Article

Enhancing Data Compression: Recent Innovations in LZ77 Algorithms

Aaron Hong and Christina Boucher *

Posted Date: 25 September 2024

doi: 10.20944/preprints202409.1877.v1

Keywords: LZ77; Data compression; Data structures; suffix Array; Burrows Wheeler Transform

Article

Enhancing Data Compression: Recent Innovations in LZ77 Algorithms

Aaron Hong [1], **Christina Boucher** [2,*]

[1] Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 32607 United States

[2] Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 32607 United States

* Correspondence: christinaboucher@ufl.edu

Abstract: The burgeoning volume of genomic data, fueled by advances in sequencing technologies, demands efficient data compression solutions. Traditional algorithms like Lempel-Ziv77 (LZ77) have been foundational in offering lossless compression, yet they often fall short when applied to the highly repetitive structures typical of genomic sequences. This review delves into the evolution of LZ77 and its derivatives, exploring specialized algorithms such as prefix-free parsing, AVL grammars, and LZ-based methods tailored for genomic data. Innovations in this field have led to enhanced compression ratios and processing efficiencies by leveraging the intrinsic redundancy within genomic datasets. We critically examine a spectrum of LZ77-based algorithms, including newer adaptations for external and semi-external memory settings, and contrast their efficacy in managing large-scale genomic data. Additionally, we discuss the potential of these algorithms to facilitate the construction of data structures such as compressed suffix trees, crucial for genomic analyses. This paper aims to provide a comprehensive guide on the current landscape and future directions of data compression technologies, equipping researchers and practitioners with insights to tackle the escalating data challenges in genomics and beyond.

Keywords: LZ77; data compression; data structures; suffix array; burrows wheeler transform

1. Introduction

In the era of increasingly larger datasets, the exponential growth of genomic data generated by advanced sequencing technologies poses significant challenges for data storage and management. The vast amounts of data being produced are not only massive in scale but also exhibit a high degree of redundancy, particularly within and across genomes of the same species. This redundancy, while beneficial for data compression, demands sophisticated algorithms to efficiently leverage these repetitive structures for enhanced storage and retrieval processes.

LZ77 and LZ78 are seminal lossless data compression algorithms developed by Abraham Lempel and Jacob Ziv, which laid the groundwork for numerous modern compression techniques. Published in 1977, LZ employs a sliding window mechanism to find repeated sequences within a fixed-size window, encoding data as a triple (distance, length, next symbol). The LZ algorithm is efficient in hardware implementations due to its use of a sliding window buffer, which stores recent sequences as references. Later, in 1978, Lempel and Ziv introduced LZ78, which builds a dynamically expanding dictionary of previously seen sequences, encoding data with a pair (index, next symbol). The LZ78 [1] diverges from LZ77 by implementing a dynamic growing dictionary, which avoids the limitations of a fixed-length window. Unlike LZ77, LZ78 maintains an explicit dictionary, often implemented as a trie. The memory usage in LZ78 depends on the size of the dictionary, while LZ77's memory usage is tied to the sliding window size. LZ78 can achieve better compression ratios for certain data types due to its adaptive dictionary approach. Both algorithms significantly influenced the field of data compression. LZW (Lempel-Ziv-Welch), an enhancement of LZ78 by Welch in 1984, became notable for its use in the GIF image format and Unix compress program. LZW eliminates the explicit output of the next symbol, thereby reducing redundancy and improving compression efficiency. Additionally, the Deflate algorithm, which combines LZ77 with Huffman coding, is used in formats like ZIP and gzip, offering a balance between compression ratio and speed. LZ77 and LZ78 have led to numerous derivative algorithms, such as LZSS, LZMA, and LZ4, each enhancing specific aspects of the original methods. These algorithms are foundational in various applications, from file compression to network data transfer, demonstrating their enduring impact on data compression technology.

This paper delves into the evolution of LZ77 and its various derivatives, focusing on innovations that tailor these algorithms for genomic data compression. We explore a range of LZ77-based algorithms developed to address the specific challenges posed by genomic datasets. These include adaptations that optimize the algorithm for external and semi-external memory settings, enhancing their applicability in handling large-scale data without compromising processing speed or efficiency. Additionally, we examine the role of these algorithms in facilitating the construction of advanced data structures, such as compressed suffix trees, crucial for genomic analyses. By improving the compression and decompression processes, these adapted algorithms help manage the storage demands of large genomic datasets, thereby supporting faster and more cost-effective genomic research.

This review also highlights recent advancements in hardware implementations of LZ77-based algorithms. These hardware solutions are designed to accelerate the compression and decompression processes, addressing the throughput limitations often encountered in software implementations. By integrating these algorithms into hardware, it is possible to achieve greater data processing speeds, making real-time genomic data analysis more feasible. Through a comprehensive examination of both software and hardware advancements in LZ77-based data compression, this paper aims to provide a detailed guide on the current state of the art and future directions in this field. Our objective is to equip researchers and practitioners with the knowledge and tools necessary to tackle the challenges of genomic data management, paving the way for more efficient and effective use of genomic information in various biological and medical applications. The time and space complexity for in memory LZ or LZ like construction algorithm is showed in Table 1. Lastly, this paper also includes pseudocode in Python for the algorithms discussed, which can aid in teaching the LZ77 algorithm. By providing clear, step-by-step code examples, the pseudocode offers an accessible resource for those looking to understand or implement these algorithms, making it especially useful in an educational context.

Table 1. Illustration the time and space complexity for in memory construction algorithm. The symbol Σ represents the alphabet size and the r is closely related to the number of repetitions in the text, which can be exponentially smaller than n.

LZ Factorization Algorithm	Year	Time Complexity	Space Complexity
Ultra-fast LZ factorization [2]	2010	$O(2n)$	$O(n)$
LZ factorization backward search [2]	2010	$O(n \log \|\Sigma\|)$	$O(n)$
KKP1 [3]	2012	$O(n)$	$O(n \log n)$
KKP2 [3]	2012	$O(n)$	$O(2n \log n)$
KKP3 [3]	2012	$O(n)$	$O(3n \log n)$
BGone [4]	2014	$O(n)$	$O(n \log n) + O(\|\Sigma\| \log n)$
LZoneT [5]	2016	$O(n)$	$O(n \log n)$
LZoneSA [5]	2016	$O(n)$	$O(n \log n) + O(\|\Sigma\|)$
RLE_LZ [6]	2017	$O(n \log r)$	$O(r \log n)$
PFP-LZ77 [7]	2023	$O(n)$	$O(6n \log n)$
Sublinear LZ [8]	2023	$O(n)$	$O(n \log n)$
LZ-End [2]	2020	$O(n \log n)$	$O(2n + n \log n)$
LZD [9]	2024	$O(n)$	$O(3n \log n)$
LZMW [9]	2024	$O(n)$	$O(3n \log n)$

2. Notation and Preliminaries

Basic Definitions

We denote a string S as a finite sequence of symbols $S = S[1..n] = S[1] \cdots S[n]$ over an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$ whose symbols can be unambiguously ordered. We denote by ε the empty string, and the length of S as $|S|$. Given a string S, we denote the reverse of S as $\mathsf{rev}(S)$, i.e., $\mathsf{rev}(S) = S[n] \cdots S[1]$.

We denote by $S[i..j]$ the substring $S[i] \cdots S[j]$ of S starting in position i and ending in position j, with $S[i..j] = \varepsilon$ if $i > j$. For a string S and $1 \le i \le n$, $S[1..i]$ is called the i-th prefix of S, and $S[i..n]$ is called the i-th suffix of S. We call a prefix $S[1..i]$ of S a *proper prefix* if $1 \le i < n$. Similarly, we call a suffix $S[i..n]$ of S a *proper suffix* if $1 < i \le n$.

We denote by $<_{\mathsf{lex}}$ the lexicographic order: for two strings $S[1..m]$ and $T[1..n]$, $S <_{\mathsf{lex}} T$ if S is a proper prefix of T, or there exists an index $1 \le i \le \min(n, m)$ such that $S[1..i-1] = T[1..i-1]$ and $S[i] <_{\mathsf{lex}} T[i]$.

Given a string S, a symbol $c \in \Sigma$, and an integer i, we define $S.\mathsf{rank}_c(i)$ as the *rank* as the number of occurrences of c in $S[1..i-1]$. We also define $S.\mathsf{select}_c(i)$ as the position in S of the i-th occurrence of c in S if it exists, and $n+1$ otherwise. For a bitvector $B[1..n]$, that is a string over $\Sigma = \{0, 1\}$, to ease the notation we will refer to $B.\mathsf{rank}_1(i)$ and $B.\mathsf{select}_1(i)$ respectively as $B.\mathsf{rank}(i)$ and $B.\mathsf{select}(i)$.

Suffix array and Inverse Suffix Array

The *suffix array* [10] is a compact and efficient data structures storing all the sorted suffixes of a string S of length n. More specifically, we note that the suffix array SA for the string S is a permutation of $1, .., n$ such that $S[\mathsf{SA}_S[i]..n]$ represents the i-th lexicographically smallest suffix of S. Complementing the suffix array is the *inverse suffix array*, we denoted it as ISA_S. Which serves as mapping back to the original indices of the suffixes, enabling one to determine the rank of each suffix in the lexicograpghical order efficiently. The ISA is defined such that $\mathsf{ISA}_S[\mathsf{SA}_S[i]] = i$ for all $i = [1, ..., n]$, thereby offering a direct correspondence between the suffix array and the original string.

