise problem is the partitioning of a language $\mathbb{L} \subseteq \{0,1\}^*$ into three languages,

(1) The language $\mathbb{L}_{YES}$ representing YES-strings
(2) The language $\mathbb{L}_{NO}$ representing NO-strings
(3) The language $\mathbb{L}_{Dis}$ representing disallowed strings

The promise algorithm is expected to distinguish between the *YES* language and *NO* language but is allowed arbitrary behaviour on the disallowed language, including not halting [11]. Consequently, we also get the guarantee that $\mathbb{L}_{YES} \cap \mathbb{L}_{NO} = \emptyset$, and we call $\mathbb{L}_{YES} \cup \mathbb{L}_{NO}$ the *promise*.

Therefore, we are interested in partitioning $\mathbb{L}$ into the two languages then designing an algorithm where the input domain is the *promise*. This partitioning is more of an art than a science; if *promise* is too large, e.g. *promise* $= \mathbb{L}$, then the promise problem becomes trivial and equivalent to a decision problem. Similarly, if *promise* is too small then the algorithm will be trivial and will not be practically useful. This project is the first step towards solving the partitioning problem: we hypothesize that *promise* can be pairs of high-compression strings and $\mathbb{L}_{Dis}$ can be pairs of low-compression strings. The experiments performed show some support for our hypothesis, and we sketch the beginning of further partitioning *promise* into the *YES* and *NO* languages by describing the domain's distribution. This partitioning does not trivialize the algorithm since the domain of high-compression strings is fairly large but still provably smaller than $\mathbb{L}$.

5 EXPERIMENTS

Methodology

The experiments are designed to confirm or reject the hypothesis that there is sufficient separation between high-compression string pairs and low-compression string pairs with regards to the string metric measures. If there is a strong separation, then the two domains should be modeled differently, that is we expect a bi-modal distribution. Let S_1 and S_2 be the two strings under consideration; let $Cr = CR(S_1) + CR(S_2)$ be the total compression ratio of the two strings per Equation 2. Moreover, a metric refers to one of the string algorithms: Edit Distance, LCS, Hamming Distance. Then the hypotheses are formulated as below,

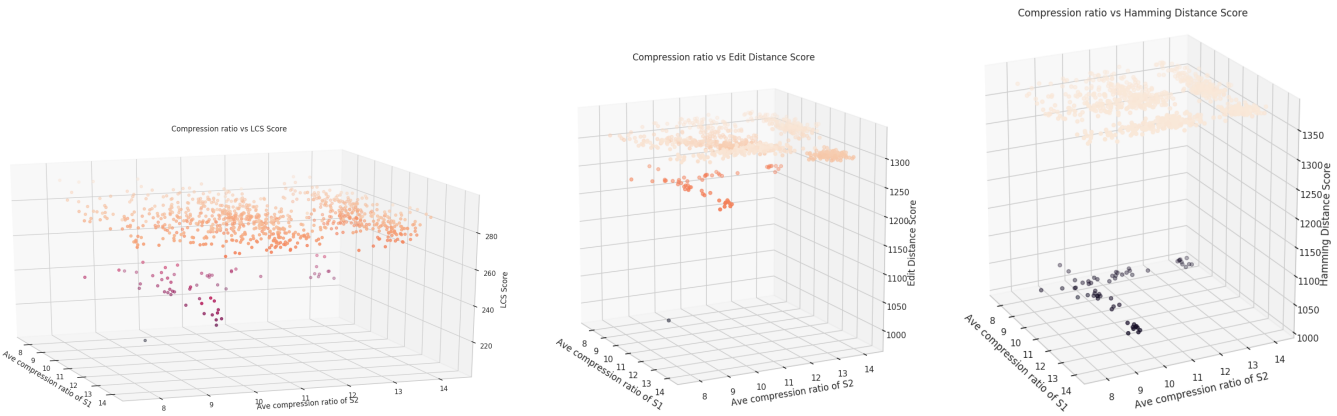
H.1 Cr and each metric are not linearly correlated
H.2 Each metric is not normally distributed

To statistically confirm these hypotheses, we first generate sufficiently large data sample. We consider string pairs with compression ratios between 5 and 20 in increments of 0.5. First we measure the standard deviation between 100 such samples for each compression ratio target for exploratory data analysis. Next, we consider the relationship between Cr and the metrics, and provide a Gaussian Mixture Model (GMM). Listing 1 shows the code used to generate the data.

