technische universität
dortmund

Master Thesis

Implementing Sublinear-Time Approximation Algorithms for the Lempel-Ziv 77 Factorization

Lukas Nalbach

December 2024

Supervisors:
Prof. Dr. Johannes Fischer
Dr. Jonas Ellert

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Algorithm (LS-11)
https://ls11-www.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

The *Lempel-Ziv 77* (LZ77) factorization [34] is a text compression scheme that decomposes a string T into z phrases (factors), where each phrase is either a leftmost occurrence of a character (*literal phrase*) or the longest substring starting at an earlier position (*referencing phrase*). An earlier occurrence position of one such substring is called its source and its starting position is called its destination. Referencing phrases can be represented by one of its sources and its length, hence the factorization can be stored using $\mathcal{O}(z \log n)$ bits, where $n = |T|$ is the length of the string.

1.1 Relevance

The Lempel-Ziv 77 (LZ77) factorization [34] is the main text compression scheme and is also used to implement compressed text indexes [33, 19, 19]. It is also used in compressed media formats, such as PNG and PDF.

The number z of phrases can be used to measure the compressibility of a string, like the number r of runs in the *Burrows-Wheeler-Transformation* [6] (BWT), the number b of phrases in the smallest *Bidirectional Macro Scheme* [40], the size g of the smallest *Context-free grammar* [7] (CFG) generating the string and the number e of nodes and edges in the *Compact Deterministic Acyclic Word Graph* [5] (CDAWG). In practice, we have $b < z < g < r < e$, and there are orders of magnitude between the measures [37].

There are variants of the LZ77 factorization. *Rightmost Lempel-Ziv* [13] requires each factor to reference its rightmost source. *LZ-End* [30] demands each source to end at the end of a previous factor. *Sliding Window LZ77* [4] requires each source to lie in a fixed-sized window before its destination. *Relative Lempel-Ziv* (RLZ) [18] uses a reference (a string consisting of snippets from the input string) to copy referencing phrases from. Finally, *non-overlapping LZ77* [31] requires each source to end before its destination. There are also parallel LZ77 construction algorithms [8, 14] and LZ77 approximation algorithms [12, 16, 32].

1.2 The Problem

As we have seen, the LZ77 factorization of a length-n string over the alphabet $[1, \sigma]$ with $\sigma = n^{\mathcal{O}(1)}$ can be computed in $\mathcal{O}(n)$ time and words of space on a word RAM of width $w = \Theta(\log n)$. However, since T can be represented using $\Theta(n \log \sigma)$ bits of space, that is, $\Theta(n/\log_\sigma n)$ words of space, there may be algorithms that need only $\Theta(n/\log_\sigma n)$ time and words of space.

There is an algorithm [27] that takes $\mathcal{O}(n/\log_\sigma n + r\log^9 n + z\log n)$ time and $\mathcal{O}(n/\log_\sigma n + r\log^8 n)$ words of space. However, this algorithm is a purely theoretical result, as it requires many large data structures and has a practically inefficient running time. The algorithm presented in [29] takes $\mathcal{O}((n\log\sigma)/\sqrt{\log n})$ time and $\mathcal{O}(n/\log\sigma)$ space, but also requires impractical data structures. In [12], an algorithm that takes $\mathcal{O}(n/\log_\sigma n + z\log^{3+\epsilon} z)$ time and $\mathcal{O}(n/\log_\sigma n)$ words of space is presented, where $\epsilon > 0$ is a small constant. As an intermediary step, this algorithm computes an *LZ-like* factorization of at most $3z$ phrases, which is defined analogously to the LZ77 factorization, but with the relaxation that referencing phrases do not have to be of maximal length. This takes $\mathcal{O}(n/\log_\sigma n)$ time and words of space. Then, the LZ-like factorization is transformed into the LZ77 factorization in $\mathcal{O}(n/\log_\sigma n + z\log^{3+\epsilon} n)$ time and $\mathcal{O}(n/\log_\sigma n)$ words of space.

We call an algorithm that computes an LZ-like factorization with at most αz phrases an LZ77 α-approximation. The computation of the 3-approximation that is part of the LZ77 construction algorithm described in [12] is based on a *τ-string synchronizing set* (SSS), which will be formally defined in Section 2.7. Intuitively, a SSS is a subset (sampling) of positions in T such that (**i**) within two occurrences of a long substring (with length $\geqslant 2\tau$) the same (relative) positions are sampled and (**ii**) the sampling is uniformly dense across T.

1.2.1 LZ77 3-Approximation

Phase 1. The first phase uses a SSS of T with size $\mathcal{O}(n/\tau)$ and an $\mathcal{O}(1)$-time and $\mathcal{O}(n/\tau)$-words of space LCE data structure based on this SSS, where $\tau = \mathcal{O}(\log_\sigma n)$. This SSS can be constructed in $\mathcal{O}(n/\tau)$ time and words of space (see Theorems 4.3 and 8.11 in [28]). (**i**) is used to compute a gapped factorization of $z' \leqslant z$ factors with phrases starting at and referencing previous sampled positions.

Phase 2. Due to (**ii**), long gaps ($\geqslant 3\tau$) between phrases are periodic and can thus be efficiently shortened or closed entirely such that only short gaps remain. To accomplish this, a data structure to compute the shortest period of length-$m \leqslant \log_2 n/(2\lceil\log_2\sigma\rceil)$ patterns in $\mathcal{O}(1)$ time is used, which can be constructed in $\mathcal{O}(\sqrt{n}\log^3_\sigma n)$ time and words of space (see Lemma 2 in [12]).

Phase 3. To close the short gaps in phase three, a data structure to locate any short pattern ($\leqslant 3\tau$) is required. In [12], an $\mathcal{O}(n/\log n)$ space data structure to compute the

leftmost occurrence of a length-$m \leqslant \log_2 n/((2+\epsilon)\lceil\log_2 \sigma\rceil)$ pattern is proposed, where $\epsilon > 0$ is a small constant (see Lemma 3 in [12]). However, this approach only works in practice if τ is very small (for example $\tau = 4$ or $\tau = 8$), which increases the size of the SSS to an impractical level. Therefore, it remains to find a compact data structure (or more generally an index) to locate short patterns. This is the main challenge when implementing the 3-approximation.

1.2.2 LZ77 Exact Algorithm

The exact LZ77 Algorithm described in [12] at first computes an LZ77 3-Approximation using the algorithm from Section 1.2.1. Then, the exact factorization is computed incrementally from left to right.

At first, it constructs a special sampling of text positions and lexicographically and colexicographically sorts the samples. The task of finding the next phrase starting at position i can then be reduced to (**i**) computing pairs of sparse prefix- and sparse suffix array intervals, (**ii**) intersecting the samples contained in those and (**iii**) filtering out samples starting at or after position i.

(**i**) is achieved using binary searches. (**ii**) and (**iii**) can be reduced to the *insertion-only orthogonal range one-reporting* (IO-OROR) problem. The algorithm uses the data structure from Theorem 1 in [38] for IO-OROR. Although this data structure provides suitable theoretical construction- and query times (and space), it is inefficient in practice. Therefore, it remains to find a practical data structure for the IO-OROR problem.

1.2.3 Practical Compression Tool

The LZ77 3-Approximation and the exact LZ77 algorithm compress the text only by replacing repetitions with referencing phrases. In practice, the compression ratio can be further increased by omitting short referencing phrases and encoding their text segments using other methods. Relating this insight to the LZ77 3-Approximation, we could omit Phases 2 and 3 and instead compress the gaps using a standard compressor.

1.3 Results

Compared with the LZ77 LPF algorithm (a simple, but space-consuming exact LZ77 algorithm, see Section 2.4), which needs $8n$ if $n < 2^{31}$, or $16n$ additional bytes of memory, else, the LZ77 3-Approximation from Section 1.2.1 needs only $0.1n$ to $0.3n$ bytes of additional memory, is 0.9-10 (typically 3-10) times faster and achieves approximation ratios between 1 and 2 (typically 1.2-1.3).

The exact LZ77 algorithm from Section 1.2.2 needs $0.2n$ to $0.6n$ additional bytes of memory. As its theoretical running time suggests, its performance improves with the

repetitiveness of the text. In practice, it is only useful for repetitive texts. Additionally, we implemented a more space-consuming version of the algorithm, which needs 10-50% more memory, but is 1-2.2 times faster.

Finally, we implemented a practical compression tool called `ssszip`, which omits Phases 2 and 3 of the LZ77 3-Approximation, and instead compresses the gaps using `zstd`[1]. Compared with state-of-the-art compressors, `ssszip` uses a predictable amount of memory ($1.1n$ to $1.5n$ bytes) and is an order of magnitude faster than state-of-the-art compressors that yield similar compression rates. Although there are compressors that are faster and/or need less memory, they yield much lower compression rates, especially with repetitive texts.

[1] `https://facebook.github.io/zstd/`

Chapter 2

Preliminaries

2.1 Syntax Definitions

2.1.1 Arrays

For $i, j \in \mathbb{Z}$, we write $\{k \in \mathbb{Z} \mid i \leqslant k \leqslant j\} = [i, j] = (i-1, j] = [i, j+1)$. Let $A[1..n]$ be an array A of size n. For $i \in [1, n]$, $A[i]$ returns the i-th value in A. Similarly, for $i, j \in [1, n]$ with $i \leqslant j$, $A[i, j]$ denotes the (sub-)array $B[1..j-i+1] = [A[i], A[i+1], ..., A[j]]$. We also allow this notation when addressing sub-arrays, i.e, $A[i, j] = A(i-1, j] = A[i, j+1)$. We also interpret A as the set $\{A[1], ..., A[n]\}$. Finally, we can interpret A as an iterated function, i.e, $A^k[i] = A^{k-1}[A[i]]$, where $k > 1$ is an integer, and $A^1[i] = A[i]$.

2.1.2 Binary Search in Pseudocode

Let $\mathcal{B}$ = "**bin-search for max** $y \in X$ **such that** $\mathcal{P}$" be a block of pseudocode, where $X = \{x_1, ..., x_N\}$ with $x_1 < ... < x_N$, and $\mathcal{P}$ is pseudocode that takes one parameter y and calls either "**report true**;" or "**report false**;" (which causes the control flow to exit the scope of $\mathcal{P}$). Then, calling $\mathcal{B}$ implicitly calls Algorithm 1, which returns the maximum $y \in X$ such that $\mathcal{P}(y)$ reports **true** using a binary search.

In Algorithm 1, we maintain the current search interval $[b, e]$, which is in initialized with $[b, e] = [1, N]$. Let $m = \lfloor (b+e)/2 \rfloor + 1$ be the current candidate position. If $\mathcal{P}(x_m)$ calls "**report true**;", then we set $b \leftarrow m$. Else, we set $e \leftarrow m - 1$. As soon as $[b, e] = \varnothing$, we return $\perp$ (see line 9). Once we have $b = e$ and $\mathcal{P}(b')$ reports **true**, we return x_b (see line 6).

2.1.3 Exponential Search in Pseudocode

Let $\mathcal{B}$ = "**exp-search for max** $y \in X$ **such that** $\mathcal{P}$" be a block of pseudocode, where $X = \{x_1, ..., x_N\}$ with $x_1 < ... < x_N$, and $\mathcal{P}$ is pseudocode that takes one parameter y, and calls either "**report true**;" or "**report false**;" (which causes the control flow to exit

Algorithm 1: bin-search for max $y \in X = \{x_1, ..., x_N\}$ **such that** $\mathcal{P}$

```
1  if X = ∅ then
2      return ⊥;
3  [b, e] ← [1, N];
4  while [b, e] ≠ ∅ do
5      m ← ⌈(b + e)/2⌉;
6      if P(x_m) reports true then
7          b ← m;
8          if b = e then
9              return x_b
10     else
11         e ← m − 1;
12 return ⊥;
```

Algorithm 2: exp-search for max $y \in X = \{x_1, ..., x_N\}$ **such that** $\mathcal{P}$

```
1  if X = ∅ then
2      return ⊥;
3  p, s ← 1;
4  while true do
5      if P(x_p) reports true then
6          if p = N then
7              return x_N;
8          p ← max(N, p + s);
9          s ← 2 · s;
10     else
11         return bin-search for max
               y' ∈ {x_b, ..., x_{p−1}} such that P;
```

the scope of $\mathcal{P}$). Then, calling $\mathcal{B}$ implicitly calls Algorithm 2, which returns the maximum $y \in X$ such that $\mathcal{P}(y)$ reports **true** using a left-to-right exponential search followed by a binary search. More precisely, we maintain a step size s and candidate position p, initialized with $p, s = 1$. Then, we iteratively run through the following loop.

Loop. If $\mathcal{P}(x_p)$ reports **true**, then we either return x_N if $p = N$, or set $p \leftarrow \min(N, p + s)$ and $s \leftarrow 2 \cdot s$, and repeat the loop, else. If $\mathcal{P}(x_p)$ reports **false**, then we break out of the loop and return "**bin-search for max** $y' \in \{x_b, ..., x_{p-1}\}$ **such that** $\mathcal{P}$;".

2.1.4 Strings

We interpret strings as arrays. Let $\Sigma = [1, \sigma]$ be an alphabet, and let $T \in \Sigma^n$ be a string. For $i, j \in [1, n]$ with $j < i$, we denote $T[i, j]$ with the empty string ε. Substrings of the form $T[1, i]$ and $T[i, n]$ are called suffixes and prefixes of T, respectively. We write $T_i = T[i, n]$. A prefix/suffix of T is proper iff it is shorter than T. The reversed string of T is $\mathsf{rev}(T) = T[n]T[n-1]...T[1]$. Two strings S_1 and S_2 are equal iff $|S_1| = |S_2|$, and $\forall i \in [1, |S_1|] : S_1[i] = S_2[i]$.

2.1.5 Lexicographical Order

Let $\prec$ be the *lexicographic order* of strings, that is, it holds $S_1 \prec S_2$ iff either S_1 is a proper prefix of S_2, or $\exists i \in [1, \min(|S_1|, |S_2|)] : \; S_1[1, i) = S_2[1, i)$ and $S_1[i] \prec S_2[i]$. We write $S_1 \preceq S_2$ instead of $\neg(S_2 \prec S_1)$. We assume that all described algorithms work in the *word-RAM model* with word width $w = \Theta(\log n)$ [22].

Let $\mathsf{SA}[1..n]$ be the array storing the unique permutation of $[1, n]$ such that $T_{\mathsf{SA}[1]} \prec ... \prec T_{\mathsf{SA}[n]}$. Since it stores the lexicographic order among all suffixes of T, we call it the *suffix array* [35]. Let $\mathsf{SA}^{-1}[1..n]$ be the array that stores the inverse permutation of SA, i.e, $\forall i \in [1, n] : \mathsf{SA}^{-1}[\mathsf{SA}[i]] = i$. We call SA^{-1} the *inverse suffix array*.

2.1.6 Word-Packed Representation of Strings

We assume that the string $T[1..n]$ is given in fixed-width integer representation, i.e, the alphabet is $\Sigma = [1, \sigma]$ and each symbol is stored in $\lceil \log_2 \sigma \rceil$ bits. Hence, the string is stored as a bitstring of length $n \cdot \lceil \log_2 \sigma \rceil$ in $\mathcal{O}(n/\log_\sigma n)$ consecutive memory words. Note that we can obtain the length-$(k \cdot \lceil \log_2 \sigma \rceil)$ bitstring representing a length-k substring S in $\mathcal{O}(1 + k/\log_\sigma n)$ time by using bit-shifts and bit-wise logical operations. Such a bitstring can then be interpreted as an integer $\mathsf{int}(S) \in [1, 2^{k \cdot \lceil \log_2 \sigma \rceil}]$.

2.1.7 Rank- and Select Queries

Let $A[1..n]$ be an array of values from the universe $U = \{u_1, u_2, ..., u_{|U|}\}$. The query $\mathsf{rank}_v(A, i)$ with $v \in U$ and $i \in [1, n]$ then returns the number of occurrences of v in A up to position i, i.e, $\mathsf{rank}_v(A, i) = |\{j \mid A[j] = v, j \in [1, i]\}|$. The query $\mathsf{select}_v(A, i)$ with $v \in U$ and $i \in \mathsf{rank}_v(A, n)$ then returns the position in A of the i-th occurrence of v in A, i.e, $\mathsf{select}_v(A, i) = \min\{j \in [1, n] \mid \mathsf{rank}_v(A, j) = i\}$.

2.1.8 Successor- and Predecessor Queries

Let $U = \{u_1, u_2, ..., u_{|U|}\}$ be an ordered universe, i.e, $u_1 < u_2 < ... < u_{|U|}$, and let $X \subseteq U$. Then, we call $\mathsf{pred}_X(v) = \max\{u \in X \mid u \leqslant v\}$ of $v \in U$ the predecessor of v in X and $\mathsf{succ}_X(v) = \min\{u \in X \mid u \geqslant v\}$ of $v \in U$ the successor of v in X.