Burrows-Wheeler Transform

The Burrows-Wheeler transform (BWT) [11] of a string S is a reversible permutation of the character of the string S. This transform is obtained sorting all rotations of a string S in lexicographic order and retaining the last character of each rotation. Let M be the matrix containing all sorted rotations of S, let F and L be the first and the last column of the matrix, then $\mathsf{BWT}_S = \mathsf{L}$. Moreover, let $\mathsf{C}[c]$ bet the number of suffixes starting with a character smaller than c. We define LF-*mapping* as $\mathsf{LF}(i, c) = \mathsf{C}[c] + \mathsf{rank}_c(\mathsf{BWT}, i)$ and $\mathsf{LF}(i) = \mathsf{LF}(i, \mathsf{BWT}[i])$. With the LF-mapping is possible to reconstruct S. It is in fact sufficient to set an iterator $s = 1$ and $S[n] = \mathsf{BWT}[1]$ and for each $i = n, n-1, \ldots, 1$ do $s = LF(s)$ and $S[i] = \mathsf{BWT}[s]$. The same function can also be used to count the occurrences of a pattern P in S. We refer to this technique as *backward search*. Given a pattern $P = [1..p]$, the *backward search* consist of p steps that preserve the following invariant: at the i-th step, sp stores the position of the first row of M prefixed by $P[i, p]$ while ep stores the position of the last row of M prefixed by $P[i, p]$. To advance from i to $i-1$, hence the name *backward search*, we use the LF-mapping on sp and ep. In practice no building algorithm computes the M, in fact the BWT of S is usually built from the suffix array of S, reaching linear time complexity as $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$, with $\mathsf{BWT}[i] = T[n]$, if $\mathsf{SA}[i] = 1$.

Given a string S and its BWT, a *run* in BWT is a maximal substring of equal characters in BWT. This leads to the definition of the *run-length encoded Burrows Wheeler Transform* (RLBWT) [12], which offers a more space-efficient variation by comprising sequences of repeated characters in the BWT sequence into a single run. Hence, RLBWT is an equivalent representation of the BWT, where each run is represented as the character of the run and its length. For example, given the BWT of a string S as *bc#bbbbccccbaaaaaaaaaaa*, the sequence of characters beginning each run is described as

$H = bc\#bcba$. Each character c in S has its run lengths encoded in a corresponding bit-vector B_c, with a 1 marking the start of each run. In this instance, the bit-vector are $B_{all} = 11110001000110000000000$, $B_a = 10000000000$, $B_b = 110001$, and $B_c = 11000$. The rank and access operations are efficiently translated into rank, select, and access operations on H, B_{all}, and B_c. This encoding strategy reduced the storage of BWT, reducing the space complexity to $\mathsf{O}(r)$, where r denotes the number of runs in the BWT.

Longest Common Extension

The Longest Common Extension array (LCE) measures the length of the longest common suffix between two suffixes of a string. If we are given a string $S[1..n]$ and two indices $1 \leq i \leq j \leq n$, then $\mathsf{LCE}(i, j)$ is defined as the length of the longest common suffix between the suffixes starting at the positions i and j in S. Additionally, $\mathrm{lce}(i, j)$ refers to the specific value obtained from an $\mathsf{LCE}(i, j)$ query, representing the length of the longest common prefix between the suffixes starting at i and j. This value can be stored in a precomputed array.

Longest Common Prefix Array

The longest Common Prefix array (LCP) is a data structure that captures the lengths of the longest common prefixes between consecutive suffixes in the SA of a string S. More specifically, $\mathsf{LCP}[i]$ represents the length of the longest common prefix between the suffixes represented by $\mathsf{SA}[i]$ and $\mathsf{SA}[i-1]$ in S. Given two strings $S[1..n]$ and $T[1..m]$ we refer to the *longest common prefix* as the substring $S[1..\ell] = T[1..\ell]$ such that $\ell = \max\{i \mid S[1..i] = T[1..i]\}$. We denote the length ℓ of the longest common prefix of S and T as $\mathtt{lcp}(S, T)$.

Range Minimum Query

Given an array $\mathsf{A}[1..n]$ of integers, and an interval $1 \leq i \leq j \leq n$, a *range minimum query* (RMQ) of the interval $[i, j]$ over the array A, denoted by $\mathsf{RMQ}_{\mathsf{A}}[i, j]$, returns the position k of the minimum value in $\mathsf{A}[i..j]$. Formally, $\mathsf{RMQ}_{\mathsf{A}}[i, j] = \mathrm{argmin}(\mathsf{A}[k] i \mid \leq k \leq j)$. RMQ is frequently paired with the LCP to answer queries of the sort, given two suffixes $S[\mathsf{SA}[i]..n]$ and $S[\mathsf{SA}[j]..n]$ of the string $S[1..n]$, find the value of $\mathtt{lcp}(S[\mathsf{SA}[i]..n], S[\mathsf{SA}[j]..n])$. This can be done using the RMQ over the LCP. Hence, given two integers $1 \leq i \leq j \leq n$ we define $\mathtt{lcp}_{\mathsf{SA}}(i, j)$ as the longest common prefix between the i-th and j-th suffix in SA order, which is equivalent to computing the lceof $\mathsf{SA}[i]$ and $\mathsf{SA}[j]$. Formally, $\mathtt{lcp}_{\mathsf{SA}}(i, j) = \mathtt{lcp}(S[\mathsf{SA}[i]..n], S[\mathsf{SA}[j]..n]) = \mathrm{lce}(\mathsf{SA}[i], \mathsf{SA}[j]) = \mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}[i+1, j]]$.

Φ and Ψ

Given a string $S[1, \ldots, n]$ and its suffix array (SA) and inverse suffix array (ISA), $\Phi[i]$ is equal to $\mathsf{SA}[\mathsf{ISA}[i] - 1]$ for $\mathsf{ISA}[i] > 1$, and $\Phi[i]$ is equal to $\mathsf{SA}[i]$ otherwise [13]. Additionally, the Ψ permutation is defined as $\Psi[i] = \mathsf{SA}[\mathsf{ISA}[i] + 1]$, with the conditions $\Psi[\mathsf{SA}[n]] = 0$ and $\Psi[0] = \mathsf{SA}[1]$.

Previous Smaller Value and Next Smaller Value queries

Previous Smaller Value (PSV) and Next Smaller Value (NSV) queries are used frequently for computing LZ, as we will see within this survey. These queries concern themselves with an integer array $\mathsf{A}[1..n]$ and a specific index $1 \leq i \leq n$. The PSV of A at index i is defined as the maximum position preceding i within the array where a value smaller than $\mathsf{A}[i]$ exists. Formally

$$\mathsf{PSV}_{\mathsf{A}}(i) = \begin{cases} \max\{1 \leq j < i \mid \mathsf{A}[j] < \mathsf{A}[i]\} & \text{if } j \text{ exists,} \\ 0 & \text{otherwise} \end{cases}$$

.

Similarly, the NSV of A at the index i is defined as the minimum position succeeding i within the array where a value smaller than $\mathsf{A}[i]$ exists. Formally,

$$\mathsf{NSV}_{\mathsf{A}}(i) = \begin{cases} \min\{i < j \leq n \,|\, \mathsf{A}[j] < \mathsf{A}[i] & \text{if } j \text{ exists,} \\ 0 & \text{otherwise} \end{cases}$$

These queries find application in various algorithmic problems, particularly in tasks involving array traversal, range queries, and optimization problems. They are often employed in algorithms for finding the nearest smaller element to the left or right for each element in an array.

Next, we define $\mathsf{NSV}_{\mathsf{lex}}[i]$ and $\mathsf{PSV}_{\mathsf{lex}}[i]$ as follows:

$$\mathsf{NSV}_{\mathsf{lex}}[i] = \min\{j \in [i+1..n] | \mathsf{SA}[j] < \mathsf{SA}[i]$$

and

$$\mathsf{PSV}_{\mathsf{lex}}[i] = \max\{j \in [1..i-1] | \mathsf{SA}[j] < \mathsf{SA}[i]$$

Lastly, we define $\mathsf{NSV}_{text}[i]$ and $\mathsf{PSV}_{text}[i]$ as follows:

$$\mathsf{NSV}_{text}[i] = \mathsf{SA}[\mathsf{NSV}_{\mathsf{lex}}(\mathsf{ISA}[i])],$$

and

$$\mathsf{PSV}_{text}[i] = \mathsf{SA}[\mathsf{PSV}_{\mathsf{lex}}(\mathsf{ISA}[i])],$$

assuming $\mathsf{SA}[0] = 0$ for all $1 \leq i \leq n$. These last equations are used frequently for computing the LZ factorization.