Listing 1: Python program used to generate sample data for experiments

```
1  def experiment_on_given_CR(CRS):
2    CR1, CR2 = CRs
3    iters = 100
4    results = np.zeros((iters, len(metrics)))
5    compression_ratios = np.zeros((iters, 2))
6    for i in range(max(2, iters)):
7      decompressed = tuple(map(decode,
8                               map(random_compression, CRs)))
9      actual_cr = tuple(map(compression_ratio, decompressed))
10     compression_ratios[i][0], compression_ratios[i][1] =
           actual_cr
11     for index, metric in enumerate(metrics):
12         results[i][index] = metrics[metric](
13             decompressed[0], decompressed[1])
14
15   results = np.std(results, axis=0)
16   compression_ratios = np.mean(compression_ratios, axis=0)
17
18   return np.concatenate((compression_ratios, results))
```

Listing 1 is the core of the code used to generate the data. Line 4 and 5 initialize the Numpy arrays with zeros. The outer loop starting on Line 6 is the repetition loop to strengthen the statistical significance; we set it to 100 due to time and computational power restrictions. Line 7 performs two maps:



(a) LCS **(b) Edit Distance** **(c) Hamming Distance**

Figure 1: 3D visualization of the relationship between compression ratio and three metrics

the inner map calls `random_compression` on the two target compression ratios which produces two random strings with compression ratios close to the target compression; the second map decompresses the returned compressed strings. Line 9 maps `compression_ratio` on the decompressed strings, which calculates the true compression ratios achieved by `random_compression`; Line 10 saves the true compression ratios to the Numpy array. The inner loop starting on Line 11 runs each string metric on the two decompressed strings and saves the values. This process is repeated `iters=100` number of times. Line 15 calculates the standard deviation between the 100 scores of each metric independently. Line 16 averages the achieved compression ratio across the `iters` repetition. These two results form the final returned value. For the second analysis on the raw scores, Line 15 computes the average instead of the standard deviation. For both analyses, `experiment_on_given_CR` is called on $S \times S : S = \{1...20\}$.

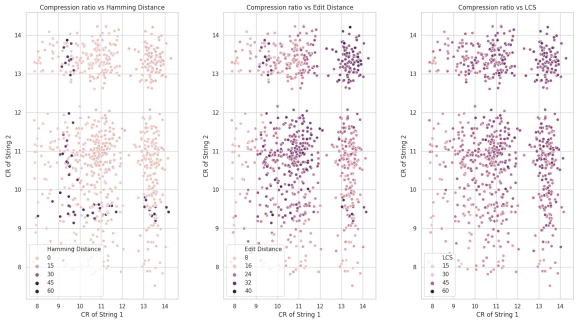


Figure 2: 2D heatmap of compression ratio and standard deviation in metrics scores

Experiment Data and Analysis

The experiments produced a significant amount of data. As described above, one version of the experiment produced standard deviations while the other produced average scores. Listing 2 shows sample data from one of the experiments.

