Chapter 6
Information Processing on Compressed Data

Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto

Abstract We survey our recent work related to information processing on compressed strings. Note that a "string" here contains any fixed-length sequence of symbols and therefore includes not only ordinary text but also a wide range of data, such as pixel sequences and time-series data. Over the past two decades, a variety of algorithms and their applications have been proposed for compressed information processing. In this survey, we mainly focus on two problems: recompression and privacy-preserving computation over compressed strings. Recompression is a framework in which algorithms transform a given compressed data into another compressed format without decompression. Recent studies have shown that a higher compression ratio can be achieved at lower cost by using an appropriate recompression algorithm such as preprocessing. Furthermore, various privacy-preserving computation models have been proposed for information retrieval, similarity computation, and pattern mining.

6.1 Restructuring Compressed Data

Data compression plays a central role in the efficient transmission and storage of data. Recent developments have also shown that data compression is a useful tool for processing highly repetitive data which contains long common substrings. Typical examples of highly repetitive data include collections of genomes taken from similar species and versioned documents. Popular compressors for highly repetitive data include Lempel-Ziv 77 (LZ77) [40], run-length encoded Burrows-Wheeler transform (RLBWT) [8], and grammar-based compression [34]. For each of these compression methods, researchers have developed techniques for operating on compressed data. For example, there are indexes based on LZ77 [37], RLBWT [17], and grammar-based compression [11]. Although recent studies [33, 36, 45] have investigated the fundamentals of these techniques and obtained a unified view of the compressibility of highly repetitive data, each compressed format still has pros and cons that cannot

Y. Takabatake · T. I · H. Sakamoto (✉)
Kyushu Institute of Technology, Kitakyushu, Japan
e-mail: hiroshi@ai.kyutech.ac.jp

N. Katoh et al. (eds.), *Sublinear Computation Paradigm*,
https://doi.org/10.1007/978-981-16-4095-7_6

be ignored in practice. LZ77 usually achieves better compression than other compression methods, the index based on RLBWT (called r-index) supports very fast pattern search, and grammar-based compression is easy to handle in both theory and practice. Thus, in order to take advantage of the virtues of the different compressed formats, it is useful to have algorithms that can efficiently convert one compressed format to another. In this section, we present some examples of these algorithms.

6.1.1 Preliminaries

Let Σ be an ordered alphabet, that is, a set of characters that has a total order. A string over Σ is a sequence of characters chosen from Σ. The length of a string w is denoted by $|w|$. For any $1 \leq i \leq |w|$, the ith character of w is denoted by $w[i]$. The substring of w starting at i and ending at j is denoted by $w[i...j]$. The substring $w[i...j]$ is called a *prefix* (resp., *suffix*) if $i = 1$ (resp., $j = |w|$). The reversed string of w is denoted by w^R, namely, $w^R = w[|w|]w[|w| - 1] \cdots w[2]w[1]$.

Let T be a string of length n over Σ. We consider the following three compression schemes for T.

LZ77: LZ77 is characterized by greedy factorization $T = f_1 f_2 \cdots f_z$ of T. The ith factor f_i is a single character if the character does not appear in $f_1 f_2 \cdots f_{i-1}$, and otherwise, the longest substring such that there is another occurrence s_i of f_i with $s_i \leq |f_1 f_2 \cdots f_{i-1}|$. The position s_i is called the reference position of the ith LZ77 factor f_i. We can store T in $O(z)$-space because each factor f_i (in the second case) can be replaced with a pair $(s_i, |f_i|)$.

BWT, RLBWT: For simplicity, we assume that T is extended by the end marker \$, which is a special character not in Σ and lexicographically smaller than any character in Σ, that is, $T[n + 1] = \$$. The Burrows-Wheeler transform [8] is a permutation L of characters in $T[1 \ldots n + 1]$ obtained as follows: $L[i]$ is the character preceding the lexicographically ith smallest suffix among all non-empty suffixes of T with the exception that $L[i] = \$$ when the ith smallest suffix is T itself (and therefore has no preceding character). The resulting string L can be interpreted as a sequence obtained by sorting characters in T according to their context (succeeding suffixes). Since characters sharing similar context tend to be identical, L is well compressible by run-length encoding. The run-length encoded BWT is called RLBWT.

Let $\mathsf{SA}[1 \ldots n + 1]$ denote the suffix array of $T[1 \ldots n + 1]$, where $\mathsf{SA}[i]$ is the starting position of the lexicographically ith smallest suffix. We consider SA as a mapping from BWT position to text position and say that the BWT position i corresponds to the text position $\mathsf{SA}[i]$. One crucial operation on the BWT string L is the so-called LF mapping that maps a BWT position i to the BWT position corresponding to text position $\mathsf{SA}[i] - 1$. LF mapping can be implemented by a rank data structure on L that returns the number of occurrences of a character c in $L[1 \ldots i]$ for any character c and BWT position i.

By using LF mapping, we can also support backward search. For any string w that appears in T, there is a unique maximal interval $[b \ldots e]$ such that the lexico-

graphically ith suffix is prefixed by w iff $i \in [b \dots e]$. Note that $e - b + 1$ is the number of occurrences of w in T and the text positions corresponding to these positions represent the occurrences of w. A single step of the backward search computes the cw-interval from the w-interval by using the same mechanism as LF mapping, where c is a character. The index based on backward search on BWT is known as the FM-index [14]. Although it was previously known that the occurrences of a pattern can be counted by a backward search implemented in RLBWT space [41], it was recently reported that RLBWT can be augmented with an $O(r)$-space data structure to report all the occurrences of the pattern efficiently. The index based on RLBWT is called the r-index [17].

Grammar compression: Grammar compression is a general framework of data compression in which a context-free grammar (CFG) $\mathcal{S} = (\Sigma, \mathcal{V}, \mathcal{D})$ that derives a single string T is considered to be a compressed representation of T, where Σ is the set of characters (terminals), $\mathcal{V}$ is the set of variables (non-terminals), $\mathcal{D}$ is the set of deterministic production rules whose right-hand sides are strings over $(\mathcal{V} \cup \Sigma)$, and the last variable derives T.[1] The compressed size of $\mathcal{S}$ is expressed by the sum of the lengths of right-hand sides of the production rules in $\mathcal{S}$. We consider run-length encoding right-hand sides of CFGs, and call such CFGs *run-length encoded CFGs (RLCFGs)*. The compressed size of an RLCFG is expressed by the sum of run-length encoded sizes of right-hand sides of the production rules.