2.2 Lempel-Ziv 77 Factorization

The Lempel-Ziv 77 factorization of a string T is the decomposition of T into z phrases $f_1 f_2 ... f_z = T$ such that any given phrase $f_j = T[i, i + |f_j|)$ with $j \in [1, z]$ and $i = 1 + \sum_{j'}^{j-1} |f_{j'}|$ is either a leftmost occurrence of a single character (*literal phrase*), or otherwise the longest prefix of T_i that occurs before i (*referencing phrase*).

We call i the destination of f_j and an occurrence position of f_j before i a source of f_j. The factorization is usually represented by a sequence of z pairs $p_1, p_2, ..., p_z$. For $j \in [1, z]$, either $p_j = \langle s_j, l_j \rangle$ stores the length $l_j = |f_j|$ and a source s_j of f_j if f_j is a referencing phrase, or $p_j = \langle T[i], 0 \rangle$ if f_j is a literal phrase. An LZ-like factorization is defined analogously to the LZ77 factorization, but with the relaxation that referencing phrases do not have to be of maximal length. It holds $z = \mathcal{O}(n/\log_\sigma n)$ [34]. Now, we discuss two algorithms for computing the LZ77 factorization (see Section 2.3 and Section 2.4).

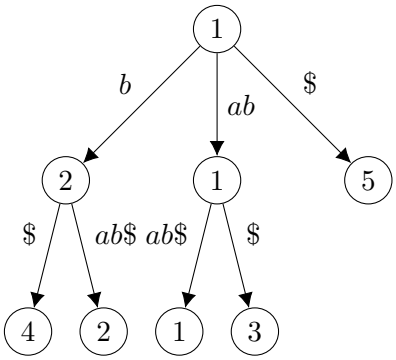


Figure 2.1: Suffix tree of $T = abab\$$, where each internal node v is marked with $\min(v)$.

2.3 LZ77 Suffix-Tree Algorithm

The LZ77 factorization can be computed using a *suffix tree*, which is the minimal tree such that for each suffix $T[i, n]$, there is a path $\langle r, ..., v\rangle$ from the root r to a leaf v, whose concatenation of edge labels is $T[i, n]$, and v is marked with i. Figure 2.1 shows an example suffix tree.

To compute the LZ77 factorization, we additionally mark each internal node v with the minimum leaf label $\min(v)$ in its subtree. Suppose we have already factorized T up to position $i - 1$. A next (j-th) factor starting at position i can then be computed by starting at the root and running down the suffix tree while matching $T[i, n]$ as long as the minimum leaf label in the current subtree is less than i ($\min(v) < i$). If there is no match, then the next factor $\langle 0, T[i]\rangle$ is literal (length 0 indicates that the phrase is literal). Else, let l denote the length of the matched substring, let v denote the node we have arrived at, and let $k = \min(v)$. Since l is the maximal length of a substring of T that starts at position i and also occurs before position i, and $T[k, k + l)$ is a previous occurrence of $T[i, i + l)$ (because $k = \min(v) < i$), we can choose $\langle k, l\rangle$ as a next factor.

Let v be a node in the suffix tree, and let s be the concatenation of the edge labels on the path from r to v. Then, let $\overline{s} = v$.

2.3.1 Example. To compute the LZ77 factorization of $T = abab\$$, we can use the suffix tree shown in Figure 2.1. The first factor is always $\langle 0, T[1]\rangle$, i.e, $\langle 0, a\rangle$. To compute the second factor ($i = 2$), we try to match $T[i] = b$ along the edge with label b going out from the root. However, we now have $\min(v) = \min(\overline{b}) = 2 \not< i = 2$, hence we have no match and output the next factor $\langle 0, T[i]\rangle = \langle 0, b\rangle$. Now, we match $T[3, 4]$ along the edge ab going out from the root and arrive at $v = \overline{ab}$, but we cannot match $T[5] = \$$, because $\min(\overline{ab\$}) = 3 \not< i = 3$, so we output $\langle \min(v), l\rangle) = \langle 1, 2\rangle$ as the third factor. Finally, we try to match $T[5]$ along the edge \$ outgoing from the root, but $\min(\overline{\$}) = 5 \not< i = 5$, so we output the final factor $\langle 0, T[i]\rangle = \langle 0, \$\rangle$. Overall, we get the LZ factorization $\langle 0, a\rangle, \langle 0, b\rangle, \langle 1, 2\rangle, \langle 0, \$\rangle$.

Since there are n leaves, and therefore $< n$ internal nodes in a suffix tree, it has less than $2n$ nodes and edges overall. Since every edge label occurs in T, edge labels can be represented by intervals $[b, e]$ to reference substrings $T[b, e]$, hence, each node and each edge need $\mathcal{O}(1)$ words of space, respectively. Therefore, a suffix tree needs overall $\mathcal{O}(n)$ words of space. It can also be constructed in $\mathcal{O}(n)$ time and words of space [15]. Suffix trees can be implemented in $\mathcal{O}(n)$ words of space such that a length-m pattern can be searched in $\mathcal{O}(m)$ expected time [21], where σ is the alphabet size of T. This results in an overall expected running time of $\sum_{j=1}^{z} \mathcal{O}(|f_j|) = \mathcal{O}(n)$ to compute the LZ factorization given a suffix tree, where f_j is the j-th factor, i.e, $T = f_1 f_2 ... f_z$.

2.3.1 Reaching Linear Time and Space

The running time can be improved to $\mathcal{O}(n)$ time by implementing two modifications. We store an array $\widetilde{\mathsf{SA}}^{-1}[1..n]$, where $\widetilde{\mathsf{SA}}^{-1}[i]$ points to the leaf with label i. Additionally, we annotate each node v with its string depth $d(v)$, which is the length of the concatenation of the edge labels on the path from r to v. A next (j-th) factor can then be found by starting at the node $v \leftarrow \widetilde{\mathsf{SA}}^{-1}[i]$, running up the suffix tree with v, and stopping as soon as $\min(v) \neq i$. If we have reached the root ($v = r$), then the next factor $\langle 0, T[i] \rangle$ is literal. Else, v is the lowest node v on the path from $\widetilde{\mathsf{SA}}^{-1}[i]$ to r with $\min(v) \neq i$, hence, $l = d(v)$ is the maximal length of a substring of T that starts at position i and also occurs before position i, i.e, at position $k = \min(v)$. Therefore, we can choose $\langle k, l \rangle$ as a next factor.

During the computation of the j-th factor starting at position i, we iterate upwards only from nodes u with $\min(u) = i$, and because factor starting positions are distinct, we iterate upwards from each node at most once. Since there are $\mathcal{O}(n)$ nodes, this results in an overall $\mathcal{O}(n)$ running time.

2.4 LZ77 LPF Algorithm

Compared with LZ77 construction algorithms that use the suffix tree (see Section 2.3), the *longest previous factor* (LPF) LZ77 construction algorithm stores only four integer arrays $\mathsf{SA}[1..n]$, $\mathsf{SA}^{-1}[1..n]$, $\mathsf{NSV}[1..n]$ and $\mathsf{PSV}[1..n]$. Asymptotically, this still requires $\mathcal{O}(n)$ words of space, but is more space-efficient than a suffix tree in practice.

2.4.1 Definition. Let $\mathsf{PSV}[1..n]$ be the array such that $\mathsf{PSV}[i]$ stores the previous smaller value (PSV) in SA before position i, i.e, $\mathsf{PSV}[i] = 0$ if $\forall j \in [1, i) : \mathsf{SA}[j] > \mathsf{SA}[i]$ and $\mathsf{PSV}[i] = \max\{j \mid j \in [1, i), \mathsf{SA}[j] < \mathsf{SA}[i]\}$, else. Analogously to PSV, let $\mathsf{NSV}[1..n]$ be the array such that $\mathsf{NSV}[i]$ stores the next smaller value (NSV) in SA after position i, i.e, $\mathsf{NSV}[i] = 0$ if $\forall j \in (i, n] : \mathsf{SA}[j] > \mathsf{SA}[i]$ and $\mathsf{NSV}[i] = \min\{j \mid j \in (i, n], \mathsf{SA}[j] < \mathsf{SA}[i]\}$, else.

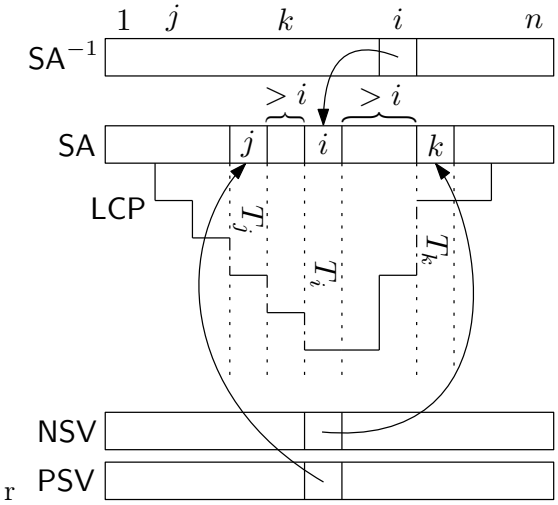


Figure 2.2: Illustration of the LPF-Array Construction. In this case, it holds $\mathsf{LCE}(j,i) > \mathsf{LCE}(k,i)$, hence $\mathsf{LPF}[i] = j$.

2.4.2 Definition. Given two text positions $i, j \in [1,n]$, let their *longest common extension* be the length of the longest substring starting at both i and j, i.e, $\mathrm{LCE}(i,j) = \max\{l \in [0, n-\max\{i,j\}+1] \mid T[i,i+l) = T[j,j+l)\}$. We call an array $\mathsf{LPF}[1..n]$ a *longest previous factor array* iff for each $i \in [1,n]$, either $\mathsf{LPF}[i] = \perp$ if $T[i]$ does not occur in $T[1,i)$, or else, $\mathsf{LPF}[i] = \arg\max_{j \in [1,i)}\{\mathsf{LCE}(j,i)\}$.

Suppose we have already factorized the text up to position $i-1$, and now want to compute the next x-th factor. The main insight of the LPF algorithm is that if $T[i]$ occurs before position i in T, then among the suffixes of T starting before i, either the lexicographically next smaller suffix T_j or the lexicographically next larger suffix T_k (or both) shares a prefix of length $|f_x|$ with T_i (see Figure 2.2). This is because if there was a suffix T_l with $l < i$ that shares a longer prefix with T_i, then T_l would lie lexicographically between T_j and T_k, which contradicts the choice of j or k. Hence, we can compute LPF as follows.

2.4.3 Construction. *Let $i \in [1,n]$, let $j' = \mathsf{PSV}[\mathsf{SA}^{-1}[i]]$ and $k' = \mathsf{NSV}[\mathsf{SA}^{-1}[i]]$. The following construction yields an* LPF *array. If $j', k' = 0$, then $\mathsf{LPF}[i] = \perp$. Else, $\mathsf{LPF}[i] = \arg\max_{s \in \{j',k'\}\setminus\{0\}}\{\mathsf{LCE}(\mathsf{SA}[s], i)\}$.*

We can compute SA in $\mathcal{O}(n)$ time [39]. Note that the definition of SA^{-1} is an $\mathcal{O}(n)$ time construction algorithm for SA^{-1}. We can compute NSV and PSV with a left-to-right scan over SA (see Algorithm 3). Suppose we have computed PSV up to position $i-1$. If $\mathsf{SA}[i-1] < \mathsf{SA}[i]$, then $\mathsf{PSV}[i] = i-1$. Else, $\mathsf{PSV}[i] = \mathsf{PSV}^l[i-1]$, where l is minimal such that $\mathsf{PSV}^l[i-1] < \mathsf{SA}[i]$ (see Figure 2.3). We compute l in lines 3-4 in Algorithm 3. Let $k \in [0,l)$. By the definition of PSV, we have $\mathsf{SA}[\mathsf{PSV}^l[i-1]] < ... < \mathsf{SA}[\mathsf{PSV}^1[i-1]] <$

Algorithm 3: nsv-psv()

```
1 for i from 1 to n do
2     p ← i − 1;
3     while p > 0 ∧ SA[p] > SA[i] do
4         NSV[p] ← i;
5         p ← PSV[p];
6     PSV[i] ← p;
7     NSV[i] ← 0;
```

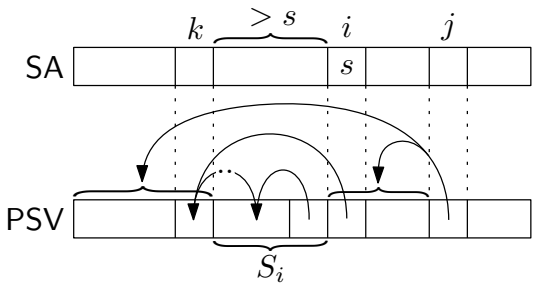


Figure 2.3: Illustration of a skipped interval during the computation of PSV.

$\mathsf{SA}[i-1]$ and $\forall x \in (k,i) : \mathsf{SA}[x] > \mathsf{SA}[\mathsf{PSV}^k[i-1]] > \mathsf{SA}[i]$, hence $\mathsf{NSV}[\mathsf{PSV}^k[i-1]] = i$, so we set $\mathsf{NSV}[p] \leftarrow i$ in line 24. It is easy to see that each entry $\mathsf{NSV}[p] \neq 0$ is assigned. Setting $\mathsf{NSV}[i] \leftarrow 0$ in line 7 ensures that the remaining entries (with $\mathsf{NSV}[p] = 0$) are correctly set.

2.4.4 Lemma. *Given* SA*, we can compute* PSV *and* NSV *in* $\mathcal{O}(n)$ *time.*

Proof. For $i \in [1,n]$, we call $S_i = (k,i)$ the i-th skipped interval, where $k = \mathsf{PSV}[i]$. We call it skipped, because it is skipped during the computation of each next entry $\mathsf{PSV}[j]$ with $j > i$. This is because each entry $\mathsf{PSV}[x]$ with $x \in [1,j) \backslash S_i$ does not point into S_i, i.e, $\mathsf{PSV}[x] \notin S_i$ (see Figure 2.3). Therefore, each position in $[1,n]$ is considered at most twice before it becomes part of some skipped interval, and the running time bound follows. □

2.4.5 Theorem. *Given* SA*,* SA^{-1}*,* PSV *and* NSV*, we can compute the LZ77 factorization in* $\mathcal{O}(n)$ *time and space.*

Proof. Suppose we have already factorized T up to position $i-1$. Intuitively, $\mathsf{SA}^{-1}[i]$ is the lexicographical rank of T_i, or the position of the suffix i in the suffix array, thus we can compute $\mathsf{LPF}[i]$ by Construction 2.4.3. If $\mathsf{LPF}[i] = \perp$, then we output a literal factor $\langle 0, T[i]\rangle$. Else, we output a referencing factor $\langle \mathsf{LPF}[i], \mathsf{LCE}(\mathsf{LPF}[i], i)\rangle$. We compute the LCE queries naively, which takes $\mathcal{O}(|f_x|)$ time for the x-th factor. The remaining computations (lookups in SA^{-1}, PSV and NSV) take $\mathcal{O}(1)$ time per factor, hence the

overall running time is $\sum_{x=1}^{z} \mathcal{O}(|f_x|) = \mathcal{O}(n)$. Since all used data structures are arrays of size n, we need $\mathcal{O}(n)$ words of space. □

Note that we do not explicitly compute the whole LPF-Array. Rather, we implicitly compute LPF only for phrase starting positions. Computing a whole LPF-Array would take $\mathcal{O}(n^2)$ time using the same method.

2.4.6 Example. Consider the text $T = abab\$$.

	1	2	3	4	5
$T =$	a	b	a	a	\$
$\mathsf{SA} =$	5	3	1	4	2
$\mathsf{SA}^{-1} =$	3	5	2	4	1
$\mathsf{PSV} =$	0	0	0	3	3
$\mathsf{NSV} =$	2	3	0	5	0

Since the first factor is always literal, we output $\langle 0, T[1]\rangle = \langle 0, a\rangle$. Now, we look up the position $i' = \mathsf{SA}^{-1}[2] = 5$ of T_2 in SA. We have $\mathsf{NSV}[5] = 0$, so there is no suffix before T_2 that is lexicographically larger than T_2. We have $j = \mathsf{PSV}[i'] = \mathsf{PSV}[5] = 3$ and $\mathsf{SA}[j] = \mathsf{SA}[3] = 1$, but T_1 does not yield a match with T_2, hence we output a literal factor $\langle 0, T[2]\rangle = \langle 0, b\rangle$. Now, we look up $i' = \mathsf{SA}^{-1}[4] = 4$. We have $\mathsf{NSV}[i'] = \mathsf{NSV}[4] = 5$, but $T_{\mathsf{SA}[5]} = T_2$ does not yield a match with T_3. However, $T_{\mathsf{SA}[\mathsf{PSV}[i']]} = T_1$ yields a match of length 2 with T_3, so we output $\langle 1, 2\rangle$ as a referencing factor. The last factor is $\langle 0, \$\rangle$. We have $z = 4$.