The FM-index

The FM-index [14] is an index based on the BWT and the SA. As we have seen the BWT by itself it is capable of counting the number of occurrences of a pattern. In order to locate all such occurrences the SA is required. We note that, in order to achieve compression, the FM-index does not store all the SA values, instead it samples some of the values while still being able to compute the values that are not sampled trough the LF-mapping. As a results, the FM-index of a string S is capable of computing the number of occurrences of a pattern P in $\mathsf{O}(|P|)$ time and can compute the positions of each one of such occurrences in $\mathsf{O}(\log^{\epsilon}|S|)$ by storing the BWT S plus $\mathsf{o}(s)$ bits of auxiliary information. That is, in order to find a pattern P in the FMI we can use the LF-mapping to find the range $[sp, ep]$ of the BWT that corresponds to all the suffixes of S that are prefixed by P. Once we obtained the range on the BWT we either land, on the last LF step, on one of the sampled positions of the SA, or we have to retrieve it. To do so we keep performing steps of the *backward search* until we find a value of the SA that is sampled. From that sample we can compute the value of the SA that we were looking for The FM-index is a compressed full-text substring index that leverages the BWT to efficiently search an input string S. The index uses the BWT to rearrange the original string, grouping similar contexts together for enhanced compressibility. To navigate and reconstruct the string, the FM-index employs rank and select structures, which efficiently count character occurrences up to any position in the BWT. Instead of storing the entire suffix array, the FM-index keeps sampled positions, aiding in quick location jumps within the string. One of its fundamental features, the backward search, allows the index to efficiently find substrings by analyzing from the end of the pattern to its beginning. This ensemble of features renders the FM-index a powerful tool for applications requiring swift and efficient string processing while minimizing memory usage.

3. Computing the LZ77 Compression

In-memory construction of the LZ algorithm involves managing a sliding window of data that efficiently encodes repeated sequences by referencing previous occurrences directly within memory.

Here, we provide a concise overview of these in-memory construction algorithms, presented in chronological order.

3.1. Early Methods of Computing LZ77

The algorithm proposed by Kreft and Navarro [2] addresses the limitation of traditional LZ77 compression regarding efficient random access. Their approach involves segmenting the input data into fixed-size segments, each compressed separately using the LZ77 algorithm, and creating an index for each segment to store pointers to the start of each phrase within the segment. This segmentation ensures phrases do not cross boundaries, allowing independent decompression of segments. Additional data structures, such as a phrase directory and offset tables, are maintained to facilitate quick location and decompression of specific phrases within a segment. Although these structures introduce some overhead, the algorithm balances compression efficiency with access speed, ensuring minimal impact on the compression ratio. Experimental results demonstrate that the method significantly improves random access times compared to traditional LZ77, making it suitable for applications requiring both efficient compression and fast data retrieval, such as large-scale data storage systems and real-time data processing systems.

Ohlebusch and Gog [15] described two linear-time LZ factorization algorithms. In their first algorithm, they optimized the precomputation of the LCP array by computing it simultaneously with the arrays for the longest previous substring (LPS) and the previous occurrence position of the first time a character appeared (PrevOcc). In their algorithm, the main procedure computes the LCP values as in the ϕ algorithm of Kärkkäinen et al. [13], which determines the length l of the LCP of a suffix S_i of the input string S and the suffix S_j that precedes S_i in the suffix array. Another key aspect of their approach is the graph representation of SA and LCP introduced by Crochemore and Ilie [16]. The graph has $|S| + 2$ nodes, each labeled with $(i, \mathsf{SA}[i])$, and each node is connected by edges labeled with the value of $\mathsf{LCP}[i+1]$. In this algorithm, they didn't store the graph itself but stored the edges in the LPS and PrevOcc arrays. Below is the Python code corresponding to this algorithm:

```
def sop(i, l, j):
    if LPS[i] == PERP:
        LPS[i] = l
        PrevOcc[i] = j
    else:
        if LPS[i] < l:
            if PrevOcc[i] > j:
                sop(PrevOcc[i], LPS[i], j)
            else:
                sop(j, LPS[i], PrevOcc[i])
            LPS[i] = j
            PrevOcc[i] = j
        else:
            if PrevOcc[i] > j:
                sop(PrevOcc[i], l, j)
            else:
                sop(j, l, PrevOcc[i])
```

The second algorithm proposed by Ohlebusch and Gog further improved the space usage for the LZ factorization computation. This algorithm is similar to the CPS2 algorithm [17] but is more space efficient because it applies the backward search on the string $T = S^{rev}\$$, where the S^{rev} is the reverse of S. Based on the S^{rev}, they replace the PSV and NSV with previous greater value (PGV) and next greater value (NGV), where the

$$\mathsf{PGV} = \max\{j : 0 \leq j < i \text{ and } \mathsf{SA}[j] > \mathsf{SA}[i]\},$$

and

$$\mathsf{NGV} = \min\{j : i < j \leq n+1 \text{ and } \mathsf{SA}[j] > \mathsf{SA}[i]\}.$$

The corresponding code for this second algorithm is provided as follows:

```
def LZ_back(T):
    i = len(T)
    j = 1
    while i > 1:
        i' = i
        [sp..ep] = backwardSearch(T[i], [1..n+1])
        while NGV[LF(j)] <= ep or PGV[LF(j)] >= sp:
            j = LF(j)
            [lb..rb] = [sp..ep]
            i = i - 1
            [sp..ep] = backwardSearch(T[i], [sp..ep])
        LPS = i' - i
        if LPS = 0:
            PrevOcc = T[i]
            i = i - 1
        elif NGV[j] <= rb:
            PrevOcc = n + 1 - SA[NGV[j]] - (LPS - 1)
        else
            PrevOcc = n + 1 - SA[PGV[j]] - (LPS - 1)
        output(PrevOcc, LPS)
```

3.2. Lazy LZ Construction

Kempa and Puglisi [18] and Goto and Bannai [19] independently developed algorithms for computing the LZ77 that are based on the intuition that given a position i in SA, then the longest previous factor of $S[\mathsf{SA}[i]..n]$ is either the suffix in position $\mathsf{SA}[\mathsf{PSV}_{\mathsf{SA}}(i)]$ or the suffix in position $\mathsf{SA}[\mathsf{NSV}_{\mathsf{SA}}(i)]$ since those are the suffixes sharing the longest prefix with the suffix in position $\mathsf{SA}[i]$ that occur before in the string. In particular, both algorithms compute the longest previous factor of a given position i in S as $\mathrm{lce}(\mathsf{PSV}_{text}[i], \mathsf{NSV}_{text}[i])$ which is equal to the $\min\{\mathrm{lce}(i, \mathsf{NSV}_{text}[i]), \mathrm{lce}(i, \mathsf{PSV}_{text}[i])\}$. Both algorithms use the naive computation of the lce, i.e., character-by-character string comparison. The methods presented by Kempa and Puglisi, and Goto and Bannai are often referred to as "lazy evaluation of LCP values" since $\mathtt{lcp}(i, \mathsf{NSV}_{text}[i])$ and $\mathtt{lcp}(i, \mathsf{PSV}_{text}[i])$ are computed only when i is a starting position of a phrase. However, the methods differ on how PSV_{text} and NSV_{text} are computed. Below is the python code corresponding to this computation.

```
def LZ_Factor(i, psv, nsv, X):
    if lcp(i, psv) > lcp(i, nsv):
        p, l = psv, lcp(i, psv)
    else:
        p, l = nsv, lcp(i, nsv)

    if l == 0:
        p = X[i]
        print(f"output factor ({p}, {l})")

    return i + max(l, 1)
```

Goto and Bannai define three algorithms, each of which requires linear time but differ based on the amount of memory they use. The differences reflect how they compute PSV_{text} and NSV_{text} The first algorithm they define computes and stores the entire PSV_{text} array and NSV_{text} arrays for the input string S using the Φ array, where $\Phi[1..n]$ is defined as $\Phi[i] = \mathsf{SA}[\mathsf{ISA}[i] - 1]$ for all $1 \leq i \leq n$. This computation of PSV and NSV was previously defined by Ohlebusch and Gog [15]. They refer to this as the BGT algorithm. BGT requires $3n \log n$ bits of working space. Goto and Bannai then define two other algorithms, which they refer to as BGS and BGL, that simulate PSV_{text} and NSV_{text} using SA and ISA as we recall from the preliminaries that $\mathsf{NSV}_{text}[i] = \mathsf{SA}[\mathsf{NSV}_{\mathsf{SA}}(\mathsf{ISA}[i])]$, and $\mathsf{PSV}_{text}[i] = \mathsf{SA}[\mathsf{PSV}_{\mathsf{SA}}(\mathsf{ISA}[i])]$. BGS is implements the computation of $\mathsf{PSV}_{\mathsf{lex}}[i]$ and $\mathsf{NSV}_{\mathsf{lex}}[i]$ as a simple scan of the text using a stack. BGL computes in lexicographic order and thus, does not require a stack. This computation of PSV and NSV was previously defined by Crochemore and Ilie [16]. BGL requires $4n \log n$ bits of working space, and BGS requires $(4n + s) \log n$ bits of working space, where s is defined as the size of the stack used by BGS.

Kempa and Puglisi also apply lazy evaluation techniques to PSV and NSV values. They use ISA and a compact auxiliary data structure that allows for arbitrary $\mathsf{NSV}_{\mathsf{lex}}$ and $\mathsf{PSV}_{\mathsf{lex}}$ queries. They refer to their algorithm as ISA9 The computation of $\mathsf{NSV}_{text}[i]$ and $\mathsf{PSV}_{text}[i]$ occurs only when i marks the start of a phrase. This method uses $(2 + 1/b)n \log n$ bits of working space and takes $\mathsf{O}(n + zb + z \log(n/b))$-time, where b represents a parameter that balances space and time within the NSV/PSV data structure. Kempa and Puglisi further demonstrate how to reduce the space needed to $(1 + \epsilon)n \log n + n + \mathsf{O}(\sigma \log n)$ bits by employing a succinct ISA representation. The lazy evaluation makes these algorithms particularly effective when the resultant LZ factorization is small. Although, the running time of the method of Kempa and Puglisi is slightly larger than linear, however, in practice it works efficiently [3].