Listing 2: Few enteries of the generated data

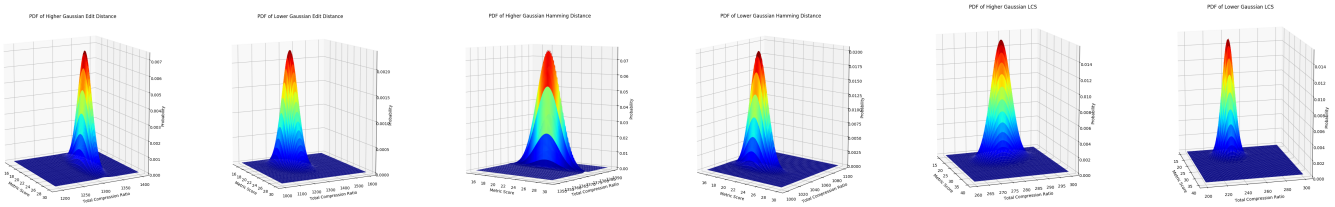
Ave CR1	Ave CR2	Hamming D.	Edit D.	LCS	CR sum
....	....	....	....	....	....
7.66	8.10	1372.89	1306.14	292.62	15.77
7.91	8.76	1372.08	1310.76	291.96	16.67
7.86	9.64	1373.02	1322.23	282.17	17.51
8.20	9.05	1373.80	1321.54	284.31	17.25
8.12	10.68	1373.76	1321.12	286.35	18.80
....	....	....	....	....	....

We reject the null hypothesis of **H.1** by performing Spearman's rank correlation test for each of the metrics and the corresponding compression ratios. A perfect Spearman score of ± 1 implies there is monotonic relationship between the variables, where the sign signifies the direction of the relationship. A perfect, score of 0 implies no tendency for variables to change monotonically. The experiments show scores close to zero as shown in Table 1.

	Spearman	**Shapiro-Wilk**
LCS	0.0111	$9.54e-42$
Hamming	0.0342	$9.54e-42$
Edit	0.0203	$9.54e-42$

Table 1: P-values of significance tests performed

Similarly, we reject the null hypothesis of **H.2** by performing the Shapiro-Wilk test. The p-values are interpreted as usual for $\alpha = 0.05$, and the experiments show p-values close to 0, per Table 1. The experiments continue based on the hypothesis that relationship between Cr and the metrics is not linear and not normally distributed.



(a) Edit Distance **(b) Hamming Distance** **(c) LCS**

Figure 3: 3D PDF of the Gaussian distributions used in the mixture

As seen in Figure 2, when all the available data is plotted with the two compression ratios (note this is not Cr) and metrics scores, the heatmap generally increases towards darker colors as the compression ratios increase. Figure 1 shows this relationship more clearly in a 3D plot; specifically, note the distinct separation between high and low metric scores on the z-axis. This clearly indicates a bimodal distribution.

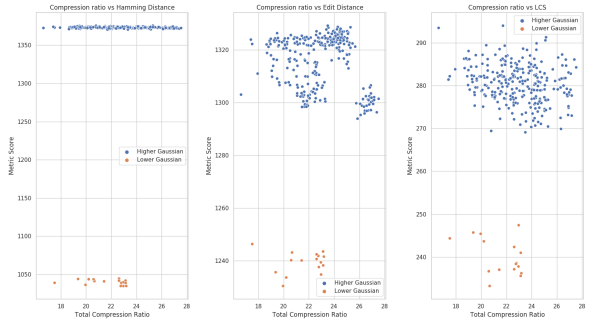


Figure 4: GMM classification of testing data into the two proposed partitions

We formalize this separation as a classification problem. That is, given Cr and the metric score, the classifier decides to put the observation in the higher or lower cluster. Moreover, the distribution in each cluster seems normally distributed so we propose a Gaussian Mixture Model (GMM) as the classifier. A Multivariate GMM is formally,

$$p(x|\theta_k) = \left(\frac{1}{\sqrt{(2\pi)^d |\Sigma_k|}} \right)^{-\frac{1}{2}(x-\mu_x)^T \Sigma_k^{-1}(x-\mu_k)} \tag{7}$$

where $\theta_k = \{\mu_k, \Sigma_k\}$, Σ is the $d \times d$, where d is the dimension of model, covariance matrix, and μ is the vector of means. The covariance matrix needs to be symmetric definite since we need the inverse [20]. GMM is considered an unsupervised learning technique, and the parameters are learned automatically. Expectation maximization is used to estimate the parameters in this case as it is generally effective for GMMs with finite normal components [17]. The components are restricted to two based on the previous analysis and hypotheses. Figure 3 shows the 3D models of the learned distributions for the two components per metric. Similarly, Figure 5 shows the distributions as 2D contour plots.