2.5 Constant Time LCE Queries in Linear Time and Space

To answer an LCE query in $\mathcal{O}(1)$ time, we introduce the longest common prefix array and range minimum queries.

2.5.1 Definition. Let $\mathsf{LCP}[1..n]$ be the array such that $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = \mathsf{LCE}(\mathsf{SA}[i - 1], \mathsf{SA}[i])$ for $i \in [2, n]$. We call LCP the *longest common prefix array* of T. Given an integer array $A[1..m]$, the *range minimum query* $\mathsf{RMQ}_A(i, j)$ returns the position in A of the minimum value in $A[i, j]$ for a given range $\varnothing \neq [i, j] \subseteq [1, m]$, i.e, $\mathsf{RMQ}_A(i, j) = \arg\min_{k\in[i,j]}\{A[k]\}$.

2.5.2 Theorem. *Given* LCP, SA^{-1} *and an* $\mathcal{O}(n)$ *space data structure to answer* $\mathsf{RMQ}_{\mathsf{LCP}}$ *queries in* $\mathcal{O}(1)$ *time, we can answer* LCE *queries in* $\mathcal{O}(1)$ *time.*

Proof. The LCP array and an $\mathcal{O}(n)$ space data structure to answer $\mathsf{RMQ}_{\mathsf{LCP}}$ queries in $\mathcal{O}(1)$ time can be computed in $\mathcal{O}(n)$ time [26, 17]. Now, we describe how the LCP array

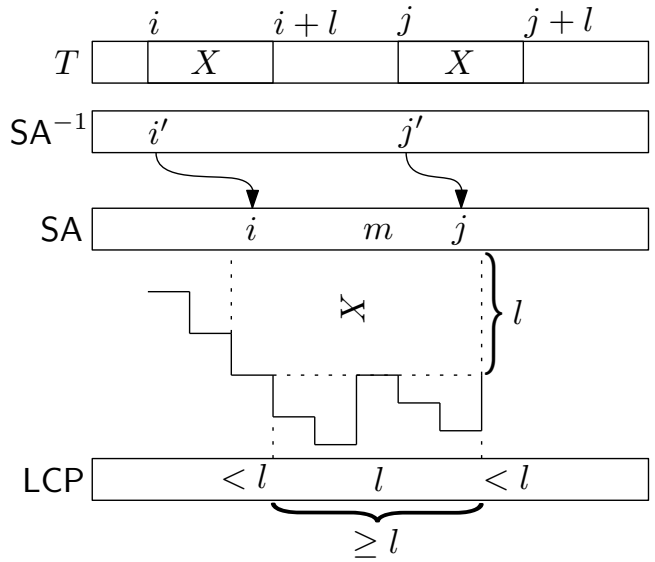


Figure 2.4: Illustration of LCE queries using RMQ queries on the LCP Array.

can be used together with an $\mathsf{RMQ}_{\mathsf{LCP}}$ data structure and SA^{-1} to answer LCE queries in $\mathcal{O}(1)$ time.

Let $i, j \in [1, n]$, let $l = \mathsf{LCE}(i, j)$. Now consider the positions $i' = \mathsf{SA}^{-1}[i]$ and $j' = \mathsf{SA}^{-1}[j]$ of i and j in SA (see Figure 2.4), and assume $i' < j'$ (else swap i and j). Let T_k be a suffix that lies lexicographically between T_i and T_j, i.e, $T_i \prec T_k \preceq T_j$. Since T_k shares a common prefix of length l with T_i and T_j, it holds $\mathsf{LCP}[\mathsf{SA}^{-1}[k]] \geqslant l$. Furthermore, there exists a suffix T_m such that $T_i \prec T_m \preceq T_j$ and $\min(\mathsf{LCE}(m, i), \mathsf{LCE}(m, j)) = l$ (otherwise would imply $\mathsf{LCE}(i, j) > l$), hence $\mathsf{LCP}[\mathsf{SA}^{-1}[m]] = l \vee \mathsf{LCP}[\mathsf{SA}^{-1}[m]+1] = l$ and therefore $\mathsf{LCP}[\mathsf{RMQ}_{\mathsf{LCP}}(i' + 1, j')] = l = \mathsf{LCE}(i, j)$. □

2.5.3 Example. Suppose we want to compute $\mathsf{LCE}(i, j) = \mathsf{LCE}(1, 3)$ in the text from Example 2.4.6.

	1	2	3	4	5
$T =$	a	b	a	b	\$
$\mathsf{SA}^{-1} =$	3	5	2	4	1
$\mathsf{LCP} =$	0	0	2	0	1

Here, we have $i' = \mathsf{SA}^{-1}[1] = 3$, $j' = \mathsf{SA}^{-1}[3] = 2$ and $\mathsf{RMQ}_{\mathsf{LCP}}(j' + 1, i') = \mathsf{RMQ}_{\mathsf{LCP}}(3, 3) = 3$, hence $\mathsf{LCE}(1, 3) = \mathsf{LCP}[3] = 2$.

2.6 Rolling Karp–Rabin Fingerprinting

Karp–Rabin fingerprints [25] can be used to efficiently hash substrings of the text to small integers such that (**i**) same substrings result in the same hash and (**ii**) there are only "few" hash collisions.

2.6.1 Definition. For a constant $c > 1$, a prime number $q = \Theta(n^c)$ and a base $b \in [\sigma, q)$, the Karp-Rabin fingerprint of the substring $T[i, j]$ is defined as $\phi(i, j) = (\sum_{k=i}^{j} T[k] \cdot b^{j-k}) \bmod q$.

Note that (**i**) holds, because the fingerprint is independent of i and j, i.e, $T[i, j] = T[i', j'] \Rightarrow \phi(i, j) = \phi(i', j')$. However, $T[i, j] \neq T[i', j'] \Rightarrow \phi(i, j) \neq \phi(i', j')$ holds only with high probability. Rolling fingerprinting is a method to compute, for a fixed substring length l, the Karp–Rabin fingerprints of all length-l substrings of T from left to right in $\mathcal{O}(n)$ time. Here, we initially compute $\phi(1, l)$. Then, we slide a length-l window over T and maintain at position $i \in [2, n - l)$ the fingerprint $\phi(i, i + l - 1)$ of the substring $T[i, i+l-1)$ covered by the window, which can be computed from the previous fingerprint $\phi(i - 1, i + l - 2)$ in $\mathcal{O}(1)$ time using a lookup table [25].

2.7 String Synchronizing Set

We say that a string is of period p iff it can be written as consecutive repetitions of its first p characters (the last repetition can be cut off). More formally, a string T is of period p iff $T[1, n - p] = T[1 + p, n]$. The period $\mathsf{per}(T)$ of T is then the minimal p such that T is of period p.

2.7.1 Definition (Definition 3.1 [28]). Given a string $T \in \Sigma^n$ and a parameter $\tau \in [1, \lfloor \frac{n}{2} \rfloor]$, a set $S = \{p_1, p_2, ..., p_s\} \subseteq [1, n-2\tau+1]$ with $p_1 < p_2 < ... < p_s$ is *τ-synchronizing* w.r.t. T iff the following two conditions hold.

- *Synchronizing condition*: For all $i, j \in [1, n-2\tau+1]$ with $T[i, i+2\tau) = T[j, j+2\tau)$, it holds $i \in S \Leftrightarrow j \in S$.
- *Density condition*: For every $i \in [1, n - 3\tau - 2]$, it holds $S \cap [i, i + \tau) = \varnothing$ iff $\mathsf{per}(T[i, i + 3\tau - 2]) \leqslant \frac{\tau}{3}$.

Now, let us describe a practical sliding-window algorithm to compute a τ-synchronizing set using a left-to-right scan over T. To this end, we need the following Theorem.

2.7.2 Theorem (Lemma 8.2 [28]). *For $\tau \in [1, \lfloor \frac{n}{2} \rfloor]$, let Q be the set of periodic text positions, i.e, $Q = \{i \in [1, n - \tau + 1] \mid \mathsf{per}(T[i, i + \tau)) \leqslant \frac{1}{3}\tau\}$. Let id be a function such that $\forall i, j \in [1, n] : \mathsf{id}(i) = \mathsf{id}(j) \Leftrightarrow T[i, i + \tau) = T[j, j + \tau)$. Then, $S = \{i \in [1, n - 2\tau + 1] \mid \min\{\mathsf{id}(j) \mid j \in [i, i + \tau] \backslash Q\} \in \{\mathsf{id}(i), \mathsf{id}(i + \tau)\}\}$ is τ-synchronizing w.r.t. T.*

We choose $\mathsf{id}(i) = \phi(i, i+\tau-1)$ and at first compute the set Q. Then, we use rolling Karp-Rabin fingerprinting with length τ, slide a window of size $\tau+1$ over T up to position $n-2\tau+1$, at position i keep track of all fingerprints $F_i = [\mathsf{id}(i), ..., \mathsf{id}(i+\tau)]$ in the current window $[i, i+\tau]$. If $\min\{\mathsf{id}(j) \mid j \in [i, i+\tau] \backslash Q\} \in \{\mathsf{id}(i), \mathsf{id}(i+\tau)\}$, then we add i to S. In practice, this algorithm yields a SSS of size $\approx \frac{2\tau}{n}$.

2.7.3 Example. Let $\tau = 3$ and let $\mathsf{id}(i)$ be the lexicographical rank of $T[i, i+\tau)$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b
$\mathsf{id} =$	5	15	18	4	14	17	2	12	12	12	12	12	12	11	10	9	8	7	5	15	18	3	13	16	1	6

We have $Q = [8, 16]$. For $i = 1$, we have $\min\{\mathsf{id}(j) \mid j \in [1, 4] \backslash [8, 13]\} = \min\{5, 15, 18, 4\} \in \{5, 4\}$, hence $i = 1 \in S$. The next included position is 4, because for $i = 4$, we have $\min\{\mathsf{id}(j) \mid j \in [4, 7] \backslash [8, 13]\} = \min\{4, 14, 17, 2\} \in \{4, 2\}$. The remaining positions are 14, 15, 16 and 19. Thus, we have $S = \{1, 4, 14, 15, 16, 19\}$.

2.8 Constant Time LCE Queries in Sublinear Time and Space

If we set $\tau = \mathcal{O}(\log_\sigma n)$, then we can use a τ-synchronizing set to answer LCE queries in $\mathcal{O}(1)$ time and $\mathcal{O}(n/\log_\sigma n)$ space. To this end, we need the following data structures.

2.8.1 Definition. Given a τ-synchronizing set S, let $T' \in [0, 3\tau)^{|S|}$ such that $T'[i] = T[p_i, \min(n, p_i + 3\tau))$ for $i \in [1, |S|]$. Let $\mathsf{LCP'}$ be the LCP array of T'.

2.8.2 Definition. Let $S = [p_1, p_2, ..., p_{|S|}] \subseteq [1, n]$ with $p_1 < p_2 < ... < p_{|S|}$ be a subset (sampling) of text positions from T (e.g. a SSS). We define the unique permutation $\mathsf{SA}_S[1..|S|]$ of $[1, |S|]$ satisfying $\forall i, j \in [1, |S|] : \mathsf{SA}_S[i] < \mathsf{SA}_S[j] \Leftrightarrow \mathsf{SA}[p_i] < \mathsf{SA}[p_j]$ to be the *sparse suffix array* of T w.r.t. S. Finally, Let the unique permutation $\mathsf{SA}_S^{-1}[1..|S|]$ of $[1, |S|]$ with $\forall i \in [1, |S|] : \mathsf{SA}_S[\mathsf{SA}_S^{-1}[i]] = i$ be the sparse *inverse suffix array* of T w.r.t. S.

The following lemma is the main useful property of string synchronizing sets and allows us to answer LCE queries by applying the same method as in Theorem 2.5.2, but using $\mathsf{LCP'}$ and $\mathsf{RMQ}_{\mathsf{LCP'}}$.

2.8.3 Lemma (Lemma 4.2 [28]). *For $i, j \in [1, |S|]$, it holds $T'_i \prec T'_j \Leftrightarrow T_{p_i} \prec T_{p_j}$.*

2.8.4 Theorem (Theorem 5.4 [28]). *We can compute in $\mathcal{O}(n/\log_\sigma n)$ time and space a data structure of size $\mathcal{O}(n/\log_\sigma n)$ that can answer LCE queries in $\mathcal{O}(1)$ time.*

Proof. We choose τ sufficiently small such that a word-packed length-3τ substring of T can be stored in one word, i.e, $3\tau\lceil\log_2 \sigma\rceil \leqslant w \Leftrightarrow \tau \leqslant \omega/(3\lceil\log_2 \sigma\rceil) = \mathcal{O}(\log_\sigma n)$ and

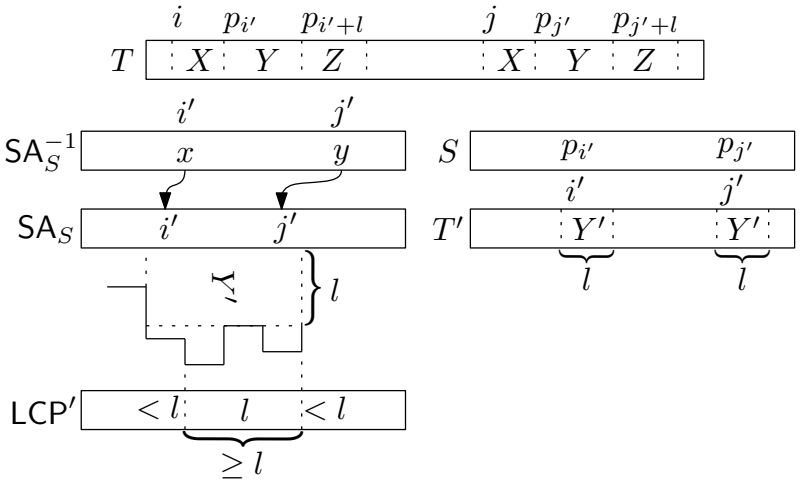


Figure 2.5: Illustration of LCE queries using RMQ queries on the LCP_S Array.

start by constructing T' in $\mathcal{O}(n/\tau) = \mathcal{O}(n/\log_\sigma n)$ time and space from the word-packed representation of T. Then, we build SA_S. Since by Lemma 2.8.3, SA_S is also the suffix array of T', we can do this in $\mathcal{O}(n/\tau) = \mathcal{O}(n/\log_\sigma n)$ time and space with an $\mathcal{O}(n)$ time suffix array construction algorithm. Then, we compute SA_S^{-1}, LCP' and an $\mathcal{O}(1)$ time $\mathsf{RMQ}_{\mathsf{LCP}'}$ query data structure in $\mathcal{O}(n/\tau)$ time and space using Definition 2.8.2, [26] and [17], respectively. Finally, we construct a packed bit-vector $B_S[1..n]$, where $B_S[i] = 1 \Leftrightarrow i \in S$ and augment it in $\mathcal{O}(n/\log_\sigma n)$ time with an $\mathcal{O}(1)$ time and $o(n/\log_\sigma n)$ space $\mathsf{rank}_1(B_S, .)$ query data structure (Proposition 2.4 [28]) to answer succ_S queries in $\mathcal{O}(1)$ time. Overall, the construction needs $\mathcal{O}(n/\tau) = \mathcal{O}(n/\log_\sigma n)$ time and space.

Since substrings of T with length $\leqslant 3\tau$ fit into one word, we can compute LCE queries on T up to length 3τ in $\mathcal{O}(1)$ time by $\mathsf{LCE}_{\leqslant 3\tau}(i,j) = \lfloor \mathsf{ctz}(\mathsf{int}(T[i, i+3\tau)) \oplus \mathsf{int}(T[j, j+3\tau)))/\lceil \log_2 \sigma \rceil \rfloor$, where $\oplus$ is the bitwise exclusive or operation and $\mathsf{ctz}(B)$ returns the number of trailing zeros of the bit-string B (here, we interpret $\mathsf{int}(T[i, i+3\tau))$ and $\mathsf{int}(T[j, j+3\tau))$ as bit-strings).