3.3. The KKP Algorithms

"Kärkkäinen, Kempa, and Puglisi [3] introduced a suite of algorithms collectively referred to as the KKP algorithms, which are named after the authors. These algorithms encompass a variety of techniques, each designed to address specific aspects of computational efficiency and data processing. The KKP algorithms can also be seen as using a lazy evaluation of the LCP values, i.e., $\texttt{lcp}(i, \mathsf{PSV}_{text}[i])$ and $\texttt{lcp}(i, \mathsf{NSV}_{text}[i])$ is a starting position of phrase.

The first algorithms, referred to as KKP3, requires $\mathsf{O}(3n \log n)$ bits of working space. KKP3 shares close ties with the algorithms developed by Goto and Bannai [19] in that it first computes the NSV_{text} and PSV_{text} arrays in the same manner as that the approach used by the BGT algorithm, i.e., by scanning the SA. Subsequently, they use these arrays for a lazy LZ factorization process that is similar to BGT. Thus, KPP3 can be seen as a merging of the BGT and BGS algorithms of Goto and Bannai. Here, we note that KPP3 diverges from the algorithms developed by Goto and Bannai in a key aspect: it does not compute the Φ array. As mentioned by the authors, the KKP3 algorithm interleaves the storage of the NSV_{text} and PSV_{text} arrays, i.e., the position $\mathsf{NSV}_{text}[i]$ and $\mathsf{PSV}_{text}[i]$ are adjacent. This can potentially improve the locality of the access to these arrays, and thus, mitigate the occurrence of cache misses. The potential of this to enhance the overall performance is mentioned by both sets of authors. The python code for KKP3 is shown in below.

```
def KKP3(SA, n):
    PSV_text = [0] * (n + 2)
    NSV_text = [0] * (n + 2)

    SA[0] = 0  # Initializing first and last elements of SA
    SA[n + 1] = 0
    top = 0

    # Processing to fill PSV and NSV
    for i in range(1, n + 2):
        while top > 0 and SA[top] > SA[i]:
            NSV_text[SA[top]] = SA[i]
            PSV_text[SA[top]] = SA[top - 1]
            top -= 1
        top += 1
        if top < len(SA):
            SA[top] = SA[i]

    # Final process utilizing LZ-Factor
    i = 1
    while i <= n:
        i = LZ_Factor(i, PSV_text[i], NSV_text[i])

    return PSV_text, NSV_text
```

KKP2 builds upon the foundation laid by KKP3, the KKP2 algorithm introduces optimizations in memory consumption for LZ factorization. Instead of computing both the NSV$_{text}$ and PSV$_{text}$ arrays prior to computing the LZ factorization, KKP2 only precomputes the NSV$_{text}$ array. Then, after computing the NSV$_{text}$, they show how to sequentially scan and rewrite the NSV$_{text}$ array, to obtain the Φ array in place. The artifact of this computation ls that the values of PSV$_{text}$ (as well as NSV$_{text}$) for each position can be obtained sequentially. This in-place computation of the Φ array that leads to the PSV$_{text}$ values is a novel aspect of their method, which reduces the number of auxiliary arrays of size n by one. The algorithm as python code shown below. The authors also describe how to eliminate the stack that is used in in both KKP3 and KKP2. This leads to KKP2n, which uses a non-local stack simulation approach, which, while having a slight impact on performance, reduced the memory overhead.

```
def KKP2(SA, n):
    Phi = [0] * (n + 2)
    SA[0] = 0
    SA[n + 1] = 0
    top = 0

    for i in range(1, n + 2):
        while top > 0 and SA[top] > SA[i]:
            Phi[SA[top]] = SA[i]
            top -= 1
        top += 1
        if top < len(SA):
            SA[top] = SA[i]

    Phi[0] = 0
    next_index = 1

    for t in range(1, n + 1):
        nsv = Phi[t]
        psv = Phi[nsv]
        if t == next_index:
            next_index = LZ_Factor(t, psv, nsv)
        Phi[t] = psv
        Phi[nsv] = t

    return Phi
```

KKP1 will be discussed in a later section where external memory methods are discussed. The name of the methods, KKP3, KKP2 and KKP1 originate from the number of arrays (of length n) that are stored in memory. KKP3 stores SA, PSV, and NSV. KKP2 stores SA and NSV. KKP1 assumes the SA is stored in external memory and just stores NSV in main memory. In their experimental results, the authors demonstrate that their algorithms surpass those the methods of Kempa and Puglisi [18], and Goto and Bannai [19]. .

3.4. The BGone Algorithm

Motivated by the fact that the KKP algorithms store two integer arrays (KKP1 can be perceived this way event though the SA is in external memory), Goto and Bannai develop algorithms for computing LZ factorization using a single array of size $\mathsf{O}(n)$ [4]. The name of the algorithm refers to the authors names and the number auxiliary arrays used. They refer to their prior work that we described in Subsection 3.2 as BGtwo. The BGone algorithm can be seen as using the following steps: (1) compute the Φ array; then (2) compute the NSV_{text} array from the Φ array in-place in linear-time; and (3) compute the LZ-factorization in linear time by rewriting Φ array to NSV_{text} array in-place, and rewriting NSV_{text} array to Φ array in-place (and sequentially obtain NSV_{text} and PSV_{text} values). The third step is exactly the algorithms implemented by KKP2. Since SA is a permutation of integers from 1 to n, Φ can be treated as an array-based implementation of a singly linked list linking the elements of SA from right to left. The algorithm for computing NSV_{text} values from SA can be simulated using the Φ array. The key difference is that while elements of SA are in lexicographic order, elements of Φ are in text order, which matches NSV_{text}. As access on SA is sequential, once a value $\Phi[i]$ is processed, it's not needed anymore, so it can be rewritten to $\mathsf{NSV}_{text}[i]$. Hence, the computation of the second step is reasonably straightforward. The last step is how to compute Φ in-place in linear-time. In order to accomplish this Goto and Bannai the SA construction algorithm of Nong [20], which runs in linear-time on constant size integer alphabets, can be extended in order to compute Φ. The construction algorithm uses *induced sorting* sorts a certain subset of suffixes, and then induces the lexicographic order of the remaining suffixes by using the lexicographic order of the subset. The construction of Nong, uses what is referred to as the *Left-Most-S-suffixes* to break up the computation. Putting these (in-place) algorithms together, Goto and Bannai give the first LZ factorization that using a single auxiliary array of length n. BGone runs in linear time and $n \log n + \mathsf{O}(|\Sigma| \log n)$ bits space, where $|\Sigma|$ is the alphabet size. Lastly, we note that BGone has two versions, which are referred to as BGoneT and BGoneSA. BGoneT computes the LZ factorization from the input string that is denoted as T. BGoneSA computes that LZ factorization assuming that the SA exists. BGoneT uses SACA-K [20] to compute the SA. The experimental results demonstrate that BGoneT algorithm is about twice as slow as existing algorithms but only half the total space. This feature renders it a feasible alternative, particularly in scenarios where total space, including disk storage, poses limitations due to the immense scale of the data.

3.5. The LZoneT and LZoneSA Algorithms

In light of the development of BGone, Liu et al. [5] introduced the LZone algorithm that has the same theoretical running time and memory usage as BGone but is more time-efficient in practice. LZone uses a single auxiliary integer array of size n but employs a more careful implementation of the Φ array. To understand this computation more precisely, we require some additional definitions. Given an input string S, we define suffix $S[i..n]$ to be either an S-type or an L-type based on its lexicographical properties. A suffix $S[i..n]$ is considered S-type if $S[i] < S[i+1]$, or if $S[i] = S[i+1]$ and $S[i+1..n]$ is also S-type. Conversely, $S[i..n]$ is L-type if $S[i] > S[i+1]$, or if $S[i] = S[i+1]$ and $S[i+1..n]$ is L-type. A suffix is defined as Left-Most S-type (LMS) if it is S-type and the preceding suffix is L-type. Similarly, a suffix is Left-Most L-type (LML) if it is L-type and the preceding suffix is S-type. We note that the last suffix (which is just the last character $S[n]$) is always LMS, and the first suffix (which is the entire string $S[1..n]$) is neither LMS-type nor LML-type. The definition of the SA can be extended based on these definitions. For example, SA_ℓ, SA_s, SA_{lms}, and SA_{lml} are the arrays storing all the sorted L-type, S-type, LMS-type, and LML-type suffixes, respectively.

Next, we provide a high-level overview of LZoneT before delving into the specifics. First, the number of L-type suffixes and S-type suffixes is computed. If the number of S-type suffixes is larger than the number of L-type suffixes, then LZoneT computes the SA_ℓ from the input string S. Next, Φ_ℓ is computed from SA_ℓ, Φ is computed from Φ_ℓ, NSV is computed from Φ, and finally, the LZ77 is computed from NSV. Otherwise, SA_s, Ψ_s, Ψ, PSV, and the LZ77 are computed. SA_ℓ is computed using induced sorting, which sorts the LMS-type suffixes of S, stores them in array $A[1..k]$, and then applies SACA-K [20] to compute SA_ℓ. Similarly, SA_s is constructed using the same algorithm but sorts the LML-type suffixes rather than the LMS-type suffixes.