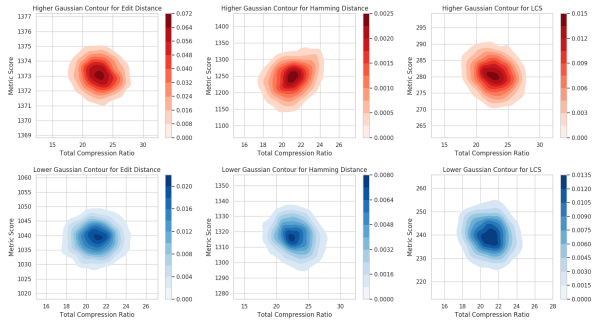


Figure 5: 2D contour plots of the PDF for the Gaussian distributions

Only the estimated μ and Σ for Edit Distance are shown below for brevity. The complete code and results are provided on GitHub [8].

$$\Sigma_1 = \begin{pmatrix} 2.066 & 0.196 \\ 0.196 & 47.132 \end{pmatrix} \Sigma_2 = \begin{pmatrix} 5.206 & -2.106 \\ -2.106 & 19.559 \end{pmatrix} \tag{8}$$

$$\mu_1 = (20.95, 240.257)$$
$$\mu_2 = (22.916, 280.477)$$

	Davies–Bouldin	Silhouette
LCS	0.20	0.84
Hamming	0.01	0.99
Edit	0.16	0.85

Table 2: GMM performance measure summary

The bi-modality of the data can then be confirmed by analyzing the fitness of the GMM models. The Davies-Bouldin and Silhouette clustering tests are performed to accomplish

this. The first measures the distance of objects within the cluster compared to distance between clusters; a lower score means a better cluster. The latter measures the similarity of each object to other objects within the cluster; a perfect +1 indicates the sample is far from neighboring clusters, a 0 indicates being close to the cluster boundaries, and a perfect -1 indicates the object is in the wrong category. As Table 2 shows, both tests indicate the GMM models are very good fits.

6 FUTURE WORK

This project is the first step in a much larger project, thus much of the work remains in the future. However, more specifically to this project, future work should explore more robust multivariate analysis. The analysis in Section 5 collapses the two compression ratios into Cr, which results in significant loss in data resolution. For example, the analysis does not differentiate between a high Cr caused by both compression ratios being high and a high Cr caused by one extremely high compression ratio. This is noticeable in the models since the relationship is not clearly bi-modal on the compression axis (see Figure 4) although the modality is apparent on the compression axes when the compression ratios are separated (see Figure 2). Future work can explore better partitioning based on these models; perhaps by using these models as a priori probability estimators of similarity. This line of work can show a much stronger relationship between the compression ratios and the scores, such that one can classify samples' scores based on the compression ratios.

7 CONCLUSION

This work makes progress in two ways: first, it formalizes the problem as a promise problem, and second it provides a meaningful partitioning of the promise. The experiments show some support for the compressibility based partitioning, however we make reserved claims in how good these partitions are (which might be sufficient for the project to move forward). Additionally, the GMMs provided can be a useful tool in better understanding the partitioning in the future. The extensive background should also be useful to future work as it properly positions the project both in relation to compression and MPC. The larger project of improving the performance of MPC protocols using compression is a promising approach, and this work establishes the value of compression when the bottleneck of performance is the size of the inputs and outputs.

REFERENCES

[1] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 503–513.

[2] Nathan Brunelle. 2017. *Super-Scalable Algorithms*. Ph.D. Dissertation. University of Virginia.