Algorithm 1: Supposing that we have parsed suffix $T[p+1 \dots]$, compute the length of the next LZ77 factor ending at p.

```
1 p' ← p;
2 w ← ε;
3 c ← T[p'];
4 while cw-interval contains a text position larger than p' do
5 |   p' ← p' − 1;
6 |   w ← cw;
7 |   c ← T[p'];
8 return min(1, p' − p);
```

6.1.2 RLBWT to LZ77

Algorithms to compute LZ77 from RLBWT are considered in [3, 32, 46, 47, 49]. An essential task when computing LZ77 is to search for the longest prefix of $T[|f_1 f_2 \cdots f_{i-1}| + 1 \cdots]$ that occurs before and compute an occurrence $s_i \leq |f_1 f_2 \cdots f_{i-1}|$ of f_i. The basic idea is to use the backward search on RLBWT of T^R to perform this task. One difficulty is ignoring the BWT positions that correspond

[1] We treat the last variable as the starting variable.

to the suffixes starting after $|f_1 f_2 \cdots f_{i-1}|$ during the backward search. In [49], it is shown that keeping at most $2r$ BWT positions is sufficient to compute the longest prefix and a reference position for the LZ77 factor. This subsection gives a brief review of this idea.

For the sake of this explanation, consider the case of LZ77 parsing from right to left (i.e., we conceptually compute the LZ77 factorization for T^R) so that backward search on T (instead of the reversed one) can be used. Supposing that we have parsed suffix $T[p+1\ldots]$, Algorithm 1 shows how to compute the length of the next factor ending at p. To check whether the cw-interval contains a text position larger than p', we partition SA into r subintervals and maintain at most two positions for each subinterval, which is the LF-mapped interval of a run of L. Suppose that the cw-interval $[b\ldots e]$ is non-empty and $[b\ldots e]$ is covered by consecutive subintervals $[b_1\ldots e_1], [b_2\ldots e_2], \ldots, [b_k\ldots e_k]$ with minimal integer k, that is, $b_1 \le b < e_1 + 1 = b_2 < e_2 + 1 = b_3 < \cdots < e_{k-1} + 1 = b_k \le e \le e_k$. If $k = 1$, the characters of L in w-interval consist of a single character c and all positions in w-interval are LF-mapped to cw-interval. Therefore, cw-interval contains a text position larger than p' iff w-interval satisfies the condition in the previous step. For the case of $k > 1$, we mark the closest positions from the boundaries of subintervals that correspond to text positions larger than p'. Using this information, we can check whether $\mathsf{SA}[b_1\ldots e_1]$ and/or $\mathsf{SA}[b_k\ldots e_k]$ contain a text position larger than p'. We also maintain the data structure to check whether a subinterval in $[b_2\ldots e_2], \ldots, [b_{k-1}\ldots e_{k-1}]$ contains a text position larger than p', and if so we compute which interval contains that position.

In this way, we can compute the lengths of LZ77 factors. The reference position for each LZ77 factor can also be computed by maintaining text positions corresponding to the marked positions in each subinterval. The data structures use only $O(r)$ words of space.

In [46], the data structures are tuned to improve the time complexity. In [47], a fast implementation for the backward search in RLBWT space was proposed and applied to the above-mentioned algorithm. In [3], an online construction of r-index was proposed and the technique was extended to an online LZ77 factorization algorithm in RLBWT space. In [32], a different approach to converting RLBWT to LZ77 was proposed.

6.1.3 Recompression on Grammar Compression

Given that there are a number of CFGs with different properties for representing strings, we may want to transform one CFG to another without explicitly decompressing the text. In this subsection, we introduce a technique called recompression which has proven to be a powerful tool in problems related to grammar compression [26–28, 31] and word equations [29, 30].

In [27], Jeż proposed an algorithm TtoG for computing an RLCFG of T in $O(N)$ time. Let TtoG(T) denote the RLCFG of T produced by TtoG. We use the term *letters* for characters and variables introduced by TtoG. A run is called a *block* in this subsection. TtoG consists of two different types of compression, namely, block compression (BComp) and pair compression (PComp).

- BComp: Given a string w over $\Sigma = [1 \dots |w|]$, BComp compresses w by replacing all blocks of length ≥ 2 with fresh letters. Note that BComp eliminates all blocks of length ≥ 2 in w.
- PComp: Given a string w over $\Sigma = [1 \dots |w|]$ that contains no block of length ≥ 2, PComp compresses w by replacing all pairs from $\acute{\Sigma}\grave{\Sigma}$ with fresh letters, where $(\acute{\Sigma}, \grave{\Sigma})$ is a partition of Σ, that is, $\Sigma = \acute{\Sigma} \cup \grave{\Sigma}$ and $\acute{\Sigma} \cap \grave{\Sigma} = \emptyset$. Given the frequency table of pairs, we can deterministically compute a partition of Σ by which at least $(|w| - 1)/4$ occurrences of pairs are replaced.

TtoG compresses $T_0 = T$ by applying BComp and PComp in turns until the string is shrunk down to a single letter. Because PComp compresses a given string by a constant factor of $3/4$, the height of TtoG(T) is $O(\lg N)$.

TtoG performs level-by-level transformation of T_0 into strings $T_1, T_2, \dots, T_{\hat{h}}$, where $|T_{\hat{h}}| = 1$. If h is even, the transformation from T_h to T_{h+1} is performed by BComp, and production rules of the form $c \to \ddot{c}^d$ are introduced. If h is odd, the transformation from T_h to T_{h+1} is performed by PComp, and production rules of the form $c \to \acute{c}\grave{c}$ are introduced. Let Σ_h be the set of letters appearing in T_h.