Next, Φ_ℓ (or Ψ_s) is computed from SA_ℓ (or SA_s). If the number of L-type suffixes is larger than the number of S-type suffixes, then Ψ_s is computed. LZoneT transforms SA_ℓ into an array-based linked list representation, linking all suffixes in SA_s from the lexicographically smallest to the largest.

LZoneT then simulates the process in SACA-K for induced sorting of the L-type suffixes by setting $A[\mathsf{SA}_s[i]] = \mathsf{SA}_s[i+1]$, $A[0] = A[\mathsf{SA}_s[1]]$, and $A[\mathsf{SA}_s[k]] = 0$. Ψ_s is obtained in $A[0]$, which is the lexicographically smallest suffix in SA_s. If the number of L-suffixes is smaller than or equal to the number of S-type suffixes, they compute Φ_ℓ using the same algorithm as described previously, but with all suffixes in SA_ℓ linked from the lexicographically largest to the smallest for induced sorting of the S-type suffixes.

Then, Ψ (or Φ) can be computed from Ψ_s (or Φ_ℓ). As previously described, if the number of L-type suffixes is larger than the number of S-type suffixes, then Ψ is computed from Ψ_s. This process is illustrated in the following Python code.

```
def compute_Psi_from_Psi_s(T, Psi_s):
    Lbkts = [None] * len(T)  # Lexicographically smallest suffixes
    Lbkte = [None] * len(T)  # Lexicographically largest suffixes
    cur = 0
    prev = 0
    next_ = 0
    pree = 0

    for i in range(len(T)):
        for j in [0, 1]:
            if j == 0:
                cur = Lbkts[i]
            else:
                cur = Psi_s[i]

            if cur is None:
                continue
            else:
                Psi[prev] = cur

            while cur != None and cur != 0 and T[cur] == i:
                if T[cur - 1] - j >= T[cur] and cur != i:
                    if Lbkts[T[cur - 1]] is None:
                        Lbkts[T[cur - 1]] = cur - 1
                        Lbkte[T[cur - 1]] = cur - 1
                    else:
                        pree = Lbkte[T[cur - 1]]
                        Psi[pree] = cur - 1
            next_ = Psi[cur]
            prev = cur
            cur = next_

    return Psi

def compute_PSV_from_psi(PSV):
    cur = PSV[0]
    prev = 0

    while cur != 0:
        while cur < prev:
            prev = PSV[prev]
        next_ = PSV[cur]
        PSV[cur] = prev
        prev = cur
        cur = next_

    return PSV
```

Finally, with the computed Ψ, LZoneT can efficiently calculate the PSV using a single integer array and minimal additional workspace by rewriting Ψ in place. Similarly, Φ_ℓ enables the computation of the NSV, mirroring the efficiency seen in the KKP2 algorithm's methods. This approach allows for a single scan to compute either Φ or Ψ from their respective starting points, simplifying the process compared to BGone, which requires dual scans to sort and process both L-suffixes and S-suffixes. This is demonstrated in the following code.

The LZoneSA algorithm is a variant of LZoneT designed to efficiently compute the LZ factorization using a pre-existing SA of the input string S. Unlike LZoneT, which constructs SA_ℓ or SA_s directly from S, LZoneSA leverages the available SA to streamline this process. By counting the L-suffixes and S-suffixes for each character c as it scans T from right to left, LZoneSA efficiently determines the relevant segments in the SA, optimizing the use of available data structures and enhancing processing speed.

3.6. LZ *Construction with the* RLBWT *and Sparse Sampling*

Policriti and Prezza [6] presented an algorithm for converting the RLBWT to LZ77. The algorithm begins by constructing the RLBWT for the reversed S using an online construction algorithm. Next, the variables that will be used are initialized, which includes: (1) i, which is the current index in S; (2) k, which is the current run in the RLBWT containing j; (3) occ, which is the position in S where the current phrase prefix $S[i-\mathrm{len},..,i-1]$ occurs; (4) the current character at position $S[i] = \mathrm{RLBWT}[k]$ in the text; and an (5) an (inclusive) interval (l,r)corresponding to the current reversed LZ phrase prefix $S[i-len,..,i-1]$ in the RLBWT. An additional data structure is used to store a set SA samples. Next, each position of S is considered starting from 0, i.e., $i = 0,..,n$ and calculate the LZ77 factors for each of these positions as follows. First, it is determined whether current character (i.e., $S[i] = c$) ends a LZ phrase so we count the number of runs that intersect interval $[l,r]$ on RLBWT. We denote this as u If $u = 1$, then the current phrase prefix $S[i-len,..,i-1]$ is followed by c in S, and consequently $S[i]$ cannot be the last character of the current LZ phrase. Otherwise, we find a SA sample (i', j') such that $l \leq j' \leq r$ and update occ to be equal to $i' - len$. Lastly, the current LZ phrase prefix length is increased and the BWT interval is updated. If both the conditions fail then the current substring does not occur in $S[0,..,i-1]$ and therefore, is a LZ phrase. The position and length of the phrase is calculated using the defined variables. This algorithm computes the LZ factors in $\mathsf{O}(r \log n)$ working space and $\mathsf{O}(n \log r)$-time.

The authors note an alternative approach to computing and storing the SA samples more efficiently by focusing on LZ77 factors, and using the observation that the number of LZ77 factors z is often significantly smaller than the number of BWT runs r. The algorithm consists of three steps. Initially, it builds the RLBWT of the $\mathsf{rev}(S)$. During this process, the algorithm marks positions in the RLBWT that correspond to the end of LZ77 factors. Next, this algorithm scans S and uses the previously marked positions to determine the starting position of the LZ77 factors. Finally, this algorithm will rebuild the RLBWT and output the LZ77 factors by using the LZ77 factor's starting position identified previously. More specifically, for each position i in the string S, starting at 0. The algorithm will read the current character, i.e., $c = S[i]$. Then perform backward search in the RLBWT using the current interval $[l, r]$ (initialized as $[0, 0]$) and the character c. This will return a new interval $[l', r']$. If this interval $[l', r']$ is empty, it indicates that the substring formed by the current LZ77 factor cannot be extended further in S. Therefore, the algorithm will output the current LZ77 factor and initialize the interval $[l, r]$ to the $[0, i]$ and insert the character c into the RLBWT. If the interval $[l', r']$ is not empty, the algorithm will extend the current LZ77 factor and adjust $[l, r]$ accordingly.

3.7. PFP-LZ77

Given an input string S, Fischer et al. [21] demonstrated a data structure that supports access to SA_S, ISA_S, BWT_S, and LCP_S, and some additional queries on LCP_S can emulate a Compressed Suffix Tree (CST). Given that insight, Boucher et al. [22] described how the output of prefix-free parsing (PFP) can be used as the main components of a data structure that supports CST queries, and thus, simulate a CST. Here, we will briefly describe PFP, the components of the PFP data structure, and the queries that are supported by the data structure. Then apply it to the LZ77 construction. The discussion of the data structure is not meant to be exhaustive.

PFP takes as input a string $S[1..n]$, and positive integers w and p, and produces a parse of S (denoted as P) and a dictionary (denoted as D) of all the unique substrings (or phrases) of the parse. We briefly go over the algorithm for producing this dictionary and parse. First, we let T be an arbitrary set of w-length strings over Σ, and $T' = T \cup \{\$^w\}$ where \$ is a special symbol lexicographically less than any symbol in Σ. We call T' as the set of *trigger strings*. We append w copies of \$ to S, i.e., $S' = S\$^w$, and consider it to be cyclic, i.e., for all i, $S'[i] = S'[(i-1 \mod n) + 1]$.

We let the dictionary $\mathsf{D} = d_1, .., d_{|\mathsf{D}|}$ be equal to the maximum set of substrings of S' such that the following holds for each d_i: i) exactly one proper prefix of d_i is contained T', ii) exactly one proper suffix of d_i is contained in T', iii) and no other substring of d_i is in T'. Therefore, since we assume $\$^w \in T'$ we can construct D by scanning $\$^w S\$^w$ from right to left to find all occurrences of T' and adding to D each substring of $\$^w S\$^w$ that starts and ends at a trigger string being inclusive of the starting and ending trigger string. We can also construct the list of occurrences of D in S, which defines the parse P. Hence, to accomplish this we let f be the Karp-Rabin fingerprint of strings of length w and slide a window of length w over $\$^w S\$^w$. We define Q to be the sequence of substrings $r = S'[s, s+w-1]$ such that $f(r) \equiv 0 (\mod p)$ or $r = \$^w$ ordered by their position in S, i.e., $Q = (S'[s_1, s_1 + w - 1], .., S'[s_k, s_k + w - 1])$ where $s_1 < s_2 < .. < s_k$. The dictionary is the set of the unique substrings s of S' such that $|s| > w$ and the first and last w characters of s are consecutive elements in Q. Lastly, the dictionary is sorted in lexicographical order and the parse is defined to be the ranks of the substrings of D in sorted order.

Lastly, we illustrate PFP using an example. Suppose we have the following string

$$S' = \texttt{AGACGACT\#AGATACT\#AGATTCGAG\#\#}$$

, $w = 2$ and a set of trigger strings $T' = \{\texttt{AC}, \texttt{TC}, \texttt{\#\#}, \texttt{T\#}\}$.