Now, we discuss the query procedure (see Algorithm 4). At first, we check if T_i and T_j mismatch within the first 3τ characters (see line 1-2). If so ($\mathsf{lce}_{\leqslant 3\tau} < 3\tau$), then we can return $\mathsf{LCE}(i,j) = \mathsf{lce}_{\leqslant 3\tau}$. Else, we compute the successors $p_{i'}$ and $p_{j'}$ of i and j in S together with their indices i' and j' in S. If $p_{i'} - i \neq p_{j'} - j$ holds (see line 6), then we return $\min\{p_{i'} - i, p_{j'} - j\} + 2\tau - 1$ by Lemma 5.2 [28]. Else, we can reduce the LCE query in T to an LCE query in T' by Lemma 5.3 [28] and answer it using the same approach as in Theorem 2.5.2 (see line 8). The first mismatch between T_i and T_j occurs after at least $p_{i'+l} - i$ and at most $p_{i'+l} - i + 3\tau$ characters, hence we can find this position using an LCE query of length $\leqslant 3\tau$ on T (see line 9) analogously to line 1. All operations during the query procedure take $\mathcal{O}(1)$ time, hence the overall query time is $\mathcal{O}(1)$. □

Algorithm 4: $\mathsf{LCE}(i, j)$

```
1 lce_{⩽3τ} ← LCE_{⩽3τ}(i, j);
2 if lce_{⩽3τ} < 3τ then
3     return lce_{⩽3τ};
4 p_{i'} ← succ_S(i);
5 p_{j'} ← succ_S(j);
6 if p_{i'} − i ≠ p_{j'} − j then // Lemma 5.2 [28]
7     return min{p_{i'} − i, p_{j'} − j} + 2τ − 1;
8 l ← LCP'[RMQ_{LCP'}(SA_S^{-1}[i'] + 1, SA_S^{-1}[j'])];
9 return p_{i'+l} − i + LCE_{⩽3τ}(p_{i'+l}, p_{j'+l});
```

2.8.5 Definition. Let $\mathsf{LCP}_S[1..|S|]$ be an array, where $\mathsf{LCP}_S[1] = 0$ and $\mathsf{LCP}_S[i] = \mathsf{LCE}(\mathsf{SA}_S[i-1], \mathsf{SA}_S[i])$ for $i \in [2, |S|]$. We call LCP_S the *sparse longest common prefix* array of T w.r.t. S.

In practice, we compute LCP_S instead of LCP'. This eliminates the need for the $\mathsf{LCE}_{\leqslant 3\tau}$ query in line 9 of Algorithm 4, i.e, we return $p_{i'} - i + \mathsf{LCP}_S[\mathsf{RMQ}_{\mathsf{LCP}_S}(\mathsf{SA}_S^{-1}[i'] + 1, \mathsf{SA}_S^{-1}[j'])]$ in line 8.

2.8.6 Example. Let us continue Example 2.7.3. There, we computed the 3-synchronizing set $S = \{1, 4, 14, 15, 16, 19\}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b
$\mathsf{SA}_S =$	2	6	1	5	4	3																				
$\mathsf{SA}_S^{-1} =$	3	1	6	5	4	2																				
$\mathsf{LCP}_S =$	0	5	5	0	3	4																				

To compute $\mathsf{LCE}(i, j) = \mathsf{LCE}(8, 9)$, we compute $\mathsf{LCE}_{\leqslant 3\tau}(8, 9) = 9$. Since $9 \not< 3\tau = 9$, we compute $p_{i'} = \mathsf{succ}_S(8) = 14$ ($i' = 3$) and $p_{j'} = \mathsf{succ}_S(9) = 14$ ($j' = 3$). Since $p_{i'} - i = 6 \neq 5 = p_{j'} - j$, we return $\min\{p_{i'} - i, p_{j'} - j\} + 2\tau - 1 = \min\{6, 5\} + 2\tau - 1 = 10$.

With this text, it is not possible to reach line 8. However, we can skip lines 1-3 if $p_{i'} - i = p_{j'} - j$ holds. To compute $\mathsf{LCE}(i, j) = \mathsf{LCE}(14, 15)$, we would then compute $p_{i'} = \mathsf{succ}_S(14) = 14$ ($i' = 3$) and $p_{j'} = \mathsf{succ}_S(15) = 15$ ($j' = 4$). Since we have $p_{i'} - i = p_{j'} - j = 0$, we return $p_{i'} - i + \mathsf{LCP}_S[\mathsf{RMQ}_{\mathsf{LCP}_S}(\mathsf{SA}_S^{-1}[j'] + 1, \mathsf{SA}_S^{-1}[i'])] = 14 - 14 + \mathsf{LCP}_S[\mathsf{RMQ}_{\mathsf{LCP}_S}(5 + 1, 6)] = \mathsf{LCP}_S[6] = 4$ in line 8.

Chapter 3

Algorithms and Data Structures

By applying the concept of longest previous factors to the sparse setting, we get the *sparse longest previous factor* array $\mathsf{LPF}_S[1..|S|]$ w.r.t. a subset (sampling) S of text positions from T, which is defined as follows.

3.0.1 Definition. We call the array $\mathsf{PSV}_S[1..|S|]$ the *sparse previous smaller value* (PSV) array w.r.t. S, where $\mathsf{PSV}_S[i] = 0$ if $\forall j \in [1, i) : \mathsf{SA}_S[j] > \mathsf{SA}_S[i]$ and $\mathsf{PSV}_S[i] = \max\{j \in [1, i) \mid \mathsf{SA}_S[j] < \mathsf{SA}[i]\}$, else. Analogously, we call the array $\mathsf{NSV}_S[1..|S|]$ the *sparse next smaller value* (NSV) array w.r.t. S, where $\mathsf{NSV}_S[i] = 0$ if $\forall j \in (i, |S|] : \mathsf{SA}_S[j] > \mathsf{SA}_S[i]$ and $\mathsf{NSV}_S[i] = \min\{j \in (i, |S|] \mid \mathsf{SA}_S[j] < \mathsf{SA}_S[i]\}$, else. We call an array $\mathsf{LPF}_S[1..|S|]$ a *sparse longest previous factor* array w.r.t. S iff for each $i \in [1, |S|]$, either $\mathsf{LPF}_S[i] = \perp$ if $T[S[i]]$ does not occur in $T[1, S[i])$, or else, $\mathsf{LPF}_S[i] = \arg\max_{j \in [1, S[i])}\{\mathsf{LCE}(j, S[i])\}$.

3.1 LZ77 Approximation Algorithms

The following LZ77 Algorithms apply the LPF algorithm to the sparse setting by implicitly computing the sparse LPF array w.r.t. a string synchronizing set S and using the $\mathcal{O}(1)$ time LCE data structure from Theorem 2.8.4, which also uses S. Additionally, they use the following data structures to close gaps in the factorization.

3.1.1 Auxiliary Data Structures

3.1.1 Lemma (Lemma 3 [12]). *Let $\epsilon \in \mathbb{R}^+$ be a constant. There is a data structure that, given a pattern $P \in \Sigma^m$ with $m \leqslant \log_2 n/((2+\epsilon)\lceil\log_2 \sigma\rceil)$, outputs the leftmost occurrence of P in T in $\mathcal{O}(1)$ time. It can be computed in $\mathcal{O}(n/\log_\sigma n)$ time and $o(n/\log_\sigma n)$ space.*

Proof. Let $m' = \lfloor \log_2 n/((2+\epsilon)(\lceil\log_2 \sigma\rceil))\rfloor$ be the maximum allowed pattern length, let $P' \in \Sigma^{2m'}$, and let $\hat{\epsilon} = \epsilon/(2+\epsilon)$. Since $2^{2m'\lceil\log_2 \sigma\rceil} \leqslant 2^{2\log_2 n/(2+\epsilon)} = n^{2/(2+\epsilon)} = n^{1-(\epsilon/(2+\epsilon))} = n^{1-\hat{\epsilon}}$, we have $\mathsf{int}(P') \in [0, n^{1-\hat{\epsilon}})$. Our data structure then stores for each allowed pattern

length $m \in [1, m']$ an array $L_m[1..2^{m\lceil \log_2 \sigma \rceil}]$, where $L_m[\mathsf{int}(P)]$ stores the leftmost occurrence of P in T for each possible P.

To construct $L_1, ..., L_{m'}$, we at first compute the leftmost m'-aligned occurrence of each possible P' in T, that is, we compute $A[1..2^{2m'\lceil \log_2 \sigma \rceil}]$, where $A[\mathsf{int}(P')]$ either stores $im'+1$ if there exists an $i \in [0, n/m - 1]$ such that $T[im' + 1, (i + 2)m'] = P'$, or $n + 1$, else. This can be done by initializing $A \leftarrow [n + 1, ..., n + 1]$ (see line 1 in Algorithm 5), and for each possible P' for i from $n/m' - 2$ down to 0, setting $A[\mathsf{int}(T[im' + 1, (i + 2)m'])] \leftarrow im' + 1$ (see lines 2-3). This takes $\mathcal{O}(n/m') = \mathcal{O}(n/\log_\sigma n)$ time.

Algorithm 5: build-L(m')

```
1  A[1..2^{2m'⌈log_2 σ⌉}] ← [n+1, ..., n+1];                    // O(n^{1−ε̂}) time
2  for i from n/m' − 2 down to 0 do                              // O(n/log_σ n) time
3      A[int(T[im'+1, (i+2)m'])] ← im'+1;
4  for m from 1 up to m' do                                      // O(√n log_σ n) time
5      L_m[1..2^{m⌈log_2 σ⌉}] ← [n+1, ..., n+1];                 // O(√n) time
6  for int(P') from 1 up to 2^{2m'⌈log_2 σ⌉} do                  // O(n^{1−ε̂} log_σ^2 n) time
7      for j from 1 up to 2m' − m + 1 do                         // O(m^2) = O(log_σ^2 n) time
8          let int(P) = int(P'[j, j+m));
9          L_m[int(P)] ← min(L_m[int(P)], A[int(P')] + j);
10 return L_1, L_2, ..., L_{m'};
```

With A, we can now build $L_1, L_2, ..., L_{m'}$. We initialize $L_m = [n + 1, ..., n + 1]$ for each $m \in [1, m']$ (see lines 4-5). Note that the leftmost occurrence of each P is contained in the leftmost m'-aligned occurrence of at least one P'. To construct L_m, we therefore consider for every possible P', each $j \in [1, 2m' - m)$ with $P = P'[j, j + m)$, and set $L_m[\mathsf{int}(P)] \leftarrow \min(L_m[\mathsf{int}(P)], A[\mathsf{int}(P')] + j)$ (see lines 6-9). Since there are $n^{1-\hat{\epsilon}}$ choices of P' and $\mathcal{O}(m^2) = \mathcal{O}(\log_\sigma^2 n)$ choices of P, this takes $\mathcal{O}(n^{1-\hat{\epsilon}} \log_\sigma^2 n) \subset o(n/\log_\sigma n)$ time. Since the construction of A takes $\mathcal{O}(n/\log_\sigma n)$ time, we get an overall running time of $\mathcal{O}(n/\log_\sigma n)$.

Regarding the total used space, note that A and each L_m has size $\mathcal{O}(n^{1-\hat{\epsilon}})$, and $m' = \mathcal{O}(\log_\sigma n)$. Therefore, we need $\mathcal{O}(n^{1-\hat{\epsilon}} \log_\sigma n) \subset o(n/\log_\sigma n)$ space in total. □

3.1.2 Lemma (Lemma 2 [12]). *There is a data structure that, given a pattern $P \in \Sigma^m$ with $m \leqslant \log_2 n/2\lceil \log_2 \sigma \rceil$, outputs the period of P in $\mathcal{O}(1)$ time. It can be computed in $\mathcal{O}(\sqrt{n} \log_\sigma^3 n)$ time and $\mathcal{O}(\sqrt{n} \log_\sigma n)$ space.*

Proof. Let $m' = \lfloor \log_2 n/2\lceil \log_2 \sigma \rceil \rfloor$ be the maximum allowed pattern length. The data structure then stores for each allowed pattern length $m \in [1, m']$ an array $Q_m[1..2^{m\lceil \log_2 \sigma \rceil}]$, where $Q_m[\mathsf{int}(P)]$ stores $\mathsf{per}(P)$, for each P. Since each array Q_m has size $2^{m\lceil \log_2 \sigma \rceil} \leqslant 2^{\log_2 n/2} =$

$\mathcal{O}(\sqrt{n})$, and there are $m' = \mathcal{O}(\log_\sigma n)$ arrays, this data structure needs $\mathcal{O}(\sqrt{n}\log_\sigma n)$ space. For each P, we compute $\mathsf{per}(P)$ naively in $\mathcal{O}(m^2) = \mathcal{O}(\log_\sigma^2 n)$ time, hence the construction takes $\mathcal{O}(\sqrt{n}\log_\sigma^3 n)$ time. □

3.1.2 LZ77 3-Approximation

At first, we discuss the LZ77 3-Approximation from [12], which we have already roughly described in Section 1.2. It sets $\tau = \lfloor \log_2 n/(8\lceil \log_2 \sigma \rceil)\rfloor$. At first, it computes a gapped factorization of $z' \leqslant z$ phrases starting at and referencing sampled positions such that gaps that are longer than 3τ are periodic and can thus be shortened to length $\leqslant 3\tau$ or closed entirely efficiently in the second phase. Then, it closes the remaining short gaps in the third phase.

3.1.3 Definition. We call the factorization $f_1 g_1 r_1 f_2 g_2 r_2 ... f_{z'} g_{z'} r_{z'} = T$ a gapped LZ factorization of T iff for each $x \in [1, z']$,

- the *perfect phrase* f_x with destination $i = 1 + \sum_{j=1}^{x-1} |f_j g_j h_j|$ is either literal (the leftmost occurrence of a single character, $|f_x| = 1$), or the longest prefix of T_i that occurs before position i in T at some source $k \in [1, i)$,
- and the reference r_x with destination $i = 1 + |f_x g_x| + \sum_{j=1}^{x-1} |f_j g_j h_j|$ is either empty ($r_x = \epsilon$), or a prefix of T_i (not necessarily the longest prefix) that occurs before position i in T at some source $k \in [1, i)$.

For $x \in [1, z']$, we call g_x a (possibly empty) gap.

3.1.4 Lemma (Lemma 5 [12]). *Any gapped LZ factorization $f_1 g_1 r_1 f_2 g_2 r_2 ... f_{z'} g_{z'} r_{z'} = T$ satisfies $z' \leqslant z$, where z is the number of factors in the exact LZ77 factorization.*

Proof. Let j be the destination of some perfect phrase f_x in the gapped factorization, and let i be the destination of the phrase f_y of the exact LZ77 factorization that contains j. Since f_y occurs before position i in T, $T[j, i + |f_y|] = f_y[j - i + 1, |f_y|]$ occurs before position j in T. Since f_x is perfect (of maximal length), it has length $\geqslant |f_y| - j + i$, hence it ends either at or after the end of f_y in T. Therefore, each phrase of the exact LZ77 factorization contains the destination of at most one perfect phrase of the gapped LZ factorization, hence $z' \leqslant z$. □

3.1.2.1 Phase 1

3.1.5 Lemma. *Given a $\tau = \lfloor \log_2 n/(8\lceil \log_2 \sigma \rceil)\rfloor$-synchronizing set S, we can compute in $\mathcal{O}(n/\log_\sigma n)$ time and space a gapped LZ factorization $f_1 g_1 r_1 f_2 g_2 r_2 ... f_{z'} g_{z'} r_{z'} = T$ such that all referencing phrases are empty, and no gap contains a position in S.*

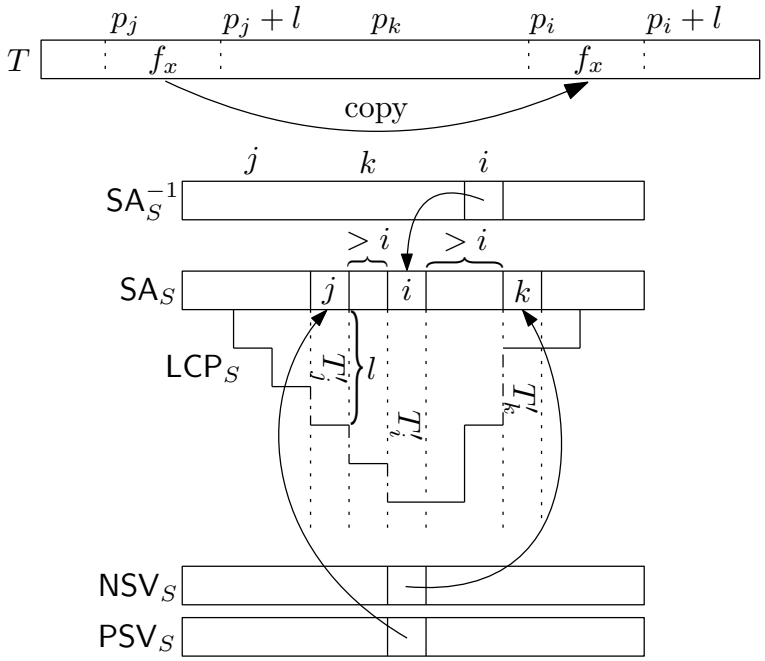


Figure 3.1: Illustration of the computation of $\mathsf{LPF}_S[i]$ in $\mathcal{O}(1)$ time.

Proof. We process S from left to right. Let $i \leftarrow 1$ be the index of the current sample in S, and let $x \leftarrow 1$ be the index of the current phrase in the gapped LZ factorization.

For each i-th sample, we check whether $T[S[i], S[i]+2\tau)$ occurs before position $S[i]$ in T by querying $p = L_{2\tau}[\mathsf{int}(T[S[i], S[i]+2\tau))]$ using Lemma 3.1.1. We can do this, because $2\tau = 2\lfloor \log_2 n/(8\lceil \log_2 \sigma \rceil)\rfloor \leqslant \log_2 n/(4\lceil \log_2 \sigma \rceil) \leqslant \log_2 n/((2+\epsilon)\lceil \log_2 \sigma \rceil)$ for $\epsilon \in (0, 2]$.