The advantage of TtoG is that it can be simulated on $\mathcal{S} = \mathcal{S}_0 = (\Sigma_0, \mathcal{V}, \mathcal{D}_0)$ without decompression. We consider the level-by-level transformation of $\mathcal{S}_0$ into CFGs $\mathcal{S}_1 = (\Sigma_1, \mathcal{V}, \mathcal{D}_1), \mathcal{S}_2 = (\Sigma_2, \mathcal{V}, \mathcal{D}_2), \dots, \mathcal{S}_{\hat{h}} = (\Sigma_{\hat{h}}, \mathcal{V}, \mathcal{D}_{\hat{h}})$, where each $\mathcal{S}_h$ generates T_h. More specifically, the compression from T_h to T_{h+1} is simulated on $\mathcal{S}_h$. We can correctly compute the letters introduced in each level $h + 1$ while modifying $\mathcal{S}_h$ into $\mathcal{S}_{h+1}$; hence, we get all the letters of TtoG(T) in the end. We note that new "variables" are never introduced and modifications are made by rewriting the right-hand sides of the original variables.

We now show how PComp is performed on $\mathcal{S}_h$ for odd h. That is, we compute $\mathcal{S}_{h+1}$ from $\mathcal{S}_h$. Note that any occurrence i of a pair $\acute{c}\grave{c}$ in T_h can be uniquely associated with a variable X that is the label of the lowest node covering the interval $[i \dots i + 1]$ in the derivation tree of $\mathcal{S}_h$ (recall that $\mathcal{S}_h$ generates T_h). We can compute the frequency table of pairs by counting pairs associated with X in $\mathcal{D}_h(X)$ and multiplying it by the number of occurrences of X in the derivation tree of $\mathcal{S}_h$. The frequency table is used to compute a partition of Σ_h, which determines the pairs to be replaced. A pair appears *explicitly* in right-hand sides or *crosses* the boundaries of variables. We can modify $\mathcal{S}_h$ so that all the crossing occurrences to be replaced appear explicitly in some right-hand side, then replace the explicit occurrences to get $\mathcal{S}_{h+1}$. In a similar way, BComp can also be performed on $\mathcal{S}_h$ for odd h.

In [23], it is shown that TtoG(T) can be used to answer the longest common extension (LCE) queries and the transformation from arbitrary CFG $\mathcal{S}$ to TtoG(T) is a key for efficient construction algorithms of LCE data structures in grammar compressed space. In [53], the recompression technique is modified to transform

arbitrary CFG $\mathcal{S}$ into the CFG obtained by the RePair algorithm [38]. RePair is known to achieve the best compression performance in practice and there are many studies on computing RePair in small space. Using *online* grammar compression algorithms, such as [43, 57], the algorithm in [53] leads to the first RePair algorithm working in compressed space.

6.2 Privacy-Preserving Similarity Computation

6.2.1 Related Work

This section reviews recent results in privacy-preserving information retrieval over strings recently presented in [59]. As the number of strings containing personal information has increased, privacy-preserving computation has become increasingly important. Secure computation based on public-key encryption is one of the great accomplishments of modern cryptography because it allows untrusted parties to compute a function based on their private inputs, while revealing nothing but the result.

Rapid progress in gene sequencing technology has expanded the range of applications of edit distance to include personalized genomic medicine, diagnosis of diseases, and preventive treatment (e.g., see [1]). However, because the genome of a person is ultimately personal information that uniquely identifies the owner, the parties involved should not share personal genomic data in plaintext. We therefore consider a secure two-party model for edit distance computation: Two untrusted parties generating their own public and private keys have strings x and y, respectively, and they want to jointly compute $f(x, y)$ for a given metric f without revealing anything about their individual strings.

Homomorphic encryption (HE) is an emerging technique for such secure multiparty computation. HE is a kind of public-key encryption between two parties Alice and Bob where Bob wants to send a secret message to Alice. In this model, Bob generates his secret key and public key prior to communication, say sk and pk, where pk is known to everyone. Alice then sends the encrypted message $E(m, pk)$ to Bob and he decrypts m by using his secret key sk using the property $E(E(m, pk), sk) = m$. If it is not necessary to specify the owner of pk and sk, we simply write $E(m)$ for simplicity.

A public-key encryption $E()$ has the additive homomorphic property if we can obtain $E(m + n)$ from $E(m)$ and $E(n)$ without decryption, and the multiplicative property is similarly defined. If $E()$ is additive, Alice can obtain the summation of many people's secret numbers without revealing their private numbers.

The first public-key encryption algorithm RSA [51] is multiplicative because it has the following property: Let (e, n) be a public key and (d, n) be a secret key, respectively, where e, d, n are integers. For a message m, its encryption is computed by $c = (m^e \mod n)$ and is decrypted by $c^d = m^{ed} \equiv m \mod n$. We can easily

check the multiplicative property $(m_1^e \mod n) \cdot (m_2^e \mod n) = (m_1 m_2)^e \mod n$. The Paillier encryption system [48] was the first system to have the additive property. This means that parties can jointly compute the encrypted value $E(x + y)$ directly based on only two encrypted integers $E(x)$ and $E(y)$.

By taking advantage of the homomorphic property, researchers have proposed HE-based privacy-preserving protocols for computing the Levenshtein distance $d(x, y)$. For example, Inan et al. [25] designed a three-party protocol where two parties securely compute $d(x, y)$ by enlisting the help of a reliable third party. Rane and Sun [50] then improved this three-party protocol to develop the first two-party protocol.

In this review, we focus on an extended Levenshtein distance called the *edit distance with moves* (EDM) which allows any substring to be moved with unit cost in addition to the standard operations of inserting, deleting, and replacing a character. Based on the EDM, we can find a set of approximately maximal common substrings appearing in two strings, which can be used to detect plagiarism in documents or long repeated segments in DNA sequences. As an example, consider two unambiguously similar strings $x = a^N b^N$ and $y = b^N a^N$, which can be transformed into each other by a single move. While the exact EDM is simply $\mathrm{EDM}(x, y) = 1$, the Levenshtein distance has the undesirable value $d(x, y) = 2N$. The n-gram distance is preferable to the Levenshtein distance in this case, but it requires huge time/space complexity depending on N.