Boucher et al. construct the following data structures from the parse P from the PFP of the input sequence S: (1) the suffix array of P (denoted as $\mathsf{SA_P}$), (2) the inverse suffix array of P (denoted as $\mathsf{ISA_P}$), (3) a cyclic bitvector (denoted as $\mathsf{B_P}$) that marks the start position of each trigger string in the string S, and (4) a bitvector (denoted as $\mathsf{B_{BWT}}$) that, for each unique proper phrase suffix of length at least w, has a 1 for the first position in the SA_S that stores that phrase. Lastly, Boucher et al. construct LCP of P but in terms of the phrases of P (not the ranks of D), which is denoted as $\mathsf{LCP_P}$.

Next, Boucher et al. construct the following data structures from the dictionary D: (1) the longest common prefix array of D (denoted as $\mathsf{LCP_D}$); and (2) a list of size D that, for each position in D, which starts with a specific proper suffix α, records the rank of α among the distinct proper suffixes of length a least w (denoted as $\mathsf{ISA_D}$).

An additional structure built from D and P by Boucher et al. is a two-dimensional discrete grid that we denote as W. W is used to support for SA_S and ISA_S queries. The x coordinates of W correspond to the entries of $\mathsf{BWT_P}$, and the y coordinates of W correspond to the phrases of D in co-lexicographic order. Given a phase p of the parse P, let $\mathsf{colex}(p)$ be the co-lexicographic rank of p in D. The set points in the grid are defined as $\mathsf{W} = \{(\mathsf{BWT_P}[i], \mathsf{colex}(\mathsf{BWT_P}[i])) \mid 1 \leq i \leq |\mathsf{P}|\}$. Given a position p and rank r, W can return how many phrases of the first p phrases in $\mathsf{BWT_P}$ have a co-lexicographic rank at least r.

We define D_w to be the set of all proper phrase suffixes of D of length at least w, and we denote $\mathsf{rev}(\mathsf{D}_w)$ to be the set of phrases of D_w reversed and stored in co-lexicographic order. Given these definitions, Boucher et al. define a table M where $\mathsf{M}[r]$ stores (1) the length ℓ of the lexicographically r-th proper suffix α of length at least w; and (2) the range in D_w of phrases starting with α reversed, starting with 0.

Using the dictionary D, the parse P and the auxiliary data structures discussed above, Boucher et al. demonstrated that a number of different queries can be supported, including SA, ISA, LCP, PSV, and NSV, among others. Using these queries, we can compute the LZ factorization.

Our method only considers positions $1 \leq i \leq n$ that are the start of an LZ factor. For each such position, we use the $\mathsf{B_P}$ to find the phrase q of P that contains position i in S (line 7). Next, we use $\mathsf{B_D}$ to find its offset within the phrase (line 8) and use this offset to identify the proper phrase suffix α to the phrase q in D.

Given a position d of α in D, we can compute $r = \mathsf{ISA_D}[d]$ that is equal to its lexicographic rank r among the phrases. Next, using M we can find the range $[y1, y2]$ of the phrases ending with α—this corresponds to our the boundary in T along the y-coordinate. We use $\mathsf{ISA_P}$ to find the phrase q of P in $\mathsf{BWT_P}$ order. We denote this as x (line 11). This defines the ranges of interest for which we consider the rightmost and leftmost point, which are $[1, x-1] \times [y1, y2] \times [1, q]$ and $[x+1, |\mathsf{P}|] \times [y1, y2] \times [1, q]$, respectively. Using these ranges and the definitions are described previously, we can compute $\mathrm{lce}(i, \mathsf{PSV}_{text}[i])$ and $\mathrm{lce}(i, \mathsf{PSV}_{text}[i])$. In our ongoing example, if i is 14 the $x = \mathsf{ISA_P}[h+1] = 5$ and $q = \mathsf{P}[\mathsf{B_P}.\mathsf{rank}(14)] = 3$. Then using $\mathsf{ISA_D}$ we find that $r = 5$, allowing us to compute the range $[y1, y2]$ to be $[2, 4]$. Within the region defined by $[0, 2], [2, 4], [0, 3]$, we see that the rightmost point is equal to $[3, 4, 2]$, which is $S[\mathsf{PSV}_{text}[14]..n]$. If $\mathsf{PSV}_{text}[i]$ exists then we define the phrase and phrase length using $\mathsf{LCP_P}$, $\mathsf{B_P}$ and $\mathsf{SA_P}$. (lines 8 and 9 of Algorithm LZ_Factor). Otherwise, we use $\mathsf{PSV_D}$, S_D and LCP_d to find the phrase and phrase length of longest previous factor within the region $\mathsf{PSV_D}[r] + 1, r$. We compute $\mathsf{NSV}_{text}[i]$ analogously (lines 14 - 19). Lastly, we compute the longest previous factor for position i as $\mathrm{lce}(\mathsf{PSV}_{text}[i], \mathsf{NSV}_{text}[i])$.

```
def LZ_Factor(PFP_data_structure, psv, nsv, x, r):
    l_psv = 0
    l_nsv = 0
    p_psv = 0
    p_nsv = 0
    o_q_prime = B_p.select(q + 1) - x
    if psv is not None:
        l_psv = 1 * lcpSA_p[psv.x + 1, x]
        p_psv = B_p.select(SA_p[psv.x]) - o_q'
    elif PSV_D[r] > 0:
        l_psv = 1 * lcpSA_D[PSV_D[r] + 1, r]
        p_psv = S_D[PSV_D[r]]

    if nsv is not None:
        l_nsv = 1 * lcpSA_p[x + 1, nsv.x]
        p_nsv = B_p.select(SA_p[nsv.x]) - o_q'
    elif NSV_D[r] > 0:
        l_nsv = 1 * lcpSA_D[r + 1, NSV_D[r]]
        p_nsv = S_D[NSV_D[r]]

    if l_psv > l_nsv:
        return (p_psv, l_psv)
    else:
        return (p_nsv, l_nsv)
```

3.8. Sublinear LZ77

To optimize the LZ77 factorization of S, Ellert [8] proposed a method that distinguishes between short phrases—i.e., length less than δ—and long phrases—i.e., length at least δ, where $\delta \in [1, |S|]$ is a parameter that is poly-logarithmic in $|S|$. In order to generate the LZ77 factor, S is traversed from left to right. For each position i during this traversal, if $i \in [1, \delta)$ then the algorithm computes the LCE(j, i) for each $j \in [1, i)$. Next, the largest value from the LCE array represents the length of the current LZ77 factor, and the associated j is its starting position. If all LCE values are zero, then the factor at position i is a literal phrase. For positions $i \geq \delta$, the algorithm applied three different methods depending on the position of the leftmost source j, which is unknown in advance. The algorithm thus computes each method and selects the one that produces the longest LZ77 factor.

The first method, referred to as *close sources*, is used if $j \in [i - \delta, i)$. This method computes the LCE for each $j' \in [i - \delta, i)$ and tracks the longest LCE, efficiently finding the best match within the nearby region. The second and third methods, referred to as *far sources*, are used if $j \in [1, i - \delta)$. These methods use two data structures, described following, to ensure efficient computation. If the phrase is long (length $\geq \delta$), a subset X of sampled positions from an array A of evenly spaced samples is used. The array A consists of distinct samples from $[1, |S|]$, $A[h] = h\delta < j + \delta < i$ with $h = \lceil j/\delta \rceil$. Sampling in this context means selecting specific positions in the text at regular intervals to create a representative subset that can be used to quickly find matching substrings. By querying X with the current position i and possible offsets, the algorithm efficiently identifies the best match. For each sample position $A[h]$, $h \in [1, |S|]$,the algorithm checks potential matches by comparing segments of the text, leveraging the precomputed samples to find the longest match. Specifically, for each offset k, the algorithm computes the lce between the substring starting at i and the substring starting at $A[h] - k$. This lce calculation helps determine if there is a longer matching substring that occurs earlier at position $A[h]$. The longest lce found indicates the length opf the LZ77 factor, and the corresponding $A[h]$ position, adjusted by the offset k is the starting position of the LZ77 factor.

The third method handles short phrases (length $\leq \delta$) by using a subset Y of samples from an array B. The array B is sorted in increasing order and designed to ensure that any segment of sufficient length includes at least one sample position. More specifically, for each sample position $B[h]$, the condition $B[h] \leq j + \mathit{factor\ length} \leq j + \delta < i$ is met, ensuring that the samples cover the relevant range of the text. For each sample position $B[h]$, the algorithm performs queries to compute the LCE for the current phrase. By querying Y, the algorithm computes the LCE for each sample point in B. By evaluating the LCE results, the algorithm can generate the LZ77 factors. These results are theoretical. Currently, there are no implementations of these algorithms.