Case 1. $p < S[i]$: Then, $T[S[i], S[i] + 2\tau)$ has a previous occurrence, and we know by the synchronizing condition (see Definition 2.7.1) that (**i**) all previous occurrences of $T[S[i], S[i] + 2\tau)$ start at positions in S, hence we can compute $\mathsf{LPF}_S[i]$ in the following way (see Figure 3.1). If $\mathsf{PSV}_S[\mathsf{SA}_S^{-1}[i]] \neq 0$, then let $p_j = S[j] = S[\mathsf{SA}_S[\mathsf{PSV}_S[\mathsf{SA}_S^{-1}[i]]]]$. Similarly, if $\mathsf{NSV}_S[\mathsf{SA}_S^{-1}[i]] \neq 0$, then let $p_k = S[k] = S[\mathsf{SA}_S[\mathsf{NSV}_S[\mathsf{SA}_S^{-1}[i]]]]$. By (**i**), at least one of p_j and p_k must be defined. If either p_j or p_k is undefined, then $\mathsf{LPF}_S[i] = p_k$ or $\mathsf{LPF}_S[i] = p_j$, respectively. Else (both p_j and p_k are defined), then $\mathsf{LPF}[i] = p_j$ if $\mathsf{LCE}(p_j, p_i) > \mathsf{LCE}(p_k, p_i)$, and $\mathsf{LPF}_S[i] = p_k$, else. Then, we choose $f_x = T[S[i], S[i] + \mathsf{LCE}(S[i], \mathsf{LPF}_S[i]))$ with source $\mathsf{LPF}_S[i]$ as the next referencing perfect phrase.

Case 2. $p = S[i]$: Then, $T[S[i], S[i]+2\tau)$ has no previous occurrence. Now, we find the maximum l such that $T[S[i], S[i]+l)$ has a previous occurrence by applying Lemma 3.1.1 for each $l \in [1, 2\tau)$. Let $l' \in [1, 2\tau)$ be maximal such that $T[S[i], S[i] + l')$ has a previous occurrence $p' = L_{l'}[\mathsf{int}(T[S[i], S[i] + l'))] < S[i]$. If $l' = 0$, then we choose $f_x = T[S[i]]$ as a literal perfect phrase. Else, we choose $f_x = T[S[i], S[i] + l)$ as a referencing perfect phrase with source p'.

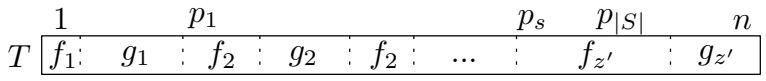


Figure 3.2: Illustration of the LZ-like factorization after Phase 1 of the LZ77 3-Approximation (see Lemma 3.1.5).

Now, we skip the samples in S before the end e of the just computed perfect phrase, that is, we increment i until $i > |S|$ or $S[i] > e$. If $i > |S|$, then we are done. Else, we increment x and repeat the procedure until $i > |S|$ holds. Since each next phrase starts at the respective next uncovered sample in S and has length at least 1, this respective next sample and therefore all samples in S will be covered by some perfect phrase (see Figure 3.2). By the construction, each phrase is perfect (of maximal length), hence the computed factorization is a gapped LZ factorization without referencing phrases.

Theorem 2.8.4 and Lemma 3.1.1 need $\mathcal{O}(n/\log_\sigma n)$ time and space. Since there are $\leqslant |S|$ samples in S, we apply Case 1 and 2 $\leqslant |S|$ times. One application of Case 1 takes $\mathcal{O}(1)$ time, hence Case 1 needs overall $\mathcal{O}(|S|) = \mathcal{O}(n/\tau) = \mathcal{O}(n/\log_\sigma n)$ time. One application of Case 2 takes $\mathcal{O}(\tau)$ time, but since we encounter Case 2 at most once per distinct length-2τ substring of T, of which there are at most $2^{2\tau\lceil\log_2\sigma\rceil}2\tau$, Case 2 needs overall $\mathcal{O}(2^{2\tau\lceil\log_2\sigma\rceil}\tau^2) = \mathcal{O}(n^{1/4}\log_\sigma^2 n) \subset \mathcal{O}(n/\log_\sigma n)$ time. Therefore, we need $\mathcal{O}(n/\log_\sigma n)$ time and space in total. □

3.1.2.2 Phase 2

3.1.6 Lemma. *Given a $\tau = \lfloor\log_2 n/(8\lceil\log_2\sigma\rceil)\rfloor$-synchronizing set S and a gapped LZ factorization $f_1g_1r_1f_2g_2r_2...f_{z'}g_{z'}r_{z'} = T$, where all referencing phrases are empty, and no gap contains a position in S, we can compute in $\mathcal{O}(n/\log_\sigma n)$ time and space a gapped LZ factorization $f'_1g'_1r'_1f'_2g'_2r'_2...f'_{z'}g'_{z'}r'_{z'} = T$ such that the length of each gap is at most 3τ, i.e, $\forall x \in [1, z'] : |g'_x| \leqslant 3\tau$.*

Proof. By the density condition (see Definition 2.7.1), each long gap g with $|g| > 3\tau$ has period $\leqslant \frac{\tau}{3}$. Thus, we can eliminate all long gaps by replacing for each long gap g with destination i, the phrases gr with $g'r'$, where r is an empty referencing phrase (by the assumption), $g' = g[1, 3\tau]$ is a new short gap and $r' = g(3\tau, |g|]$ is a new referencing phrase with source $i + 3\tau - \mathsf{per}(T[i, i + 3\tau))$ (see Figure 3.3). To compute the period p of $T[i, i + 3\tau)$, we can use Lemma 3.1.2, which works, because $3\tau = 3\lfloor\log_2 n/(8\lceil\log_2\sigma\rceil)\rfloor \leqslant \log_2 n/(2\lceil\log_2\sigma\rceil)$.

Lemma 3.1.2 needs $\mathcal{O}(\sqrt{n}\log_\sigma^3 n) \subset \mathcal{O}(n/\log_\sigma n)$ time and space. Since there are $z' \leqslant z = \mathcal{O}(n/\log_\sigma n)$ gaps, we can identify each long gap in $\mathcal{O}(n/\log_\sigma n)$ time. Then, we spend additional $\mathcal{O}(1)$ time to replace each long gap, hence we need overall $\mathcal{O}(n/\log_\sigma n)$ time and space. □

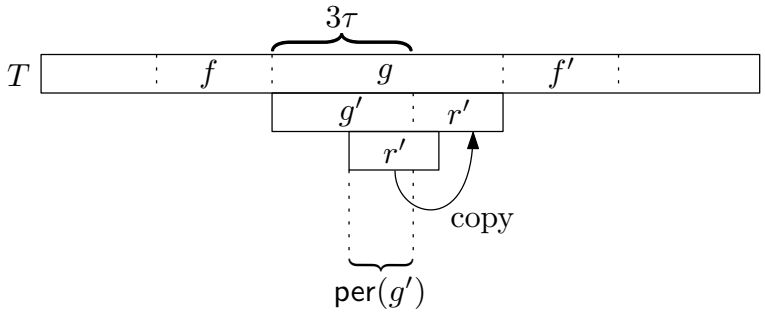


Figure 3.3: Illustration of the shortening of a long gap in Phase 2 of the LZ77 3-Approximation (see Lemma 3.1.6).

3.1.2.3 Phase 3

3.1.7 Lemma. *Given a* $\tau = \lfloor \log_2 n/(8\lceil \log_2 \sigma \rceil)\rfloor$*-synchronizing set* S *and a gapped LZ factorization* $f_1 g_1 r_1 f_2 g_2 r_2 ... f_{z'} g_{z'} r_{z'} = T$*, where the length of each gap is at most* 3τ*, i.e,* $\forall x \in [1, z'] : |g_x| \leqslant 3\tau$*, we can compute in* $\mathcal{O}(n/\log_\sigma n)$ *time and space an LZ-like factorization of* T *with* $\leqslant 3z$ *phrases, where* z *is the number of factors in the exact LZ77 factorization.*

Proof. We call a gap referencing if it has a previous occurrence. At first, we identify all non-referencing non-empty gaps (step 1). Then, we iteratively replace those until all remaining gaps are referencing (step 2). Finally, we replace the referencing gaps with references (step 3).

Step 1. Since all gaps have length $\leqslant 3\tau$, we can classify all gaps using Lemma 3.1.1. Let g be a gap with destination i. If $L_{|g|}[\mathsf{int}(T[i, i + |g|))] = i$, then g is non-referencing.

Step 2. Let g be a non-referencing gap. We find the maximum $l \in [0, |g|]$ such that $T[i, i + l)$ has a previous occurrence $p = L_l[\mathsf{int}(T[i, i + l))]$ (if $l > 0$) using Lemma 3.1.1 and replace g with $g' r f g''$, where g' is a new empty gap, r is a new empty reference, $f = T[i, i+\max(1, l))$ is a new perfect phrase (with source p if $l > 0$) and $g'' = g[l+\max(1, l), |g|]$ is a new gap (see Figure 3.4). Finally, we classify g'' and repeat Step 2 for g'' if it is non-referencing.

Step 3. Since the current factorization still is a gapped LZ factorization, there remain $z'' \in [z', z]$ perfect phrases and $\leqslant z' \leqslant z$ non-empty referencing gaps and non-empty references, each. Now, we discard all empty factors and use Lemma 3.1.1 to replace each non-empty non-referencing gap g at destination i with a reference with source $L_{|g|}[\mathsf{int}(T[i, i + |g|))]$. Therefore, this results in an LZ-like factorization with $\leqslant z'' + 2z' \leqslant 3z' \leqslant 3z$ phrases.

Theorem 2.8.4 and Lemma 3.1.1 need $\mathcal{O}(n/\log_\sigma n)$ time and space, and since we query it with pattern length $\leqslant 3\tau = 3\lfloor \log_2 n/(8\lceil \log_2 \sigma \rceil)\rfloor \leqslant \log_2 n/((2 + \epsilon)\lceil \log_2 \sigma \rceil)$ for $\epsilon \in (0, \frac{2}{3}]$, we do not exceed the maximum allowed pattern length. Since there are $\leqslant z'$ gaps in

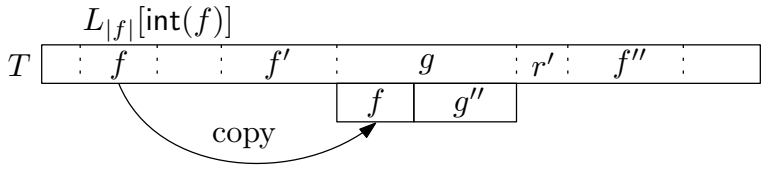


Figure 3.4: Illustration of the shortening of a short gap in Phase 3 of the LZ77 3-Approximation (see Lemma 3.1.7).

the initial factorization, and we spend $\mathcal{O}(1)$ time to classify one gap, step 1 takes overall $\mathcal{O}(z') = \mathcal{O}(z) = \mathcal{O}(n/\log_\sigma n)$ time. Similarly, step 3 needs $\mathcal{O}(1)$ time per referencing gap and therefore $\mathcal{O}(z') = \mathcal{O}(n/\log_\sigma n)$ time overall.

It now remains to limit the time spent during step 2. Since there are $2^{3\tau\lceil\log_2\sigma\rceil}3\tau = \mathcal{O}(n^{3/8}\tau)$ distinct length-3τ substrings in T, and each non-referencing gap g is a leftmost occurrence of a length-$\leqslant 3\tau$ substring of T, we process $\mathcal{O}(n^{3/8}\tau)$ non-referencing gaps overall. Furthermore, we spend $\mathcal{O}(\tau)$ time per non-referencing gap, hence step 3 needs overall $\mathcal{O}(n^{3/8}\tau^2) = \mathcal{O}(n^{3/8}\log_\sigma^2 n) \subset \mathcal{O}(n/\log_\sigma n)$ time. □

Theorem 3.1.8 follows directly from the chained application of Lemma 3.1.5, Lemma 3.1.6 and Lemma 3.1.7.

3.1.8 Theorem (Theorem 1 [12]). *We can compute in $\mathcal{O}(n/\log_\sigma n)$ time and space an LZ-like factorization with at most $3z$ factors, where z is the number of factors in the exact LZ77 factorization.*

3.1.2.4 Computing the Factorization From Left To Right

The LZ77 3-Approximation has been split into multiple phases such that their respective running times can be analyzed more easily. However, the resulting factorization can also be computed from left to right (see Algorithm 8). We start by discussing two auxiliary algorithms that are needed in the main algorithm. Then, we describe the main algorithm.

3.1.2.4.1 Auxiliary Algorithms $\mathsf{prev\text{-}occ}(i, l_{\mathsf{max}})$ (see Algorithm 7) returns $\langle T[b], 0\rangle$ if $T[i]$ has no previous occurrence or $\langle s, l\rangle$, where l is the maximum length $\leqslant l_{\mathsf{max}}$ such that the first occurrence s of $T[i, i+l)$ lies before i. $\mathsf{prev\text{-}occ}(i, l_{\mathsf{max}})$ runs in $\mathcal{O}(\max(1, l_{\mathsf{max}} - l))$ time.

Let $i \in [1, |S|]$. If $T[S[i], S[i]+2\tau)$ has a previous occurrence, then $\mathsf{longest\text{-}prev\text{-}factor}(i)$ (see Algorithm 6) returns $\langle \mathsf{LPF}[i], p\rangle$, where p is a previous occurrence of $T[S[i], S[i] + \mathsf{LPF}[i])$. It runs in $\mathcal{O}(1)$ time and is correct due to the same argument as in Lemma 3.1.5.

3.1.2.4.2 Main Algorithm Here, we maintain the position $i \in [1, |S|]$ of the current sample in S and the start b and end e of the current gap $[b, e) \subseteq [1, n]$. We run through

Algorithm 6: longest-prev-factor(i)

```
1  ⟨s, l⟩ ← ⟨⊥, 0⟩;
2  if PSV_S[SA_S^{-1}[i]] ≠ ⊥ then
3  |   s' ← S[SA_S[PSV_S[SA_S^{-1}[i]]]];
4  |   l' ← LCE(s', S[i]);
5  |   if l' > l then
6  |   |   ⟨s, l⟩ ← ⟨s', l'⟩;
7  if NSV_S[SA_S^{-1}[i]] ≠ ⊥ then
8  |   s' ← S[SA_S[NSV_S[SA_S^{-1}[i]]]];
9  |   l' ← LCE(s', S[i]);
10 |   if l' > l then
11 |   |   ⟨s, l⟩ ← ⟨s', l'⟩;
12 return ⟨s, l⟩;
```

Algorithm 7: prev-occ(i, l_{max})

```
1  l ← l_max;
2  while l > 0 do
3  |   s ← L_l[int(T[i, i + l])];
4  |   if s < i then
5  |   |   return ⟨s, l⟩;
6  |   l ← l − 1;
7  return ⟨T[i], 0⟩;
```

$z' + 1$ iterations in the main while-loop. Each iteration can be divided into two steps, where we refer to the perfect phrases in the initial gapped LZ factorization resulting from Lemma 3.1.5.

Step 1. In lines 4-12 of the j-th iteration (see Figure 3.5), we close the gap $[b, e)$ between the end b of the perfect phrase f_{j-1} in T (or 1 if $j = 1$) and the start e of the next perfect phrase f_j in T (or $n + 1$ if $j = z' + 1$). Overall, we apply Lemma 3.1.7 and Lemma 3.1.6 to the gap if it is not empty. We start by setting $e' \leftarrow \min(b + 3\tau, e)$. If $[b, e)$ is a long gap (see Lemma 3.1.6), then $e' < e$, and we compute the period p of $T[b, e')$ in line 6 using Lemma 3.1.2. If the remaining short gap $[b, e')$ is not empty, we iteratively shorten it from the left in the while loop in lines 7-9. In each iteration, we call prev-occ$(b, e' - b)$ (see Algorithm 7), output the factor it returns, and increment b by its length. Let k be the number of iterations this while loop runs through.

In each (except for possibly the last) iteration, we output perfect factors to shorten non-referencing gaps. This takes $\mathcal{O}(\tau)$ time per gap as required in the proof of Lemma 3.1.7. In the last iteration (which may also be the first iteration), we may close a referencing gap. If so, then this takes $\mathcal{O}(1)$ time, because prev-occ$(b, e' - b)$ returns a factor with length $e' - b$, and this takes $\mathcal{O}(\max(1, l_{\mathsf{max}} - l)) = \mathcal{O}(\max(1, (e' - b) - (e' - b))) = \mathcal{O}(1)$ time as argued in Paragraph 3.1.2.4.1.

Finally, we potentially output a referencing factor (line 11) to fully close the long gap $[b, e)$ with a reference $[e', e)$ and source $e' - p = e' - \mathsf{per}(T[b, e'))$ as described in the proof of Lemma 3.1.6.