Although computation of $\mathrm{EDM}(x, y)$ is NP-hard [55], Cormode and Muthukrishnan [12] were able to find an almost linear-time approximation algorithm. Many techniques have been proposed for computing the EDM. For example, Ganczorz et al. [18] proposed a lightweight probabilistic algorithm. In these algorithms, each string x is transformed into a characteristic vector v_x consisting of nonnegative integers representing the frequencies of particular substrings of x. For two strings x and y, we then have the approximate distance guaranteeing $L_1(v_x, v_y) = O(\lg^* N \lg N)\mathrm{EDM}(x, y)$ for $N = |x| + |y|$.

In Appendix A of [15], there is a subtle flaw in the ESP algorithm [12] that achieves this $O(\lg^* N \lg N)$ bound. However, this flaw can be remedied by an alternative algorithm called HSP [15]. Because $\lg^* N$ increases extremely slowly,[2] we employ $L_1(v_x, v_y)$ as a reasonable approximation to $\mathrm{EDM}(x, y)$.

Basically, the ESP tree is a special type of grammar compression referred to in the previous section where the length of the right-hand side of any production rule is just two or three. Therefore, $\mathrm{EDM}(x, y)$ is approximated by the compressed expressions for the strings x and y. The relationship between grammar compression (including ESP) and its applications has been widely investigated in the past two decades (see, e.g., [10, 21, 24, 39, 42, 52, 54, 56–58]).

[2] $\lg^* N$ is the number of times the logarithm function lg can be iteratively applied to N until $\lg^* N \leq 1$.

Recently, Nakagawa et al. proposed the first secure two-party protocol for EDM (sEDM) [44] based on HE. However, their algorithm suffers from a bottleneck during the step where the parties construct a shared labeling scheme. Yoshimoto improved the previous algorithm to make it easier to use in practice [59]. We review the practical algorithm here.

6.2.2 Edit Distance with Moves

Based on the notation for strings in the previous section, $\mathrm{EDM}(S, S')$ is the length of the shortest sequence of edit operations that transforms S into S', where the permitted operations (each having unit cost) are inserting, deleting, or renaming one symbol at any position, or moving an arbitrary substring. Unfortunately, as Theorem 6.1 states, computing $\mathrm{EDM}(S, S')$ is NP-hard even if the renaming operations are not allowed [55], so we focus on an approximation algorithm for EDM, called Edit-Sensitive Parsing (ESP) [12].

Theorem 6.1 (Shapira and Storer [55]) *Determining* $\mathrm{EDM}(x, y)$ *is NP-hard even if only three unit-cost operations are allowed, namely, inserting a character, deleting a character, and moving a substring.*

ESP constructs a parsing tree, called an ESP tree, for a given string S, where internal nodes are labeled *consistently*, that is, internal nodes have a common name if and only if they derive the same string. After two ESP trees T_S and $T_{S'}$ are constructed for given strings S and S' for comparison in EDM, the characteristic vectors v_S and $v_{S'}$ are defined such that $v_S[i]$ is the frequency of the ith label in T_S. $EDM(S, S')$ is then approximated by $L_1(v_S, v_{S'})$ with the following lower/upper bounds.

Theorem 6.2 (Cormode and Muthukrishnan [12]) *Let* T_S *and* $T_{S'}$ *be consistently labeled ESP trees for* $S, S' \in \Sigma^*$*, and let* v_S *be the characteristic vector for* S*, where* $v_S[k]$ *is the frequency of label* k *in* T_S*. Then,*

$$\frac{1}{2}\mathrm{EDM}(S, S') \leq L_1(v_S, v_{S'}) = O(\lg^* N \lg N)\mathrm{EDM}(S, S')$$

for $L_1(v_S, v_{S'}) = \sum_{i=1}^{k} |v_S[i] - v_{S'}[i]|$.

In Fig. 6.1, we illustrate an example of consistent labeling of the trees T_S and $T_{S'}$ together with the resulting characteristic vectors. Since the strings S and S' are parsed offline, the problem of preserving privacy is reduced to designing a secure protocol for creating consistent labels and computing the L_1-distance between the trees.

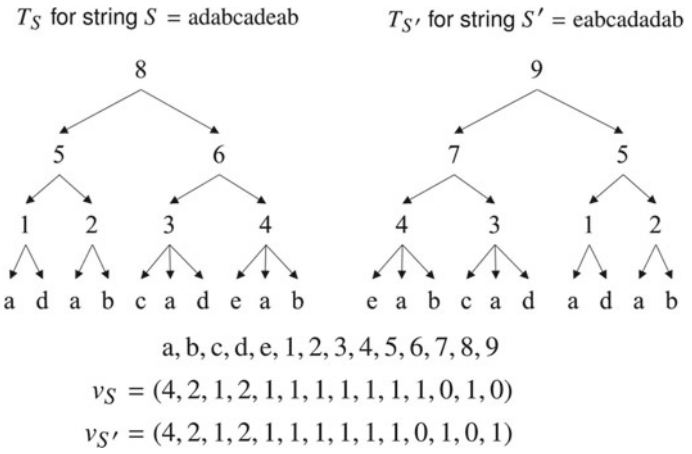

Fig. 6.1 Example of approximate EDM. For strings $S = adabcadeab$ and $S' = eabcadadab$, S is transformed into S' by two moves of substrings, that is, $\mathrm{EDM}(S, S') = 2$. After constructing ESP trees T_S and $T_{S'}$ with consistent labeling, the corresponding characteristic vectors v_S and $v_{S'}$ are computed offline. The exact $\mathrm{EDM}(S, S')$ is approximated by $L_1(v_S, v_{S'}) = 4$

6.2.3 Homomorphic Encryption

We now briefly review the framework of homomorphic encryption. Let (pk, sk) be a key pair for a public-key encryption scheme, and let $E_{pk}(x)$ be the encrypted value of a message x and $D_{sk}(C)$ be the decrypted value of a ciphertext C, respectively. We say that the encryption scheme is *additively homomorphic* if we have the following properties: (1) There is an operation $h_+(\cdot, \cdot)$ for $E_{pk}(x)$ and $E_{pk}(y)$ such that $D_{sk}(h_+(E_{pk}(x), E_{pk}(y))) = x + y$. (2) For any r, we can compute the scalar multiplication such that $D_{sk}(r \cdot E_{pk}(x)) = r \cdot x$.