3.9. The External KKP Algorithms

Kärkkäinen et al. [23] showed how the LZ parse can be constructed in external memory by giving the following external memory algorithms: Longest Previous Factor (EM_LPF), External Memory LZscan (EM_LZscan), and semi-external KKP (SE_KKP), each tailored to address the challenges of large-scale data processing beyond the constraints of internal memory. EM_LPF uses eSAIS [24] to compute the SA and LCP arrays in external memory. Following the computation of these arrays, EM_LPF employs the algorithm developed by Crochemore et al. [16] to execute the LZ factorization. A critical aspect of EM_LPF is its heavy reliance on disk I/O operations and substantial hard disk space requirements, estimated by the authors to be around $54n$ bytes. This significant demand for disk resources can be a limiting factor, especially in systems where disk performance or space is constrained. The EM_LZscan is an adaptation of the LZscan [25], which computes the LZ factorization based on matching statistics. This algorithm is particularly noted for its efficient handling of large datasets in external memory. By optimizing the original LZscan for external memory use, EM_LZscan provides a more feasible solution for processing large text corpora where internal memory limitations are a bottleneck. Lastly, SE_KKP represents a semi-external memory approach. It first computes the SA using the SAscan [26]. Following this, SE_KKP computes the PSV_{text} and NSV_{text}. Finally, the algorithm calculates the LZ factorization using PSV_{text} and NSV_{text}, and random access to the original input. This semi-external method strikes a balance between the use of internal and external memory, making it a versatile choice for varying data processing scenarios.

4. Experiments

We evaluated the performance of several previously described algorithms, specifically KKP2, RLE_LZ, PFP-LZ77, and SE-KKP. KKP1 and KKP3 were excluded from the evaluation due to their implementations being unable to support large datasets. Despite reaching out to the authors of the LZone algorithm, we did not receive a response, we cannot access to the source code. Moreover, there is currently no available implementation for the sublinear LZ77 algorithm. We selected SE-KKP to represent external memory LZ77 construction, as it demonstrated the best performance in external memory settings in a prior benchmark study [7].

4.1. Results on Pizza&Chili

All methods, except the RLE_LZ algorithm, required less than 5GB of memory and completed under 20 minutes, as shown in Figure 1. The RLE_LZ consistenly took the longest time but consumed the least memory. The SE-KKP algorithm always used more time and momory to both the PFP-LZ77 and KKP2 algorithms. Specifically, the SE-KKP consumed 32.98GB, 7.10GB, 36.51GB, 8.45GB, 11.54GB, 20.23GB, 31.30GB, and 2.47GB of hard disk space across different scenarios. Additionally, the performance of PFP-LZ77 and KKP2 is similar.

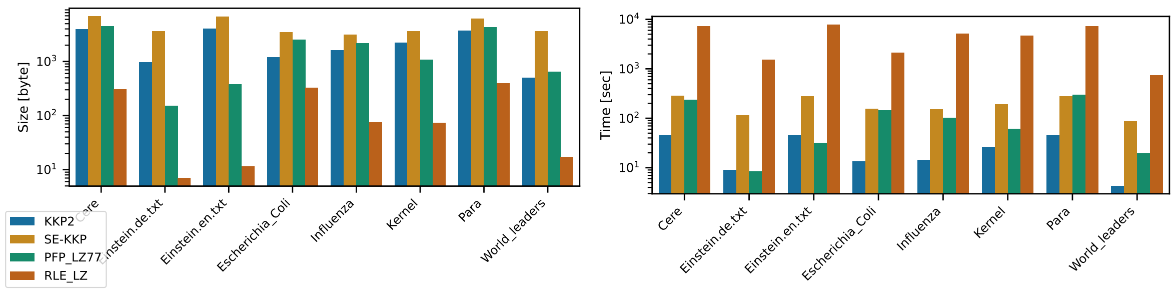

Figure 1. Time and memory required by all algorithms; the left graph shows the memory required to build the LZ77 factors, the graph on the right shows the time required to compute the LZ77.

4.2. Results on Salmonella

In this experiment, we set the time limit of 24 hours. For the dataset with 1,000 copies of Salmonella, the RLE_LZ algorithm was unable complete the experiment within the 24 hours. Although RLE_LZ consumed the least memory, it required the most time across all datasets. The SE-KKP algorithm consistently used the most memory and required 19.71GB, 38.78GB,211.03GB, and 428.60GB of hard disk space in various scenarios. KKP2 was the fastest algorithm in this experiment, and its memory usage was similar to that of PFP-LZ77.

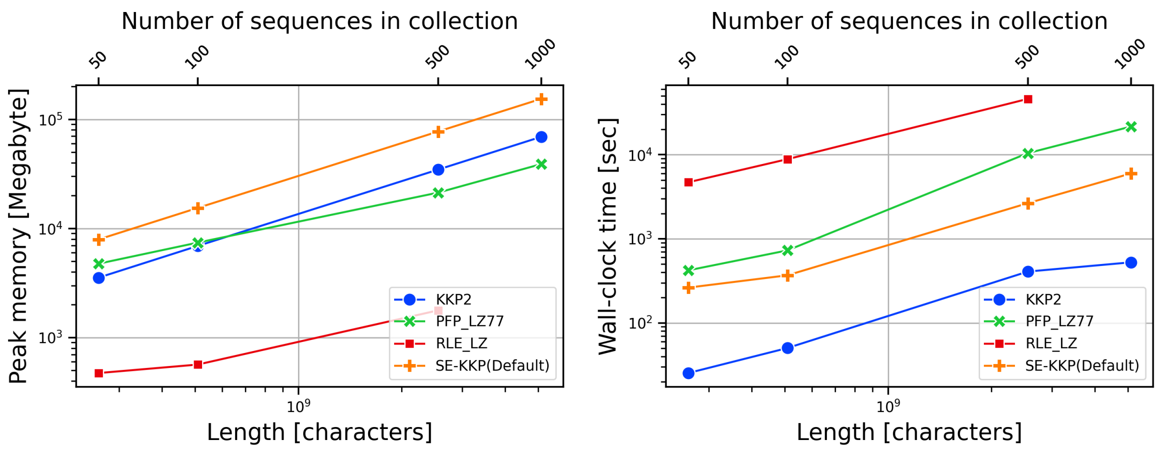

Figure 2. Time and Memory required by all the algorithms on the `Salmonella` dataset; the left graph shows the memory required to build the LZ77 factors, and the graph on the right shows the time required to compute the LZ77.

5. Conclusion and Future Work

Throughout this review, we have thoroughly explored the evolution and enhancement of the LZ77 algorithm and its derivatives, particularly focusing on their applications in genomic data compression. The continuous growth in genomic data necessitates efficient compression solutions that can effectively manage and utilize these vast datasets. Traditional LZ77 algorithms, while foundational in the realm of lossless data compression, require significant adaptations to meet the unique demands of genomic sequences characterized by their repetitive nature. The advancements discussed in this paper highlight significant strides in algorithmic innovation aimed at optimizing LZ77 for better handling of large-scale genomic data. We have examined a variety of specialized algorithms that not only improve compression ratios but also enhance processing efficiencies in both software and hardware implementations. These adaptations make it feasible to manage, store, and analyze genomic data more effectively, supporting the broader goals of genomic research and clinical applications.

Moreover, the integration of LZ77-based algorithms into hardware platforms represents a promising avenue for overcoming the throughput limitations of software-based compression. This approach can potentially unlock real-time processing capabilities, essential for applications requiring immediate data analysis, such as compression of pangenomic datasets. As we look to the future, the ongoing development of LZ-based compression technologies will undoubtedly play a critical role in the bioinformatics field. Continued innovations are expected to further refine these algorithms, making them more adaptable to the evolving landscape of genomic data. Additionally, exploring synergies between algorithmic improvements and hardware accelerations will be crucial in achieving the scalability and efficiency required for future applications in biological data compression, software distribution, streaming media, and medical imaging data. Below we highlight some specific areas that warrant future research.

5.1. Merging LZ77 Factorizations for Repeated Text

There are many large sequence datasets that grow rapidly, such as GenomeTrakr [27] and MetaSub [28], which update frequently. Currently, when applying LZ factorization to these datasets, we need to regenerate the LZ factorization each time the datasets are updated. This process is time-consuming and computationally expensive. However, if we can update the LZ77 factors generated from previous versions of the dataset, it would save a significant amount of time and computational resources. As described by [29] and [30], dynamic data structures and algorithms for merging BWT's have been successfully applied to address similar challenges in other contexts. These techniques can be adapted to the LZ77 construction process.

To implement this feature, we can split the input string into different chunks, compute the LZ77 factors for each chunk independently, and then combine them into a global factorization. When new data is appended to the input string, we can treat the appended part as a new chunk and merge its LZ77 factors with the previously generated global factors. Special attention must be given to the boundary conditions. Specifically, we need to check the start position of the last factor in the existing factorization to determine if the new appended characters affect this factor. By efficiently managing these updates and merging operations, we can maintain an up-to-date LZ77 factorization without the need to recompute everything from scratch, thereby significantly improving computational efficiency.

5.2. External Memory Construction

There has been very few external memory construction methods, and as datasets continue to grow substantially larger, the need for efficient external memory algorithms becomes increasingly critical. For instance, consider the `PFP-LZ77` algorithm. Previously, the PFP parsing for the input string S was computed in a single step within main memory. However, this process can be adapted to handle larger datasets by dividing the input string into manageable chunks that fit into main memory. Each chunk can then be processed independently using the `PFP-LZ77` algorithm to produce partial LZ77 factorizations. After processing all chunks, a merging phase can combine these partial factorizations into a global factorization. Multi-threading can be applied to this process, significantly speeding up the processing time. Furthermore, using disk-based data structures such as B-trees can facilitate the efficient search operations required by the LZ77 algorithm, enhancing the algorithm's performance in an external memory context.