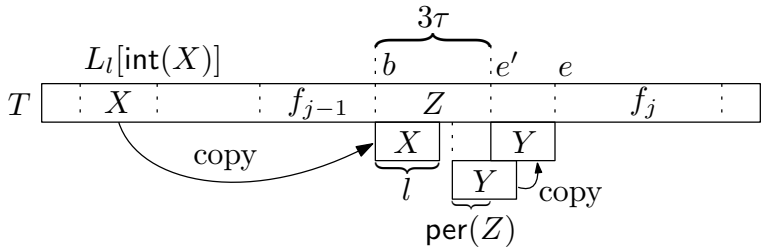


Figure 3.5: Illustration of the j-th iteration of the LZ77 3-Approximation (see Algorithm 8).

Step 2. In lines 13-25 of the j-th iteration, we compute the perfect phrase f_j if $j \leqslant z'$, or break out of the main while loop in line 14, else ($b = n + 1$, see lines 13-14). If $j \leqslant z'$, then we consider Cases 1 and 2 as in the proof of Lemma 3.1.5.

Case 1. $T[S[i], S[i]+2\tau)$ has no previous occurrence: Then, we can output the perfect factor returned by $\mathsf{prev\text{-}occ}(b, 2\tau - 1)$ in line 16. This takes $\mathcal{O}(\tau)$ time as required in the proof of Lemma 3.1.5.

Case 2. $T[S[i], S[i] + 2\tau)$ has a previous occurrence: By the same argument as in Paragraph 3.1.2.4.1, we can output $\mathsf{longest\text{-}prev\text{-}factor}(i)$ in line 18. This takes $\mathcal{O}(1)$ time as required in the proof of Lemma 3.1.5.

Finally, we try to find the next sample in S starting after the end b of the perfect phrase f_j by incrementing i (see lines 20-21). If there is no next sample, then the next gap ends at the end of T, i.e, at $n + 1$ (see line 23). Else, we can set $e \leftarrow S[i]$ in line 25.

3.1.9 Example. In Example 2.7.3, we computed the 3-synchronizing set $S = \{1, 4, 14, 15, 16, 19\}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b
$\mathsf{SA}_S =$	2	6	1	5	4	3																				
$\mathsf{SA}_S^{-1} =$	3	1	6	5	4	2																				
$\mathsf{PSV}_S =$	0	1	0	3	3	3																				
$\mathsf{NSV}_S =$	3	3	0	5	6	0																				

In the first iteration, we consider the first sample $S[i] = S[1] = 1$. We have $b, e, e' = 1$. Since $T[1, 2\tau) = abcabc$ has no previous occurrence, we call $\mathsf{prev\text{-}occ}(1, 5)$ in line 19 and output $\langle 0, T[1]\rangle$ as a literal factor. Then, we set $b \leftarrow b + \max(1, 1) = 2$, increment i until $i = 2$, because then, $S[i] = S[2] = 4 \geqslant b = 2$. Finally, we set $e = S[i] = 4$. In the second iteration, we set $e' = 4$ and output 2 literal factors $\langle 0, b\rangle$ and $\langle 0, c\rangle$ in line 9. Now, we have $b, e = 4$. Since $T[b, b + 2\tau) = abcabb$ has no previous occurrence, we call $\mathsf{prev\text{-}occ}(4, 5)$ in line 19 and output $\langle 1, 5\rangle$ as a referencing factor. We set $b = 9$, increment i until $i = 3$ and set $e = S[3] = 14$. In the third iteration, we set $e' = 14$ and output

Algorithm 8: lz77-3-approximation()

```
1  i ← 1;                                          // index of the current sample in S
2  b ← 1; e ← S[1];                                // beginning and end of the current gap
3  while true do
4      e' ← min(b + 3τ, e);                        // end of the current short gap
5      if e' < e then                              // the gap is long
6          p ← Q_{3τ}[int(T[b, e'))];              // period of T[b, b + 3τ)
7      while b < e' do                             // close the short gap
8          ⟨s, l⟩ ← prev-occ(b, e' − b);
9          output ⟨s, l⟩;
10         b ← b + max(1, l);
11     if e' < e then                              // the gap is long
12         output ⟨e' − p, e − e'⟩;                // output a referencing factor
13         b ← e;                                  // close the long gap
14     if b = n + 1 then
15         break;
16     if L_{2τ}[int(T[b, b + 2τ))] = b then       // T[b, b + 2τ) has no prev. occ.
17         output prev-occ(b, 2τ − 1);             // output a lit./ref. perfect factor
18     else                                        // T[b, b + 2τ) has a previous occurrence
19         output longest-prev-factor(i);          // output a referencing perfect factor
20     b ← b + max(1, l);                          // the new gap starts after the perfect phrase
21     while i ⩽ |S| ∧ b > S[i] do                 // find the next sampled position
22         i ← i + 1;
23     if i > |S| then                             // there is no next sample
24         e ← n + 1;                              // the new gap ends at the end of T
25     else
26         e ← S[i];                               // the new gap ends at the next sample
```

$\langle 8, 5\rangle$ in line 9. Now, we have $b = 14$. Since $T[b, b + 2\tau) = bbbbba$ has no previous occurrence, we output $\langle 8, 5\rangle$ in line 17, set $b = 19$ and increment i until $i = 6$. In the last iteration, we have $b, e, e' = 19$. Since $T[b, b + 2\tau) = abcabc$ has a previous occurrence, we call longest-prev-factor(6). There, we consider the lexicographically next smaller and larger sampled suffixes $S[\mathsf{SA}_S[\mathsf{PSV}_S[\mathsf{SA}_S^{-1}[6]]]] = 4$ and $S[\mathsf{SA}_S[\mathsf{NSV}_S[\mathsf{SA}_S^{-1}[6]]]] = 1$, respectively. Since T_1 yields a longer match (length 8) with $T_b = T_{19}$ than T_4, we output $\langle 1, 8\rangle$ as the final factor.

We have $z' = 7$. It holds $z = 6$, because in the exact factorization, we have $\langle 8, 10\rangle$ as the third factor with destination 9. Thus, the approximation ratio is $z'/z = 7/6 = 1.1\overline{6}$.

3.2 LZ77 Exact Algorithm

We compute the exact LZ77 factorization incrementally from left to right. At first, we construct a special sampling of text positions from an LZ-like factorization, and lexicographically and co-lexicographically sort the samples. The task of finding the next phrase starting at position i can then be reduced to (**i**) computing pairs of sparse prefix- and sparse suffix array intervals, (**ii**) intersecting the samples contained in those, and (**iii**) filtering out samples starting at or after position i.

3.2.1 Preliminaries

We at first define the problem of *insertion-only orthogonal range one-reporting*, which we will use for (**ii** and (**iii**)). Then, we discuss sparse prefix- and suffix sorting, and how to compute the sparse prefix- and suffix array interval of a given pattern w.r.t. a sample set (see (**i**)). Finally, we define and show how to compute the special sample set.

3.2.1 Definition. Let π be a permutation of $[1, N]$, and let $\mathcal{U} = \{(i, \pi(i)) \mid i \in [1, N]\}$ be the set of valid points. The problem of *insertion-only orthogonal range one-reporting* (IO-OROR) is to maintain an initially empty set of points $\mathcal{P} \subseteq \mathcal{U}$ and support the following operations.

- Insert a point $p \in \mathcal{U} \backslash \mathcal{P}$ into $\mathcal{P}$, and
- Given a query rectangle $\mathcal{Q} = [x_1, x_2] \times [y_1, y_2]$, output a point $p \in \mathcal{Q} \cap \mathcal{P}$, or report $\mathcal{Q} \cap \mathcal{P} = \varnothing$.

If the sequence $I = (i_1, \pi(i_1)), ..., (i_M, \pi(i_M))$ of inserted points is known beforehand, i.e, $(i_j, \pi(i_j))$ is inserted after $(i_k, \pi(i_k))$, for all $j \in [1, M]$ and $k \in [1, j)$, then we call $\langle \pi, I\rangle$ an instance of the IO-OROR problem with fixed insertion order.

3.2.2 Definition. We define the unique permutation $\mathsf{PA}_S[1..|S|]$ of $[1, |S|]$ satisfying $\forall i, j \in [1, |S|] : \mathsf{PA}_S[i] < \mathsf{PA}_S[j] \Leftrightarrow T[1, S[i]] \prec T[1, S[j]]$ to be the *sparse prefix array* of T w.r.t.

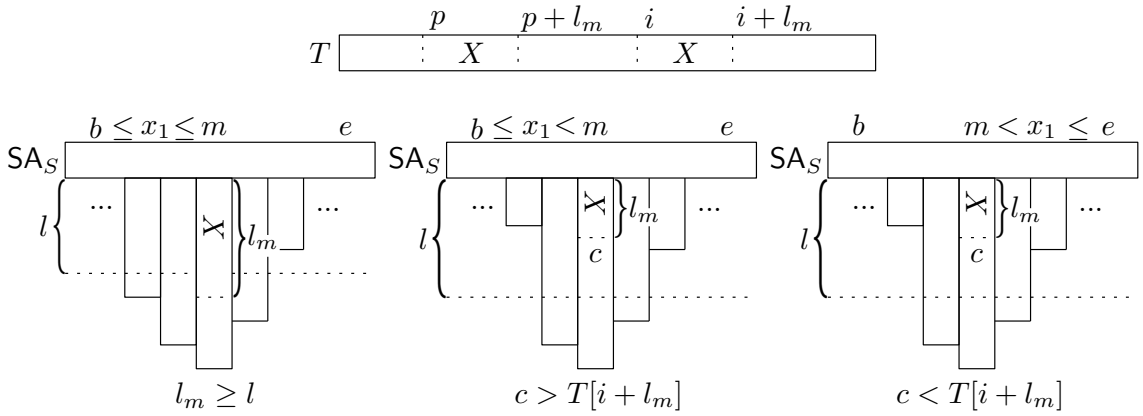


Figure 3.6: Illustration of the possible cases during the computation of x_1 in Lemma 3.2.5.

S. Let the unique permutation $\mathsf{PA}_S^{-1}[1..|S|]$ of $[1, |S|]$ with $\forall i \in [1, |S|] : \mathsf{PA}_S[\mathsf{PA}_S^{-1}[i]] = i$ be the *sparse inverse prefix array* of T w.r.t. S. For a substring $T[i, i + l)$, the *sparse prefix array interval* $\mathsf{piv}_S(T[i, i + l))$ w.r.t. S is the maximum interval $[x_1, x_2] \subseteq [1, |S|]$ such that $\forall v \in \mathsf{PA}_S[x_1, x_2] : T(S[v] - l, S[v]] = T[i, i + l)$. The *sparse suffix array interval* $\mathsf{siv}_S(T[i, i + l))$ w.r.t. S is defined analogously as the maximum interval $[y_1, y_2] \subseteq [1, |S|]$ such that $\forall v \in \mathsf{SA}_S[y_1, y_2] : T[S[v], S[v] + l) = T[i, i + l)$.

3.2.3 Observation. Let $\mathcal{P} = \{(\mathsf{PA}_S^{-1}[i], \mathsf{SA}_S^{-1}[i]) \mid i \in [1, |S|]\}$ and $\mathcal{I} = (\mathsf{PA}_S^{-1}[1], \mathsf{SA}_S^{-1}[1]), ..., (\mathsf{PA}_S^{-1}[|S|], \mathsf{SA}_S^{-1}[|S|])$. Then, $\langle \mathcal{P}, \mathcal{I} \rangle$ is an instance of the IO-OROR problem with fixed insertion order.

3.2.4 Lemma. *We can compute* $\mathsf{PA}_S[1..|S|]$ *and* $\mathsf{SA}_S[1..|S|]$ *in* $\mathcal{O}(n/\log_\sigma n + |S| \log |S|)$ *time and* $\mathcal{O}(n/\log_\sigma n + N)$ *space.*

Proof. We build the data structure for $\mathcal{O}(1)$ time LCE queries in $\mathcal{O}(n/\log_\sigma n)$ space from Theorem 2.8.4 for T. Then, we initialize an array $A[1..|S|] = [1, .., |S|]$. Now, we use comparison-based sorting to sort A such that $A = \mathsf{SA}_S$. Let $i, j \in [1, |S|]$. We start by computing $l = \mathsf{LCE}(A[i], A[j])$. If $\max(i, j) + l = n + 1$, then $T_{A[i]} \prec T_{A[j]} \Leftrightarrow A[i] > A[j]$. Else, we report $T_{A[i]} \prec T_{A[j]} \Leftrightarrow T[A[i] + l] \prec T[A[j] + l]$. Since one comparison takes $\mathcal{O}(1)$ time, the sorting takes overall $\mathcal{O}(|S| \log |S|)$ time. $\mathsf{PA}_S[1..|S|]$ can be computed analogously using the same LCE query data structure for $\mathsf{rev}(T)$. □

3.2.5 Lemma. *Let* $T[i, i + l)$ *be a substring. Given* $\mathsf{PA}_S[1..|S|]$ *and* $\mathsf{SA}_S[1..|S|]$ *and a data structure for* $\mathcal{O}(1)$ *time* LCE *queries, we can compute* $\mathsf{piv}_S(T[i, i + l))$ *and* $\mathsf{siv}_S(T[i, i + l))$ *in* $\mathcal{O}(\log |S|)$ *time and* $\mathcal{O}(1)$ *additional space.*

Proof. To compute $[x_1, x_2] = \mathsf{siv}_S(T[i, i + l))$, we perform binary searches over the interval $[1, |S|]$ for x_1 and x_2, respectively.

Algorithm 9: min-sample($[i, i + l)$)

```
1  [b, e] ← [1, |S|];
2  while e > b do
3      m ← ⌊(b + e)/2⌋;
4      p ← S[SA_S[m]];
5      l_m ← LCE(p, i);
6      if l_m ⩾ l then
7          e ← m;
8      else if p + l_m = n + 1 ∨ T[p + l_m] ≺ T[i + l_m] then
9          b ← m + 1;
10     else
11         e ← m − 1;
12 if [b, e] = ∅ ∨
       LCE(S[SA_S[b]], i) < l then
13     return ∅;
14 return b;
```

The algorithm min-sample(i, l) (see Algorithm 9) returns the rank x_1 of the lexicographically smallest sample that starts with $T[i, i + l)$, or $\perp$ if it does not exist. We start with the interval $[b, e] = [1, |S|]$. As long as $e > b$, we consider the rank $m = (b + e)/2$ in the middle between b and e, compute its corresponding suffix $p = S[\mathsf{SA}_S[m]]$, and its match length $l_m = \mathsf{LCE}(p, i)$ with T_i. If $l_m \geqslant l$, then $T[i, i + l) = T[p, p + l)$, hence $x_1 \in [b, m]$ (see the left-hand side in Figure 3.6), so we set $e \leftarrow m$. Else, if $T_p \prec T_i$ (see the right-hand side), then $x_1 \in (m, e]$, so we set $b \leftarrow m + 1$. Else ($T_i \prec T_p$), then $x_1 \in [b, m)$ (see the left-hand side), so we set $e \leftarrow m - 1$.

At each step, we only filter out samples whose corresponding length-l substrings are either lexicographically larger (lines 7 and 11) or smaller (line 9) than $T[i, i + l)$, or are equal to $T[i, i + l)$, but do not correspond to the lexicographically smallest sampled suffix $T_{S[\mathsf{SA}_S[x_1]]}$ starting with $T[i, i + l)$ (line 7). Furthermore, the search interval $[b, e]$ is halved in each iteration, hence the while loop runs through $\mathcal{O}(\log |S|)$ iterations.

If $[x_1, x_2] = \varnothing$, then either $[b, e] = \varnothing$ or $b = e \wedge \mathsf{LCE}(S[\mathsf{SA}_S[b]], i) < l$, and we return $\varnothing$ in line 12. Else ($[x_1, x_2] \neq \varnothing$), we get $[b, e] = [x_1, x_1] \neq \varnothing$ and $\mathsf{LCE}(S[\mathsf{SA}_S[b]], i) \geqslant l$, and return $b = x_1$ in line 14.

Computing x_2 and $\mathsf{piv}_S(T[i, i + l)$ is analogous. □

Recall that in the algorithm for the exact LZ77 factorization, we compute pairs of sparse prefix- and suffix array intervals w.r.t. a sampling of text positions. By choosing

this sampling of text positions as in Definition 3.2.6, we can limit the number of computed pairs of sparse prefix- and suffix array intervals per phrase to δ for an integer parameter $\delta \geqslant 1$.

3.2.6 Definition. Let $S = [p_1, p_2, ..., p_{|S|}] \subseteq [1, n]$ with $p_1 < p_2 < ... < p_{|S|}$ be a subset (sampling) of text positions.

- **Leftmost-Substring-Covering.** We say that S is *leftmost-substring-covering* iff in every leftmost occurrence $T[i, i + l)$ of a substring, at least one position is sampled, i.e, $S \cap [i, i + l) \neq \varnothing$.

- δ-**Density.** Let $\delta \geqslant 1$ be an integer parameter. We say that S is δ-dense iff there is at least one sample in every size-δ window, i.e, $\forall w = [i, i + \delta) \subseteq [1, n] : S \cap w \neq \varnothing$.