An additive homomorphic encryption scheme that allows a sufficient number of these operations is called an additive HE.[3] Paillier's encryption scheme [48] is the first secure additive HE. However, there are not many functions that can be evaluated by using only additive homomorphism and scalar multiplication.

The multiplication $D_{sk}(h_\times(E_{pk}(x), E_{pk}(y))) = x \cdot y$ is another important homomorphism. If we allow both additive and multiplicative homomorphism as well as scalar multiplication (called a fully homomorphic encryption, FHE [19] for short), it follows that we can perform any arithmetic operation on ciphertexts. For example, if we can use sufficiently large number of additive operations and a single multiplicative operation over ciphertexts, we obtain the inner-product of two encrypted vectors.

However, there is a trade-off between the available homomorphic operations and their computational cost. To avoid this difficulty, we focus on leveled HE (LHE) where the number of homomorphic multiplications is restricted beforehand. In particular,

[3] In general, the number of applicable operations over ciphertexts is bounded by the size of (pk, sk).

L2HE (Additive HE that allows a single homomorphic multiplication) has attracted a great deal of attention. The BGN encryption system is the first L2HE and was invented by Boneh et al. [6] by assuming a single multiplication and sufficient numbers of additions. Using BGN, we can securely evaluate formulas in disjunctive normal form. Following this pioneering study, many practical L2HE protocols have been proposed [2, 9, 16, 22].

In terms of EDM computation, although Nakagawa et al. [44] introduced an algorithm for computing the EDM based on L2HE, their algorithm is very slow for large strings. Following on from this work, Yoshimoto et al. proposed another novel secure computation of EDM for large strings based on the faster L2HE proposed by Attrapadung et al. [2]. To our knowledge, there is no secure two-party protocol for EDM computation that uses only the additive homomorphic property. Whether we can compute EDM using a two-party protocol based on additive HE alone is an interesting question.

For the benefit of the reader, we give a simple review of the mechanism used by BGN, the first L2HE. For plaintexts $m_1, m_2 \in \{1, \ldots, M\}$ and their corresponding ciphertexts C_1 and C_2, the ciphertexts of $m_1 + m_2$ and $m_1 m_2$ can be computed directly from C_1 and C_2 without decrypting m_1 and m_2, provided $m_1 + m_2, m_1 m_2 \leq M$.

For large primes q_1 and q_2, the BGN encryption scheme is based on two multiplicative cyclic groups $\mathbb{G}$ and $\mathbb{G}'$ of order $q_1 q_2$, two generators g_1 and g_2 of $\mathbb{G}$, an inverse function $(\cdot)^{-1} : \mathbb{G} \to \mathbb{G}$, and a bihomomorphism $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}'$. By definition, $e(\cdot, x)$ and $e(x, \cdot)$ are group homomorphisms for all $x \in \mathbb{G}$. In addition, we assume that both the inverse function $(\cdot)^{-1}$ and the bihomomorphism e can be computed in polynomial time in terms of the security parameter $\log_2 q_1 q_2$. Such a system $(\mathbb{G}, \mathbb{G}', g_1, g_2, (\cdot)^{-1}, e)$ can be generated by, for example, letting $\mathbb{G}$ be a subgroup of a supersingular elliptic curve and e be a Tate pairing [6]. The BGN encryption scheme proceeds as follows.

Key generation: Randomly generate two sufficiently large primes q_1 and q_2, then use these to define $(\mathbb{G}, \mathbb{G}', g_1, g_2, (\cdot)^{-1}, e)$ as described above. Choose two random generators g and u of $\mathbb{G}$, set $h = u^{q_2}$, and let M be a positive integer bounded above by a polynomial function of the security parameter $\log_2 p_1 p_2$. The public key is then $pk = (p_1 p_2, \mathbb{G}, \mathbb{G}', e, g, h, M)$ and the private key is $sk = q_1$.

Encryption: Encrypt the message $m \in \{0, \ldots, M\}$ using pk and a random $r \in \mathbb{Z}_n$ to $C = g^m h^r \in \mathbb{G}$ yielding the ciphertext C.

Decryption: Find the integer m such that $C^{q_1} = (g^m h^r)^{q_1} = (g^{q_1})^m$ using a polynomial time algorithm. There is a known algorithm for this with time complexity of $O(\sqrt{M})$.

Homomorphic properties: For the ciphertexts $C_1 = g^{m_1} h^{r_1}$ and $C_2 = g^{m_2} h^{r_2}$ in $\mathbb{G}$ corresponding to the messages m_1 and m_2, anyone can calculate the encrypted value of $m_1 + m_2$ and $m_1 m_2$ directly from C_1 and C_2 without knowing m_1 and m_2, as follows.

– Additive homomorphism:

$$C_a = C_1 C_2 h^r = (g^{m_1} h^{r_1})(g^{m_2} h^{r_2}) h^r = g^{m_1 + m_2} h^{r_1 + r_2 + r}$$

gives the encrypted value of $m_1 + m_2$.

– Multiplicative homomorphism: $C_m = e(C_1, C_2)h^r \in \mathbb{G}'$ gives the encrypted value of $m_1 m_2$, because

$$\begin{aligned} C_m^{q_1} &= e(C_1, C_2) \\ &= \left[e(g_1, g_2)^{m_1 m_2} e(g_1, g_2)^{q_2 m_1 r_1} e(g_2, g_1)^{q_2 m_2 r_1} e(g_1, g_2)^{q_2 r}\right]^{q_1} \\ &= \left(e(g_1, g_2)^{q_1}\right)^{m_1 m_2}, \end{aligned}$$

where we decrypt C_m, by computing $m_1 m_2$ from $(g(g_1, g_2)^{q_1})^{m_1 m_2}$ and $e(g_1, g_2)^{q_1}$.

Note that $C_1, C_2 \in \mathbb{G}'$ also have additive homomorphic properties, so BGN allows a single multiplication and unlimited additions over ciphertexts.