5.3. GPU Implementation

Exploring the implementation of LZ77 on GPU is a promising direction for future research. GPUs are well-suited for parallel processing tasks due to their ability to handle many computations simultaneously. While some existing research has successfully applied GPUs to fundamental data structures like suffix arrays [31], more complex algorithms such as LZ have not been extensively explored. A key challenge in implementing LZ on GPUs is efficiently partitioning the input data and managing memory to ensure each GPU thread operates independently. Once approach is to divide the input string into smaller chunks that fit into GPU memory and process each chunk in parallel. After processing, the partial LZ77 factorizations can be merged into a global factorization.

Minimizing data transfer between the CPU and GPU and using asynchronous transfers can further enhance performance. Future research could focus on optimizing these parallel algorithms, exploring hybrid CPU-GPU approaches, and testing the implementation on real-world datasets to validate its effectiveness.

Data Availability Statement: We used the real-world datasets from the Pizza&Chili [32], Salmonella genomes obtained from the NCBI website. The Pizza&Chili repetitive corpus consists of repetitive texts that vary in length and alphabet size, can be accessed at https://pizzachili.dcc.uchile.cl/biblio.html. For the Salmonella datasets, We employed four variants containing 50, 100, 500, and 1,000 copies of the Salmonella genome sequence which is available at https://www.ncbi.nlm.nih.gov/datasets/genome/?taxon=590.

References

1. Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* **1978**, *24*, 530–536. doi:10.1109/TIT.1978.1055934.
2. Kreft, S.; Navarro, G. LZ77-Like Compression with Fast Random Access. Data Compression Conference (DCC), 2010, pp. 239–248. doi:10.1109/DCC.2010.29.
3. Kärkkäinen, J.; Kempa, D.; Puglisi, S.J. Linear Time Lempel-Ziv Factorization: Simple, Fast, Small. Combinatorial Pattern Matching (CPM), 2013, pp. 189–200.
4. Goto, K.; Bannai, H. Space Efficient Linear Time Lempel-Ziv Factorization for Small Alphabets. Data Compression Conference (DCC), 2014, pp. 163–172. doi:10.1109/DCC.2014.62.
5. Liu, W.J.; Nong, G.; Chan, W.h.; Wu, Y. Improving a lightweight LZ77 computation algorithm for running faster. *Software: Practice and Experience* **2016**, *46*, 1201–1217.
6. Policriti, A.; Prezza, N. LZ77 Computation Based on the Run-Length Encoded BWT. *Algorithmica* **2018**, *80*, 1986–2011. doi:10.1007/s00453-017-0327-z.
7. Hong, A.; Rossi, M.; Boucher, C. LZ77 via Prefix-Free Parsing. Symposium on Algorithm Engineering and Experiments (ALENEX), 2023, pp. 123–134. doi:10.1137/1.9781611977561.ch11.
8. Ellert, J. Sublinear Time Lempel-Ziv (LZ77) Factorization. String Processing and Information Retrieval (SPIRE), 2023, pp. 171–187.
9. Köppl, D. Computing LZ78-Derivates with Suffix Trees. Data Compression Conference (DCC), 2024, pp. 133–142.
10. Manber, U.; Myers, G.W. Suffix Arrays: A New Method for On-line String Searches. *SIAM Journal of Computing* **1993**, *22*, 935–948.
11. Burrows, M.; Wheeler, D.J. A Block-Sorting Lossless Data Compression Algorithm. Technical Report 0769518966, 1994.
12. Mäkinen, V.; Navarro, G. Run-length FM-index. *Burrows Wheeler Transform: Ten Years Later"(Aug. 2004)* **2004**.
13. Kärkkäinen, J.; Manzini, G.; Puglisi, S.J. Permuted Longest-Common-Prefix Array. Combinatorial Pattern Matching, 2009, pp. 181–192.
14. Ferragina, P.; Manzini, G. Opportunistic Data Structures with Applications. IEEE Symposium on Foundations of Computer Science (FOCS), 2000, pp. 390–398.
15. Ohlebusch, E.; Gog, S. Lempel-Ziv factorization revisited. Symposium on Combinatorial Pattern Matching (CPM), 2011, pp. 15–26.
16. Crochemore, M.; Ilie, L. Computing longest previous factor in linear time and applications. *Information Processing Letters* **2008**, *106*, 75–80.
17. Chen, G.; Puglisi, S.J.; Smyth, W. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science* **2008**, *1*, 605–623. doi:10.1007/s11786-007-0024-4.
18. Kempa, D.; Puglisi, S.J. Lempel-Ziv Factorization: Simple, Fast, Practical. SIAM Symposium on Algorithm Engineering and Experiments (ALENEX), 2013, pp. 103–112.
19. Goto, K.; Bannai, H. Simpler and faster Lempel Ziv factorization. IEEE Data Compression Conference (DCC), 2013, pp. 133–142.
20. Nong, G. Practical linear-time O(1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems* **2013**, *31*, 1–15.
21. Fischer, J.; Mäkinen, V.; Navarro, G. Faster Entropy-Bounded Compressed Suffix Trees. *Theoretical Computer Science* **2009**, *410*, 5354–5364.

22. Boucher, C.; Cvacho, O.; Gagie, T.; Holub, J.; Manzini, G.; Navarro, G.; Rossi, M. PFP Compressed Suffix Trees. SIAM Symposium on Algorithm Engineering and Experiments (ALENEX), 2021, pp. 60–72.
23. Kärkkäinen, J.; Kempa, D.; Puglisi, S.J. Lempel-Ziv Parsing in External Memory. IEEE Data Compression Conference (DCC), 2014, pp. 153–162. doi:10.1109/DCC.2014.78.
24. Bingmann, T.; Fischer, J.; Osipov, V. Inducing Suffix and LCP Arrays in External Memory. *ACM Journal of Experimental Algorithmics* **2016**, *21*. doi:10.1145/2975593.
25. Kärkkäinen, J.; Kempa, D.; Puglisi, S.J. Lightweight Lempel-Ziv Parsing. International Symposium on Experimental Algorithms (SEA), 2013, pp. 139–150.
26. Kärkkäinen, J.; Kempa, D. Engineering a Lightweight External Memory Suffix Array Construction Algorithm. *Mathematics in Computer Science* **2017**, *11*, 137–149. doi:10.1007/s11786-016-0281-1.
27. Timme, R.E.; Sanchez Leon, M.; Allard, M.W. Utilizing the Public GenomeTrakr Database for Foodborne Pathogen Traceback. In *Foodborne Bacterial Pathogens: Methods and Protocols*; Bridier, A., Ed.; 2019; pp. 201–212. doi:10.1007/978-1-4939-9000-9_17.
28. Mason, C.; Afshinnekoo, E.; Ahsannudin, S.; Ghedin, E.; Read, T.; Fraser, C.; Dudley, J.; Hernandez, M.; Bowler, C.; Stolovitzky, G.; Chernonetz, A.; Gray, A.; Darling, A.; Burke, C.; Labaj, P.P.; Graf, A.; Noushmehr, H.; Moraes, s.; Dias-Neto, E.; Ugalde, J.; Guo, Y.; Zhou, Y.; Xie, Z.; Zheng, D.; Zhou, H.; Shi, L.; Zhu, S.; Tang, A.; Ivanković, T.; Siam, R.; Rascovan, N.; Richard, H.; Lafontaine, I.; Baron, C.; Nedunuri, N.; Prithiviraj, B.; Hyat, S.; Mehr, S.; Banihashemi, K.; Segata, N.; Suzuki, H.; Alpuche Aranda, C.M.; Martinez, J.; Christopher Dada, A.; Osuolale, O.; Oguntoyinbo, F.; Dybwad, M.; Oliveira, M.; Fernandes, A.; Chatziefthimiou, A.D.; Chaker, S.; Alexeev, D.; Chuvelev, D.; Kurilshikov, A.; Schuster, S.; Siwo, G.H.; Jang, S.; Seo, S.C.; Hwang, S.H.; Ossowski, S.; Bezdan, D.; Udekwu, K.; Lungjdahl, P.O.; Nikolayeva, O.; Sezerman, U.; Kelly, F.; Metrustry, S.; Elhaik, E.; Gonnet, G.; Schriml, L.; Mongodin, E.; Huttenhower, C.; Gilbert, J.; Vayndorf, E.; Blaser, M.; Schadt, E.; Eisen, J.; Beitel, C.; Hirschberg, D.; Consortium, T.M.I. The Metagenomics and Metadesign of the Subways and Urban Biomes (MetaSUB) International Consortium inaugural meeting report. *Microbiome* **2016**, *4*, 24. doi:10.1186/s40168-016-0168-z.
29. Oliva, M.; Rossi, M.; Sirén, J.; Manzini, G.; Kahveci, T.; Gagie, T.; Boucher, C. Efficiently Merging r-indexes. IEEE Data Compression Conference (DCC), 2021, pp. 203–212. doi:10.1109/DCC50243.2021.00028.
30. Muggli, M.D.; Alipanahi, B.; Boucher, C. Building large updatable colored de Bruijn graphs via merging. *Bioinformatics* **2019**, *35*, i51–i60. doi:10.1093/bioinformatics/btz350.
31. Büren, F.; Jünger, D.; Kobus, R.; Hundt, C.; Schmidt, B. Suffix Array Construction on Multi-GPU Systems. International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2019, p. 183–194.
32. Pizza & Chili repetitive corpus. Available at http://pizzachili.dcc.uchile.cl/repcorpus.html. Accessed 16 April 2020.