A δ-dense leftmost-substring-covering sampling can be constructed by including exactly the end positions of phrases in an LZ-like factorization and adding additional samples.

3.2.7 Lemma. *We can construct a δ-dense leftmost-substring-covering sample-set $\mathcal{S}$ of size $|\mathcal{S}| \leqslant 3z + \lceil n/\delta \rceil$ in $\mathcal{O}(n/\log_\sigma n + n/\delta)$ time and space.*

Proof. Let $\mathcal{F} = \langle s_1, l_1 \rangle, ..., \langle s_{z'}, l_{z'} \rangle$ be the LZ-like factorization resulting from Theorem 3.1.8, where s_i and l_i are the source and the length of the i-th phrase, respectively. Let $p_i = \sum_{j=1}^{i} \max(1, l_j)$ for $i \in [1, z']$ be the end position of its i-th phrase.

To construct $\mathcal{S}$, we compute $\mathcal{F}$ in $\mathcal{O}(n/\log_\sigma n)$ time and space and choose $\mathcal{S} = \mathcal{E} \cup \Delta$, where $\mathcal{E} = \{p_i \mid i \in [1, z']\}$ is the set of end positions of phrases in $\mathcal{F}$ and $\Delta = \{i\delta + 1 \mid i \in [0, \lfloor n/\delta \rfloor]\}$ is a δ-dense sampling set. Therefore, $\mathcal{S}$ is also δ-dense. Computing Δ takes $\mathcal{O}(n/\delta)$ time, hence the construction takes overall $\mathcal{O}(n/\log_\sigma n + n/\delta)$ time.

Now, we show via contradiction that $\mathcal{E}$ is leftmost-substring-covering. Suppose there was a leftmost occurrence $T[i, i + l)$ of a substring, and $\mathcal{S} \cap [i, i + l) = \varnothing$. Then, $T[i, i + l)$ must be contained in a referencing phrase f_j of $\mathcal{F}$. Since f_j has a previous occurrence, this contradicts the fact that $T[i, i + l)$ is a leftmost occurrence of a substring, hence $\mathcal{E}$ and also $\mathcal{S}$ is leftmost-substring-covering.

It holds $|\mathcal{E}| = z' \leqslant 3z$ (according to Theorem 3.1.8) and $|\Delta| = \lceil n/\delta \rceil$, hence $|\mathcal{S}| \leqslant |\mathcal{E}| + |\Delta| = 3z + \lceil n/\delta \rceil$. □

3.2.8 Corollary. *We can construct a $\Theta(n/z)$-dense leftmost-substring-covering sample-set of size $\Theta(z)$ in $\mathcal{O}(n/\log_\sigma n)$ time and space.*

Proof. We choose $\delta = \Theta(n/z') = \Theta(n/z)$ [34] and construct $\mathcal{S}$ as in Lemma 3.2.7, hence $z \leqslant |\mathcal{S}| \leqslant 3z + n/\Theta(n/z)$ and therefore $|\mathcal{S}| = \Theta(z)$. □

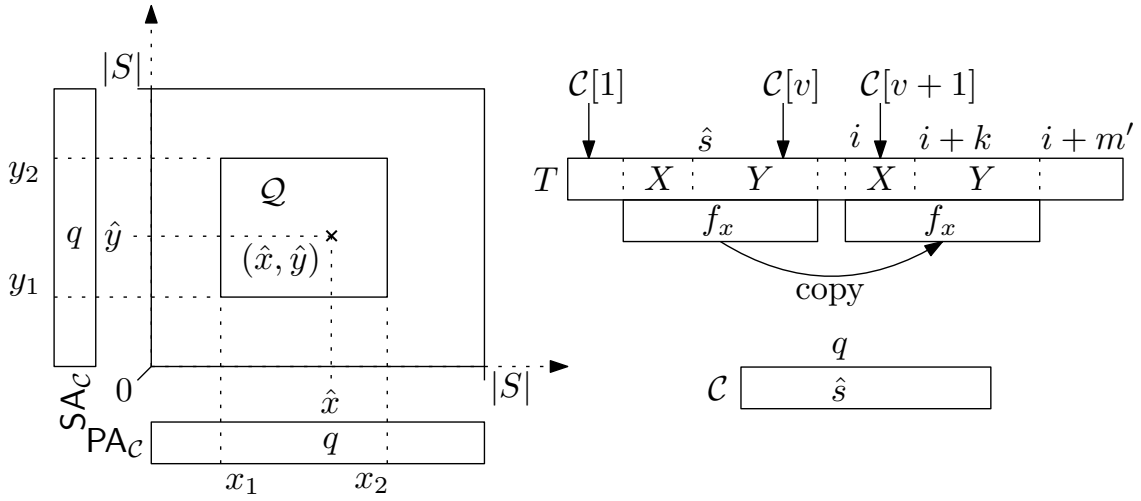


Figure 3.7: Illustration of the computation of a perfect factor during **(II) Far Sources** in Theorem 3.2.9.

3.2.2 Algorithm for finding Perfect Phrases

3.2.9 Theorem. *Let $\mathcal{C}$ be an array storing a sorted δ-dense leftmost-substring-covering sample set of size N. Let $\mathcal{R}$ be a data structure for insertion-only orthogonal range one-reporting (see Definition 3.2.1), let $i_{\mathcal{R}}$ and $q_{\mathcal{R}}$ be the times for inserting into and querying this data structure, and let $s_{\mathcal{R}}$ be its size after N insertions. After an $\mathcal{O}(n/\log_\sigma n + N \log N)$ time preprocessing, and in $\mathcal{O}(n/\log_\sigma n + N + s_{\mathcal{R}})$ space, we can maintain a subset $\mathcal{V} = \{\mathcal{C}[1], ..., \mathcal{C}[v]\} \subseteq \mathcal{C}$ of sample positions with the following operations.*

- *Insert the next sample $\mathcal{C}[v+1] \in \mathcal{C} \backslash \mathcal{V}$ into $\mathcal{V}$ in $\mathcal{O}(i_{\mathcal{R}})$ time, and*
- *Given a text position $i \in (\mathcal{C}[v], \mathcal{C}[v+1]]$, compute a perfect phrase for destination i, i.e, find the maximum length l such that there exists a source $s \in [1, i)$ with $T[i, i+l) = T[s, s+l)$, in $\mathcal{O}(\delta \log l(\log N + q_{\mathcal{R}}))$ time.*

Proof. **Preprocessing.** We build the data structure for $\mathcal{O}(1)$ time LCE queries in $\mathcal{O}(n/\log_\sigma n)$ space from Theorem 2.8.4 for T and $\mathsf{rev}(T)$. This takes $\mathcal{O}(n/\log_\sigma n)$ time and space. We also compute $\mathsf{PA}_{\mathcal{C}}$ and $\mathsf{SA}_{\mathcal{C}}$ in $\mathcal{O}(n/\log_\sigma n + N \log N)$ time and $\mathcal{O}(N)$ additional space using Lemma 3.2.4. Then, we compute $\mathsf{PA}_{\mathcal{C}}^{-1}$ and $\mathsf{SA}_{\mathcal{C}}^{-1}$ according to Definition 3.2.2 in $\mathcal{O}(N)$ time and space. As required, the preprocessing takes $\mathcal{O}(n/\log_\sigma n + N \log N)$ time, and we need $\mathcal{O}(n/\log_\sigma n + N + s_{\mathcal{R}})$ space.

Invariant. We maintain that $\mathcal{R}$ holds exactly the points $\mathcal{P}_{\mathcal{V}} = \{(\mathsf{PA}_{\mathcal{C}}^{-1}[1], \mathsf{SA}_{\mathcal{C}}^{-1}[1]), ..., (\mathsf{PA}_{\mathcal{C}}^{-1}[v], \mathsf{SA}_{\mathcal{C}}^{-1}[v])\}$.

Insertions. Given the next sample $\mathcal{C}[v+1]$ to insert, we compute the point $p = (\mathsf{PA}_{\mathcal{C}}^{-1}[v+1], \mathsf{SA}_{\mathcal{C}}^{-1}[v+1])$ in $\mathcal{O}(1)$ time and insert p into $\mathcal{R}$ in $\mathcal{O}(i_{\mathcal{R}})$ time to maintain the invariant.

Queries. To compute a perfect phrase with length l for destination $i \in (\mathcal{C}[v], \mathcal{C}[v+1]]$, we use two methods, and keep track of the source s' that yields the maximum LCE with position i (if it exists).

(I) Close Sources. For $p \in [i-\delta, i)$, we compute $\mathsf{LCE}(p, i)$. If s' does not exist, or $\mathsf{LCE}(p,i) > \mathsf{LCE}(s', i)$, then we update $s' \leftarrow p$. This takes $\mathcal{O}(\delta)$ time overall.

(II) Far Sources. Here, we find sources that consist of a head X and a tail Y, where the head is the suffix of a sampled prefix before i, and the tail is a prefix of the suffix starting at the same sample (see Figure 3.7).

For each possible head length $k \in [1, \delta]$, we compute $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k))$. If $[x_1, x_2] = \varnothing$, then we discard this k. Else, we compute the maximum length $l' \in [k, n-i]$ such that we can combine the tail $T[i+k-1, i+l')$ with the head $T[i, i+k)$, i.e, there is a sample $q \in \mathcal{V}$ with $q \leqslant \mathcal{C}[v]$ that occurs both in $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$, where $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(T[i+k-1, i+l'))$. Since $T[i, i+l') = T(q-k, q+l'-k]$, and $q \leqslant \mathcal{C}[v] < i$, the source $q-k+1$ then yields an LZ factor of length l' for destination i. If $l' > \mathsf{LCE}(s', i)$, then we update $s' \leftarrow q-k+1$.

Exponential Search. To compute l', we use an exponential search. Let $[b, e]$ be the current search interval during this search, let $m \in [b, e]$ be the candidate length, and let $[\hat{y}_1, \hat{y}_2] = \mathsf{siv}_{\mathcal{C}}(T[i+k, i+m))$. For each sample index $j \in [1, N]$, it holds

$$j \in \mathsf{PA}_{\mathcal{C}}[x_1, x_2] \cap \mathsf{SA}_{\mathcal{C}}[\hat{y}_1, \hat{y}_2] \cap [1, v] \Leftrightarrow (\mathsf{PA}_{\mathcal{C}}^{-1}[j], \mathsf{SA}_{\mathcal{C}}^{-1}[j]) \in \mathcal{P}_{\mathcal{V}} \cap ([x_1, x_2] \times [\hat{y}_1, \hat{y}_2]).$$

Therefore, we can query $\mathcal{R}$ with the rectangle $\mathcal{Q} = [x_1, x_2] \times [\hat{y}_1, \hat{y}_2]$ and proceed based on the result.

- **Case 1.** $\mathcal{R}$ reports $\mathcal{P}_{\mathcal{V}} \cap \mathcal{Q} = \varnothing$. Then, there is no sample before or at the position $\mathcal{C}[v]$ that occurs both in $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$, hence we continue the search for l' in the interval $[b, m)$.

- **Case 2.** The query returns a point $p = (\tilde{x}, \tilde{y}) \in \mathcal{P}_{\mathcal{V}} \cap \mathcal{Q}$. Then, we have a new source $\hat{s} = \mathcal{C}[\mathsf{SA}_{\mathcal{C}}[\tilde{y}]] - k + 1$ (which yields an LZ factor of length m for destination i), set $q \leftarrow \hat{s}$ and continue the search for l' in the interval $[m, e]$.

Running Time of (II) Far Sources. At each step of an exponential search for l', we compute one sparse suffix array interval and issue at most one query to $\mathcal{R}$. Therefore, one such search takes $\mathcal{O}(\log l'(\log N + q_{\mathcal{R}}))$ time. For each k, we compute one sparse prefix array interval and perform at most one such exponential search. Since there are $\leqslant \delta$ choices for k, this takes overall $\mathcal{O}(\delta \log l(\log N + q_{\mathcal{R}}))$ time.

Correctness. Now it remains to show that the computed phrase is perfect (of maximal length). Therefore, we will show that we find either the leftmost source λ of $T[i, i+l)$ or another source that yields an LZ factor with the same length. We distinguish between close and far leftmost sources.

- **Case 1.** $\lambda \in [i-\delta, i)$. Then, we consider λ for $p = \lambda$ during **(I) Close Sources**.

- **Case 2.** $\lambda < i-\delta$. If $l < \delta$, then $T[\lambda, \lambda+\delta]$ is the leftmost occurrence of a substring, hence there is a sample $\mathcal{C}[\xi] \in [\lambda, \lambda+\delta)$, because $\mathcal{C}$ is leftmost-substring covering. Else (if $l \geqslant \delta$), then the same holds due to the fact that $\mathcal{C}$ is δ-dense. Thus, for $k = \mathcal{C}[\xi]-\lambda+1$ during **(II) Far Sources**, we have $\xi \in \mathsf{PA}_{\mathcal{C}}[x_1, x_2] \cap \mathsf{SA}_{\mathcal{C}}[y_1, y_2] \cap [1, v]$, and for $m = l$, the query to $\mathcal{R}$ with the rectangle $[x_1, x_2] \times [y_1, y_2]$ will report a point $(\mathsf{PA}_{\mathcal{C}}^{-1}[\rho], \mathsf{SA}_{\mathcal{C}}^{-1}[\rho])$, where either $\rho = \xi$, or $\rho \in (\xi, v]$ is another sample index such that $T(\mathcal{C}[\rho]-k, \mathcal{C}[\rho]+l-k] = T[i, i+l)$. □

Algorithm 10: perfect-factor-naive(i)

```
1  ⟨s, l⟩ = ⟨0, T[i]⟩;
2  for p from max(1, i − δ) to i − 1 do
3      lce_p ← LCE(p, i);
4      if lce_p > l then
5          s ← p;
6          l ← lce_p;
7  for k from 1 to min(δ, n − i + 1) do
8      [x1, x2] ← spa-interval([i, i + k));
9      if [x1, x2] = ∅ then
10         continue;
11     exp-search for max m ∈ [k, n − i + 1] such that
12         [y1, y2] ← ssa-interval([i + k − 1, i + m));
13         if [y1, y2] = ∅ then
14             report false;
15         p = (x̃, ỹ) ← R.query([x1, x2], [y1, y2]);
16         if p = ⊥ then
17             report false;
18         if m > l then
19             l ← m;
20             s ← C[SA_C[ỹ]] − k + 1;
21     report true;
22 return ⟨s, l⟩;
```

3.2.2.1 Pseudocode

To better illustrate the algorithm described in Theorem 3.2.9, we discuss its pseudocode (see Algorithm 10). We begin by initializing the factor $\langle s, l\rangle$ as a literal factor $\langle 0, T[i]\rangle$ for destination i. This ensures that we return a literal factor in line 22 if we do not find a referencing factor. Lines 2-6 correspond to **(I) Close Sources** and trivially translate to pseudocode.

Lines 7-21 correspond to **(II) Far Sources**. Here, we consider each possible value for k, and compute the sparse prefix array (spa) interval $[x_1, x_2]$ of the head $T[i, i+k)$ (line 8). If it is empty, then we skip this k and continue with the next iteration. Else, we perform the exponential search for l' as described in Theorem 3.2.9. Here, m is the current candidate position in the overall search range $[k, n-i+1]$.

Analogously to line 8, we at first compute the sparse suffix array (ssa) interval $[y_1, y_2]$ of the tail $T[i+k-1, i+m)$. If it is empty, then we report back to the exponential search that $l' < m$ must hold (see line 14), because there is no such tail starting at a sample position. Else, we check if there is a sample whose prefix ends with the head $T[i, i+k)$ and whose suffix starts with the tail $T[i+k-1, i+m)$. More formally, we check if there is a sample that occurs both in $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$. To this end, we query $\mathcal{R}$ with the rectangle $[x_1, x_2] \times [y_1, y_2]$. If there is no such sample ($\mathcal{R}$ returns $p = \varnothing$), then we cannot combine the head with the tail, and report back to the exponential search that $l' < m$ must hold (see line 17). Else, the source $\mathcal{C}[\mathsf{SA}_{\mathcal{C}}[\tilde{y}]] - k + 1$ yields an LZ factor of length m with destination i, and we update the current source s and its length l if $m + k - 1 > l$ (see lines 18-20). Finally, we return the perfect factor $\langle s, l\rangle$ in line 22.

3.2.3 Main Algorithm

3.2.10 Theorem. *We can compute the LZ77 factorization in $\mathcal{O}(n/\log_\sigma n + z\log^{3+\epsilon} z)$ time and $\mathcal{O}(n/\log_\sigma n)$ space.*

Proof. We use Theorem 3.2.9 and compute perfect phrases from left to right (see Algorithm 11 and Figure 3.7). After computing phrase $f_x = T[i, i+l)$, we insert the samples $\mathcal{C} \cap [i, i+l)$ into $\mathcal{V}$ (see lines 4-6).