6.2.4 L2HE-Based Algorithm for Secure EDM

We now explain the algorithm for computing approximate EDM based on L2HE [59]. Two parties $\mathcal{A}, \mathcal{B}$ have strings $S_\mathcal{A}, S_\mathcal{B}$, respectively. First, they compute the corresponding ESP trees $T_\mathcal{A}$ and $T_\mathcal{B}$ offline and they assign tentative labels to internal nodes of $T_\mathcal{A}$ and $T_\mathcal{B}$ using a hash function $h : X \to \{1, 2, \ldots, n\}$ for $X \subseteq \{0, 1, \ldots, m\}$ of n different labels in $T_\mathcal{A}$ and $T_\mathcal{B}$ with a fixed m. The goal is to securely relabel X using a bijection: $X \to \{1, 2, \ldots, n\}$, as described in Algorithm 2. We suppose that $\mathcal{A}$ and $\mathcal{B}$ generate their own public and private keys prior to the computation.

In Algorithm 2, we assume an L2HE scheme allowing a single multiplicative operation and a sufficient number of additive operations over encrypted integers. Because these operations are usually implemented by AND ($\cdot$) and XOR ($\oplus$) logic gates (e.g., [7]), we introduce the following notation for these gates. First, $E_\mathcal{A}(x)$ denotes the ciphertext generated by encrypting plaintext x with $\mathcal{A}$'s public key, and $E_\mathcal{A}(x, y, z)$ is an abbreviation for the vector $(E_\mathcal{A}(x), E_\mathcal{A}(y), E_\mathcal{A}(z)$. Here, $E_\mathcal{A}(x, y, z) \cdot E_\mathcal{A}(a, b, c)$ denotes $(E_\mathcal{A}(x \cdot a), E_\mathcal{A}(y \cdot b), E_\mathcal{A}(z \cdot c))$ and $E_\mathcal{A}(x, y, z) \oplus E_\mathcal{A}(a, b, c)$ denotes $(E_\mathcal{A}(x \oplus a), E_\mathcal{A}(y \oplus b), E_\mathcal{A}(z \oplus c))$ for each bit $x, y, z, a, b, c \in \{0, 1\}$. Using this notation, we describe the proposed protocol in Algorithm 2.

Next, we define the protocol security based on a model where both parties are assumed to be *semi-honest*, that is, corrupt parties merely cooperate to gather information out of the protocol, but do not deviate from the protocol specification. The security is defined as follows.

Definition 6.1 (*S*emi-honest security [20]) A protocol is secure against semi-honest adversaries if each party's observation of the protocol can be simulated using only the input they hold and the output that they receive from the protocol.

Intuitively, this definition tells us that a corrupt party is unable to learn any extra information that cannot be derived from the input and output explicitly (for details, see [20]). Under this assumption, since the algorithm is symmetric with respect

Algorithm 2 for consistently labeling $T_{\mathcal{A}}$ and $T_{\mathcal{B}}$ [59]

Preprocessing (tentative labeling): Parties $\mathcal{A}$ and $\mathcal{B}$ agree to use a hash function H with a range $\{0, \ldots, m\}$ for sufficiently large m. Both parties compute $T_{\mathcal{A}}$ and $T_{\mathcal{B}}$ corresponding to their respective strings offline. The label of internal node u is assigned $H(s(u))$ where $s(u)$ is the string of all leaves of u. Now, parties $\mathcal{A}$ and $\mathcal{B}$ have tentative label sets $[T_{\mathcal{A}}], [T_{\mathcal{B}}] \subseteq \{0, \ldots, m\}$, respectively.

Goal: Change all the labels using a bijection: $[T_{\mathcal{A}}] \cup [T_{\mathcal{B}}] \to \{1, \ldots, n\}$ without either party having to reveal anything about their private strings.

Notation: $E_{\mathcal{A}}(x)$ denotes the ciphertext of a message x encrypted by an L2HE with $\mathcal{A}$'s public key.

Sharing a dictionary:
Step 1: Party $\mathcal{A}$ computes the bit vector $\mathbf{X}[1 \ldots m]$ such that $\mathbf{X}[\ell] = 1$ iff $\ell \in [T_{\mathcal{A}}]$. Similarly, party $\mathcal{B}$ computes $\mathbf{Y}[1 \ldots m]$ such that $\mathbf{Y}[\ell] = 1$ iff $\ell \in [T_{\mathcal{B}}]$.
Step 2: $\mathcal{A}$ sends $E_{\mathcal{A}}(\mathbf{X})$ to $\mathcal{B}$ and $\mathcal{B}$ sends $E_{\mathcal{B}}(\mathbf{Y})$ to $\mathcal{A}$.
Step 3: $\mathcal{B}$ computes $(E_{\mathcal{A}}(\mathbf{X}) \oplus E_{\mathcal{A}}(\mathbf{Y})) \oplus (E_{\mathcal{A}}(\mathbf{X}) \cdot E_{\mathcal{A}}(\mathbf{Y})) = E_{\mathcal{A}}(\mathbf{X} \cup \mathbf{Y})$ and $\mathcal{A}$ computes $(E_{\mathcal{B}}(\mathbf{X}) \oplus E_{\mathcal{B}}(\mathbf{Y})) \oplus (E_{\mathcal{B}}(\mathbf{X}) \cdot E_{\mathcal{B}}(\mathbf{Y})) = E_{\mathcal{B}}(\mathbf{X} \cup \mathbf{Y})$.

Relabeling $[T_{\mathcal{A}}]$ **using** $E_{\mathcal{A}}(\mathbf{X} \cup \mathbf{Y})$ ($[T_{\mathcal{B}}]$ is relabeled in a symmetrical fashion)
Step 4: $\mathcal{A}$ computes $E_{\mathcal{B}}(L_\ell) = E_{\mathcal{B}}\left(\sum_{i=1}^{\ell} (\mathbf{X} \cup \mathbf{Y})[i]\right)$ for all $\ell \in [T_{\mathcal{A}}]$.
Step 5: $\mathcal{A}$ sends all $E_{\mathcal{B}}(L_\ell + r_\ell)$ to $\mathcal{B}$ choosing r_ℓ uniformly at random from $\mathbb{N}$.
Step 6: $\mathcal{B}$ decrypts all $L_\ell + r_\ell$ and sends them back to $\mathcal{A}$.
Step 7: $\mathcal{A}$ recreates $L_\ell \in \{1, \ldots, n\}$ for all $\ell \in [T_{\mathcal{A}}]$ by subtracting r_ℓ.

to $\mathcal{A}$ and $\mathcal{B}$, the following theorem proves the security of our algorithm's against semi-honest adversaries.