[3] Geoffroy Couteau. 2019. A note on the communication complexity of multiparty computation in the correlated randomness model. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 473–503.

[4] M. Crochemore, G. M. Landau, and M. Ziv-ukelson. 2002. A subquadratic sequence alignment algorithm for unrestricted cost matrices. In *In Symposium of Discrete Algorithms (SODA)*. 679–688.

[5] Garth A Dickie. 2018. Compression-aware partial sort of streaming columnar data. US Patent 9,959,299.

[6] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. 2018. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security* 2, 2-3 (2018), 70–246.

[7] Paolo Ferragina, Igor Nitto, and Rossano Venturini. 2009. On the bit-complexity of Lempel-Ziv compression. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 768–777.

[8] Yonathan Fisseha and Nathaniel Saxe. 2020. compression-aware-algorithms. https://github.com/yonathanF/compression-aware-algorithms.

[9] John Gantz and David Reinsel. 2012. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future* 2007, 2012 (2012), 1–16.

[10] Norjihan Abdul Ghani, Suraya Hamid, Ibrahim Abaker Targio Hashem, and Ejaz Ahmed. 2019. Social media big data analytics: A survey. *Computers in Human Behavior* 101 (2019), 417–428.

[11] Oded Goldreich. 2006. On promise problems: A survey. In *Theoretical computer science*. Springer, 254–290.

[12] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. 2016. Compression-aware in-memory query processing: Vision, system design and beyond. In *Data Management on New Hardware*. Springer, 40–56.

[13] W-J Huang, Nirmal Saxena, and Edward J McCluskey. 2000. A reliable LZ data compressor on reconfigurable coprocessors. In *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No. PR00871)*. IEEE, 249–258.

[14] Ioannis Ioannidis and Ananth Grama. 2003. An efficient protocol for Yao's millionaires' problem. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. IEEE, 6–pp.

[15] N Jesper Larsson and Alistair Moffat. 2000. Off-line dictionary-based compression. *Proc. IEEE* 88, 11 (2000), 1722–1732.

[16] Shun-Dong Li, Yi-Qi Dai, and Qi-You You. 2005. Efficient solution to Yao's Millionaires' problem. *Dianzi Xuebao(Acta Electronica Sinica)* 33, 5 (2005), 769–773.

[17] Todd K Moon. 1996. The expectation-maximization algorithm. *IEEE Signal processing magazine* 13, 6 (1996), 47–60.

[18] Mark Nelson and Jean-Loup Gailly. 1995. The data compression book 2nd edition. *M & T Books, New York, NY* (1995).

[19] ITURM Recommendation. 2009. 1677-1 International Morse Code. *ITUR recommendation, International Telecommunication Union, October* (2009).

[20] Douglas A Reynolds. 2009. Gaussian Mixture Models. *Encyclopedia of biometrics* 741 (2009).

[21] Aly E Salama, Ahmed H Khalil, et al. 2007. Design and implementation of FPGA-based systolic array for LZ data compression. In *2007 IEEE International Symposium on Circuits and Systems*. IEEE, 3691–3695.

[22] Claude Elwood Shannon. 1948. A mathematical theory of communication. *Bell system technical journal* 27, 3 (1948), 379–423.

[23] Li Shundong, Dai Yiqi, and You Qiyou. 2005. Secure multi-party computation solution to Yao's millionaires' problem based on set-inclusion. *Progress in Natural Science* 15, 9 (2005), 851–856.

[24] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE transactions on consumer electronics* 38, 1 (1992), xviii–xxxiv.

[25] Sophia Yakoubov. 2019. A gentle introduction to Yao's Garbled circuits.

[26] Andrew C Yao. 1982. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*. IEEE, 160–164.

[27] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY* 23, 3 (1977), 337–343.

[28] J. Ziv and A. Lempel. 2006. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.* 24, 5 (Sept. 2006), 530–536. https://doi.org/10.1109/TIT.1978.1055934