Theorem 6.3 (Yoshimoto et al. [59]) *Let* $[T_{\mathcal{A}}]$ *be the set of labels appearing in* $T_{\mathcal{A}}$. *The only knowledge that a semi-honest* $\mathcal{A}$ *can gain by executing Algorithm 2 is the distribution of the labels* $\{L_\ell \mid \ell \in [T_{\mathcal{A}}]\}$ *over* $[1, \ldots, n]$.

Theorem 6.4 (Yoshimoto et al. [59]) *Algorithm 2 assigns consistent labels using the injection:* $[T_{\mathcal{A}}] \cup [T_{\mathcal{B}}] \to \{1, 2, \ldots, n\}$ *without revealing the parties' private information. It has round and communication complexities of* $O(1)$ *and* $O(\alpha(n \lg n + m + rn))$, *respectively, where* $n = |[T_{\mathcal{A}}] \cup [T_{\mathcal{B}}]|$, m *is the modulus of the rolling hash used for preprocessing,* $r = \max\{r_1, \ldots, r_n\}$ *is the security parameter, and* α *is the cost of executing a single encryption, decryption, or homomorphic operation.*

Table 6.1 Comparison of the communication and round complexities of secure EDM computation models [44, 59] as well as a naive algorithm. Here, N is the total length of both parties' input strings, n is the number of characteristic substrings determining the approximate EDM, and m is the range of the rolling hash $H(\cdot)$ for the substrings satisfying $m > n$. "Naive" is the baseline method that uses $H(\cdot)$ as the labeling function for the characteristic substrings

Method	#Communication	#Round
Naive	$O(m \lg m)$	$O(1)$
Nakagawa et al. [44]	$O(n \lg n)$	$O(\lg N)$
Yoshimoto et al. [59]	$O(n \lg n + m)$	$O(1)$

6.2.5 Result and Open Question

The complexities of related algorithms are summarized in Table 6.1. Computing the approximate EDM involves two phases: the shared labeling of characteristic substrings (Phase 1) and the L_1-distance computation of characteristic vectors (Phase 2).

Let the parties have strings x and y, respectively. In the offline case (i.e., there is no need for privacy-preserving communication), they construct the respective parsing trees T_x and T_y by the bottom-up parsing called ESP [12], where the node labels must be *consistent*, meaning that two labels are equal if they correspond to the same substring. In such an ESP tree, a substring derived by an internal node is called a characteristic substring. In a privacy-preserving model, the two parties need to jointly compute these consistent labels without revealing whether a characteristic substring is common to both of them (Phase 1). After computing all the labels in T_x and T_y, they jointly compute the L_1-distance of two characteristic vectors containing the frequencies of all labels in T_x and T_y (Phase 2).

As reported in [44], a bottleneck exists in Phase 1. The task is to design a bijection $f : X \cup Y \to \{1, 2, \ldots, n\}$ where X and Y ($|X \cup Y| = n$) are the sets of characteristic substrings for the parties, respectively. Since X and Y are computable without communication, the goal is to jointly compute $f(w)$ for any $w \in X$ without revealing whether $w \in Y$. This problem is closely related to the private set operation (PSO) where parties possessing their private sets want to obtain the results for several set operations, such as intersection or union. Applying the Bloom filter [5] and HE techniques, various protocols for PSO have been proposed [4, 13, 35]. However, these protocols are not directly applicable to our problem because they require at least three parties for the security constraints. In contrast, the algorithm reviewed here introduced a novel secure two-party protocol for Phase 1.

As shown in Table 6.1, the recent result eliminates the $O(\lg N)$ round complexity using the proposed method that can achieve $O(1)$ round complexity while maintaining the efficiency of communication complexity. Furthermore, the practical performance of the algorithms for real DNA sequences was reported in [44].

Acknowledgements The authors were supported by JST CREST (JPMJCR1402) and JSPS KAKENHI (16K16009, 17H01791, 17H00762, and 18K18111).

References

1. M. Akgün, A.O. Bayrak, B. Ozer, M.S. Sağiroğlu, Privacy preserving processing of genomic data: A survey. Journal of Biomedical Informatics **56**, 103–111 (2015)
2. N. Attrapadung, G. Hanaoka, S. Mitsunari, Y. Sakai, K. Shimizu, T. Teruya, Efficient two-level homomorphic encryption in prime-order bilinear groups and a fast implementation in webassembly. In *ASIACCS* (2018), pp. 685–697
3. H. Bannai, T. Gagie, T. I, Refining the r-index. Theor. Comput. Sci. **812**, 96–108 (2020)
4. M. Blanton, E. Aguiar, Private and oblivious set and multiset operations. In *ASIACCS* (2012), pp. 40–41
5. B.H. Bloom, Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
6. D. Boneh, E.J. Goh, K. Nissim, Evaluating 2-DNF formulas on ciphertexts. In *TCC* (2005), pp. 325–341
7. Z. Brakerski, C. Gentry, V. Vaikuntanathan, (Leveled) Fully homomorphic encryption without bootstrapping. In *ITCS* (2012), pp. 309–325
8. M. Burrows, D.J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm* (Technical report, HP Labs, 1994)
9. D. Catalano, D. Fiore, Using linearly-homomorphic encryption to evaluate degree-2 functions on encrypted data. In *CCS* (2015), pp. 1518–1529
10. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. IEEE Trans. Inform. Theory, 51(7), 2554–2576, 2005
11. F. Claude, G. Navarro, Improved grammar-based compressed indexes. In *SPIRE* (2012), pp. 180–192
12. G. Cormode, S. Muthukrishnan, The string edit distance matching problem with moves. ACM Trans. Algor. **3**(1), 2 (2007)
13. A. Davidson, C. Cid, An efficient toolkit for computing private set operations. In *ACISP* (2017), pp. 261–278
14. P. Ferragina, G. Manzini, Opportunistic data structures with applications. In *FOCS* (2000), pp. 390–398
15. J. Fischer, T. I, D. Köppl, Deterministic sparse suffix sorting on rewritable texts (2015)
16. D.M. Freeman, Converting pairing-based cryptosystems from composite-order groups to prime-order groups. In *EUROCRYPT* (2010), pp. 44–61
17. T. Gagie, G. Navarro, N. Prezza, Fully functional suffix trees and optimal text searching in bwt-runs bounded space. J. ACM **67**(1), 2:1–2:54 (2020)
18. M. Ganczorz, P. Gawrychowski, A. Jez, T. Kociumaka, Edit distance with block operations. In *ESA* (2018), pp. 33:1–33:14
19. C. Gentry, Fully homomorphic encryption using ideal lattices. In *STOC* (2009)
20. O. Goldreich, *Foundations of Cryptography*, vol. Volume (Cambridge University Press, II, 2004)
21. K. Goto, H. Bannai, S. Inenaga, M. Takeda, LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *CPM* (2015), pp. 219–230
22. G. Herold, J. Hesse, D. Hofheinz, C. Ràfols, A. Rupp, Polynomial spaces: a new framework for composite-to-prime-order transformations. In *CRYPTO* (2014), pp. 261–279
23. T. I, Longest common extensions with recompression. In *CPM* 2017, pp. 18:1–18:15
24. T. I, W. Matsubara, K. Shimohira, S. Inenaga, H. Bannai, M. Takeda, K. Narisawa, A. Shinohara, Detecting regularities on grammar-compressed strings. Inf. Comput. **240**, 74–89 (2015)

25. A. Inan, S. Kaya, Y. Saygin, E. Savas, A. Hintoglu, A. Levi, Privacy preserving clustering on horizontally partitioned data. Data and Knowledge Engineering **63**(3), 646–666 (2007)
26. A. Jeż, Compressed membership for NFA (DFA) with compressed labels is in NP (P). In *STACS* (2012), pp. 136–147
27. A. Jeż. Approximation of grammar-based compression via recompression. Theor. Comput. Sci., 592:115–134, 2015
28. A. Jeż, Faster fully compressed pattern matching by recompression. ACM Trans. Algor. **11**(3), 20:1–20:43 (2015)
29. A. Jeż. One-variable word equations in linear time. Algorithmica, 74(1), 1–48, 2016
30. A. Jeż, Recompression: A simple and powerful technique for word equations. J. ACM **63**(1), 4 (2016)
31. A. Jeż, M. Lohrey, Approximation of smallest linear tree grammar. In *STACS* (2014), pp. 445–457
32. D. Kempa, Optimal construction of compressed indexes for highly repetitive texts. In *SODA* (2019), pp. 1344–1357
33. D. Kempa, N. Prezza, At the roots of dictionary compression: string attractors. In *STOC* (2018), pp. 827–840
34. J.C. Kieffer, E.H. Yang, Grammar-based codes: A new class of universal lossless source codes. IEEE Trans. Information Theory **46**(3), 737–754 (2000)
35. L. Kissner, D.X. Song, Privacy-preserving set operations. In *CRYPTO* (2005), pp. 241–257
36. T. Kociumaka, G. Navarro, N. Prezza, Towards a definitive measure of repetitiveness (2019)
37. S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. Theor. Comput. Sci., 483:115–133, 2013
38. N.J. Larsson, A. Moffat, Offline dictionary-based compression. In *DCC* (1999), pp. 296–305
39. E. Lehman, *Approximation Algorithms for Grammar-Based Compression* (MIT, 2002). (PhD thesis)
40. A. Lempel, J. Ziv, On the complexity of finite sequences. IEEE Trans. Information Theory **22**(1), 75–81 (1976)
41. V. Mäkinen, G. Navarro, J. Sirén, N. Välimäki, Storage and retrieval of highly repetitive sequence collections. Journal of Computational Biology **17**(3), 281–308 (2010)
42. S. Maruyama, H. Sakamoto, and M. Takeda. An online algorithm for lightweight grammar-based compression. Algorithms, 5:213–235, 2012
43. T. Masaki, T. Kida, Online grammar transformation based on Re-Pair algorithm. In *DCC* (2016), pp. 349–358
44. S. Nakagawa, T. Sakamoto, Y. Takabatake, T. I, K. Shin, H. Sakamoto, Privacy-preserving string edit distance with moves. In *SISAP* (2018), pp. 226–240
45. G. Navarro, N. Prezza, Universal compressed text indexing. Theor. Comput. Sci. **762**, 41–50 (2019)
46. T. Nishimoto, Y. Tabei, Conversion from RLBWT to LZ77. In *CPM* (2019), pp. 9:1–9:12
47. T. Ohno, K. Sakai, Y. Takabatake, T. I, H. Sakamoto, A faster implementation of online RLBWT and its application to LZ77 parsing. J. Discrete Algorithms **52–53**, 18–28 (2018)
48. P. Paillier, Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT* (1999), pp. 223–238
49. A. Policriti, N. Prezza, Computing LZ77 in run-compressed space. In *DCC* (2016), pp. 23–32
50. S. Rane, W. Sun, Privacy preserving string comparisons based on levenshtein distance. In *WIFS* (2010), pp. 1–6
51. R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM, 21(2), 120–126, 1978
52. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comp. Sci., 302(1–3):211–222, 2003
53. K. Sakai, T. Ohno, K. Goto, Y. Takabatake, T. I, H. Sakamoto, Repair in compressed space and time. In *DCC* (2019), pp. 518–527
54. H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. J. Discrete Algorithms, 3(2–4), 416–430, 2005

55. D. Shapira and J.A. Storer. Edit distance with move operations. J. Discrete Algorithms, 5(2), 380–392, 2007
56. Y. Tabei, H. Saigo, Y. Yamanishi, S.J. Puglisi, Scalable partial least squares regression on grammar-compressed data matrices. In *KDD* (2016), pp. 1875–1884
57. Y. Takabatake, T. I, H. Sakamoto, A space-optimal grammar compression. In *ESA* (2017), pp. 67:1–67:15
58. E.-H. Yang, D.-K. He, Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform - part two: with context models. IEEE Trans. Inform. Theory **49**(11), 2874–2894 (2003)
59. Y. Yoshimoto, M. Kataoka, Y. Takabatake, T. I, K. Shin, H. Sakamoto, Faster privacy-preserving computation of edit distance with moves. In *WALCOM* (2020), pp. 308–320