The preprocessing takes $\mathcal{O}(n/\log_\sigma n + N\log N) = \mathcal{O}(n/\log_\sigma n + z\log z)$ time and $\mathcal{O}(n/\log_\sigma n + N) = \mathcal{O}(n/\log_\sigma n)$ space. We implement $\mathcal{R}$ using Theorem 1 [38], which yields $i_{\mathcal{R}} = \mathcal{O}(\log^{3+\epsilon} N)$, $q_{\mathcal{R}} = \mathcal{O}(\log N)$ and $s_{\mathcal{R}} = \mathcal{O}(N)$ for an arbitrarily small constant $\epsilon > 0$. Inserting all N points into $\mathcal{V}$ takes $\mathcal{O}(N \cdot i_{\mathcal{R}}) = \mathcal{O}(N\log^{3+\epsilon} N) = \mathcal{O}(z\log^{3+\epsilon} z)$ time. Computing phrase f_i takes $\mathcal{O}(\delta\log|f_i|(\log N + q_{\mathcal{R}})) = \mathcal{O}((n/z)\log n\log z)$ time. Hence, computing all phrases takes $\mathcal{O}(z\log n\log z)$ time.

The overall time is $\mathcal{O}(n/\log_\sigma n + z\log z + z\log^{3+\epsilon} z + z\log n\log z) = \mathcal{O}(n/\log_\sigma n + z\log^{3+\epsilon} z)$. The overall space is $\mathcal{O}(n/\log_\sigma n)$. □

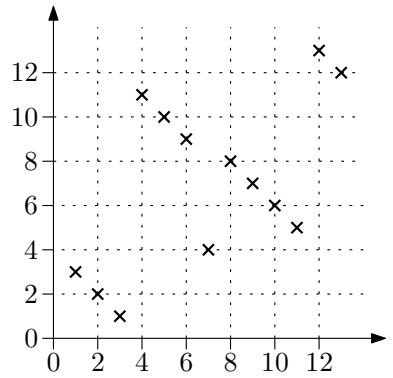
Algorithm 11: lz77-exact-naive()

```
1 i ← 1;
2 v ← 0;
3 while i ⩽ n do
4     while v < N ∧ C[v + 1] < i do
5         v ← v + 1;
6         R.insert(PA_C^{-1}[v], SA_C^{-1}[v]);
7     ⟨s, l⟩ ← perfect-factor-naive(i);
8     i ← i + max(1, l);
9     output ⟨s, l⟩;
```

3.2.11 Example. In Example 3.1.9, we computed the approximate factorization $\langle 0, a\rangle$, $\langle 0, b\rangle, \langle 0, c\rangle, \langle 1, 5\rangle, \langle 8, 5\rangle, \langle 8, 5\rangle, \langle 1, 8\rangle$. This yields $\mathcal{E} = \{1, 2, 3, 8, 13, 18, 26\}$. We set $\delta = \lceil n/z' \rceil = \lceil 26/7 \rceil = 4$. Thus, we get $\Delta = \{1, 5, 9, 13, 17, 21, 25\}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\mathcal{C} =$	1	2	3	4	5	8	9	13	17	18	21	25	26
$\mathsf{PA}_{\mathcal{C}} =$	1	4	12	2	5	6	13	7	8	9	10	3	11
$\mathsf{PA}_{\mathcal{C}}^{-1} =$	1	4	12	2	5	6	8	9	10	11	13	3	7
$\mathsf{SA}_{\mathcal{C}} =$	12	4	1	13	10	9	8	7	6	5	2	11	3
$\mathsf{SA}_{\mathcal{C}}^{-1} =$	3	11	13	2	10	9	8	7	6	5	12	1	4



$\mathcal{P} = \{(1, 3), (4, 11), (12, 13), (2, 2), (5, 10), (6, 9), (8, 8), (9, 7), (10, 6), (11, 5), (13, 12), (3, 1), (7, 4)\}$.

Iterations 1-3. In the first 3 iterations of the framework algorithm, we output the literal factors $\langle 0, a\rangle$, $\langle 0,\ b\rangle$ and $\langle 0, c\rangle$, and insert $(1, 3)$ and $(4, 11)$ into $\mathcal{R}$.

Iteration 4. In the fourth iteration ($i = 4$), we insert $(12, 13)$ into $\mathcal{R}$ and call perfect-factor-naive(4). Here, we consider the sources $1, 2$ and 3 in lines 2-6. For $p = 1$, we get $\mathsf{lce}_p = 5$ and set $\langle s, l\rangle = \langle 1, 5\rangle$. For $k = 1$, we get $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(a) = [1, 3]$ and start the exponential search with $m = k$. There, we have $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(a) = [1, 3]$ and the query $\mathcal{Q} = [1, 3] \times [1, 3]$ to $\mathcal{R}$ returns $p = (1, 3) \in [1, 3] \times [1, 3] = \mathcal{Q}$, and we report **true** in line 21. The exponential search then runs through lines 12-21 and reports **true** for $m = 2$, $m = 4$ and $m = 5$. However, since $m \not> 5 = l$, we do not reach lines 19-20. For $k = 2$ and $k = 3$, we have $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(ab) = [4, 7]$ and $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(abc) = [12, 12]$, respec-

tively, and the exponential searches end with $m = 5$ and $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(bcab) = [11, 11]$ and $[y_1, y_2] = \mathsf{piv}_{\mathcal{C}}(cab) = [13, 13]$, respectively. Again, s and l are not altered, because $m \not> 5 = l$. Finally, we output $\langle 1, 5\rangle$ in line 22.

Iteration 5. In the next iteration, we have $i = 9$, and insert $(2, 2)$, $(5, 10)$ and $(6, 9)$ into $\mathcal{R}$. In perfect-factor-naive(9), we set $\langle s, l\rangle = \langle 8, 10\rangle$ for $p = 8$. For $k = 1$ and $m = 1$, the query $\mathcal{Q} = [4, 11] \times [4, 11]$ returns $p = (4, 11)$, but since $m = 1 \not> 10 = l$, we do not reach lines 19-20. The exponential search ends with $m = 10$ and $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(b^{10}) = [9, 9]$. Here, $\mathcal{R}$ is queried with $\mathcal{Q} = [4, 11] \times [9, 9]$ and returns $p = (6, 9)$. However, since $m = 10 \not> 10 = l$, s and l are not altered.

Iteration 6. In the last iteration, we ha ve $i = 19$, and insert $(8, 8)$, $(9, 7)$, $(10, 68)$ and $(11, 5)$ into $\mathcal{R}$. In perfect-factor-naive(19), we do not find any close source that yields an LZ factor. For $k = 1$ and $m = 1$, we get $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(a) = [1, 3]$ and $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(a) = [1, 3]$, the query $\mathcal{Q} = [1, 3] \times [1, 3]$ to $\mathcal{R}$ returns e.g. $p = (2, 2)$, and we set $s \leftarrow \mathcal{C}[\mathsf{SA}_{\mathcal{C}}[2]] - k + 1 = 4 - 1 + 1 = 4$ and $l \leftarrow m = 1$ in lines 19-20. For $m = 2$, $m = 4$ and $m = 8$, the query to $\mathcal{R}$ succeeds. For $m = 8$, i.e, $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(abcabcab) = [3, 3]$, the query $[1, 3] \times [3, 3]$ returns $p = (1, 3)$, and we set $s \leftarrow \mathcal{C}[\mathsf{SA}_{\mathcal{C}}[3]] - k + 1 = 1 - 1 + 1 = 1$ and $l \leftarrow m = 8$ in lines 19-20. For $k = 2$ and $k = 3$, we find the same LZ factor, but do not update $\langle s, l\rangle$, because $m \not> l = 8$. Finally, we output $\langle 1, 8\rangle$ as the final factor.

Chapter 4

Implementation

Now, we discuss how we implemented the LZ77 algorithms from Chapter 3. Suppose we have a text T of length $n = 2^{32}$ with an alphabet of size $\sigma = 256$. According to Theorem 3.1.8, we then have to choose $\tau \leqslant \lfloor \log_2 2^{32}/(8\lceil \log_2 256 \rceil) \rfloor = 32/64 = 1/2$. Therefore, the running time of the LZ77 3-Approximation (see Theorem 3.1.8) is a purely theoretical result. However, the algorithm can still be adapted to perform well in practice by choosing a larger value for τ. Then, we cannot use the index described in Lemma 3.1.1, because it requires $m \leqslant \log_2 n/((2+\epsilon)\lceil \log_2 \sigma \rceil)$, i.e, for $n = 2^{32}$ and $\sigma = 256$, we have $m \leqslant 32/(2 \cdot 8) = 2$, which is impractical. Therefore, we will discuss practical approaches for compressing gaps between the LPF phrases in Section 4.2. In practice, we also admit LPF phrases that are shorter than 2τ, as this increases the compression rate.

4.1 $\mathsf{LCE}^{\mathsf{L}}$ Queries

Let $\mathsf{LCE}^{\mathsf{L}}(i,j) = \max\{l \in [1, \min\{i,j\}] \mid T(i-l,i] = T(j-l,j]\}$ for $i,j \in [1,n]$. To answer $\mathsf{LCE}^{\mathsf{L}}$ queries, we could store $\mathsf{rev}(T)$ and use the $\mathcal{O}(1)$ time LCE data structure from Theorem 2.8.4 for $\mathsf{rev}(T)$. However, if the mismatch before i and j occurs within the first 2048 characters, then scanning for the mismatch is faster in practice. Since storing $\mathsf{rev}(T)$ increases peak memory consumption, and we will only issue $\mathsf{LCE}^{\mathsf{L}}$ queries where scanning is faster, we do not use Theorem 2.8.4 for $\mathsf{rev}(T)$ in practice.

There are only two cases where we have to answer $\mathsf{LCE}^{\mathsf{L}}$ queries. The first is in Section 4.3, where we extend the computed LPF phrases to the left down to the end of the last computed LPF phrase. Since there is a sample in each size-τ window in T, we scan at most τ characters. Since we set $\tau = 512$ in practice, scanning is faster than using Theorem 2.8.4 in this case. The second case is during the computation of sparse prefix array intervals of length-$\leqslant \delta$ substrings of T (see Theorem 3.2.9). In practice, we limit δ to 256 (see Section 4.5.5), hence scanning is again faster.

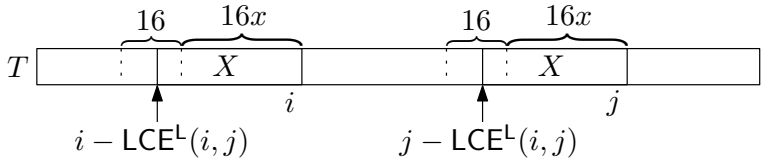


Figure 4.1: Illustration of the computation of a $\mathsf{LCE}^{\mathsf{L}}(i,j)$, where $X = T(i-\mathsf{LCE}^{\mathsf{L}}(i,j), i]$.

We answer the query $\mathsf{LCE}^{\mathsf{L}}(i,j)$ by at first scanning for a mismatch using 128-bit instructions, i.e, $128/8 = 16$ characters are matched simultaneously. Let x be the number of matched 16-character blocks, i.e, x is minimal such that $T(i-16(x+1), i-16x] \neq T(j-16(x+1), j-16x]$ (see Figure 4.1). Then, we return $\lfloor \mathsf{clz}(\mathsf{int}(T(i-16(x+1), i-16x]) \oplus \mathsf{int}(T(j-16(x+1), j-16x]))/\lceil \log \sigma \rceil \rfloor$ similar to Theorem 2.8.4, where $\oplus$ is the bitwise exclusive or operator and $\mathsf{clz}(B)$ returns the number of leading zeros in the bit-string B.

4.2 Compressing Gaps

We can reduce the size of the index for finding patterns in gaps by omitting the requirement that the computed factors are of maximal length. This can be done by storing previous occurrences for patterns of specific lengths only and allowing even more approximation.

4.2.1 Index Data Structures

4.2.1.1 Tries

4.2.1 Definition. Given a set $\mathcal{S} = \{S_1, S_2, ..., S_N\}$ of prefix-free strings, the trie $\mathcal{T}_{\mathcal{S}}$ for $\mathcal{S}$ is the minimal tree with single-character edge labels and root r such that for each $S \in \mathcal{S}$, there exists a path $\langle r, ..., v\rangle$ from r to a leaf v whose concatenation of edge labels is S. A compact trie is then defined analogously, but admits edge labels with multiple characters and requires each internal node to have at least two children.

The index for locating short patterns can then be implemented with a compact trie $\mathcal{T}_{\mathcal{L}}$ over all substrings $\mathcal{L}$ of T with length l, where each trie node v is marked with the first occurrence $f(v)$ of $\overline{v}$ in T, and l is a suitably small maximum pattern length. Let P be a pattern, and let $P[1,p]$ be the longest prefix of P that occurs in T. Since the suffix tree is the compact trie over all suffixes of T, we can use the same method to search for a pattern in $\mathcal{T}_{\mathcal{L}}$ (see Section 2.3). Suppose that we have arrived at (the end of) some edge (v, v') of $\mathcal{T}_{\mathcal{L}}$. Now, we can report $f(v')$ as a first of occurrence of $P[1,p)$. We can also use a more sophisticated trie implementation like a *top-k trie* [11], which stores the k most frequent substrings of T.

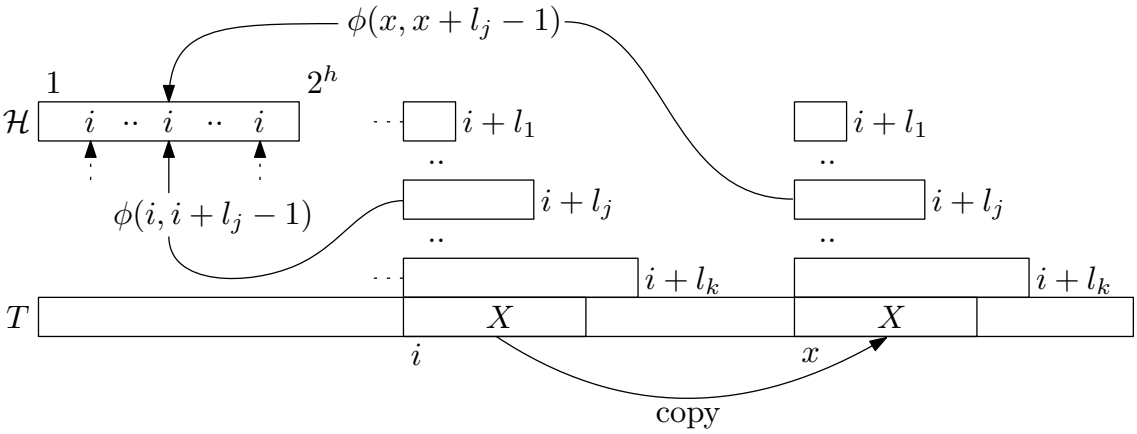


Figure 4.2: Illustration of the rolling hash index. In this case, $j \in [1, k]$ maximizes $\mathsf{LCE}(\mathcal{H}[\phi(x, x + l_j - 1) \circledcirc \mathsf{mask} + 1], x)$.

4.2.1.2 Hashing

Another option is to use hashing. We build k hash maps $\mathcal{H}_1, \mathcal{H}_2, ..., \mathcal{H}_k$ for pattern lengths $l_1, l_2, .., l_k$, where $\mathcal{H}_i$ maps $\mathsf{int}(S) \rightarrow p$, for each length-$i$ substring S of T with first occurrence p. If we want to compute the longest previous occurrence of a substring of T starting at position i that we can find with $\mathcal{H}_1, \mathcal{H}_2, ..., \mathcal{H}_k$, then we compute the set $\mathsf{Occ} = \{o = \mathcal{H}[\mathsf{int}(T[i, i + l_j))] \mid \bot \neq o < i \wedge j \in [1, k]\}$ and output the position $p = \arg\max_{o \in \mathsf{Occ}}\{\mathsf{LCE}(o, i)\}$ and length $\mathsf{LCE}(p, i)$.

Since there can generally be $\mathcal{O}(\min(\sigma^l, n))$ distinct substrings of length l in a string, we have $|\mathcal{H}_i| = \mathcal{O}(\min(\sigma^l, n))$ in general. Thus, this approach works only for very short pattern lengths, and patterns that occur infrequently in T occupy an unnecessarily large amount of space.

4.2.1.3 Hashing + Fingerprinting

This issue can be solved by using fingerprinting (see Section 2.6), i.e, we can use fingerprints of substrings of arbitrary lengths to address occurrences stored in a global hash table of arbitrary size. This approach requires a predictable amount of memory and ensures that only positions of frequently occurring patterns are stored in the global hash table.

For an integer parameter h and pattern lengths $L = \{l_1, l_2, .., l_k\}$, we incrementally build a global hash table as an array $\mathcal{H}$ of size $|\mathcal{H}| = 2^h$ while factorizing T greedily from left to right. We require that $|\mathcal{H}|$ is a power of 2 in order to quickly compute, for a given fingerprint ϕ, a position $\phi \bmod 2^h = \phi \circledcirc \mathsf{mask} + 1$ in $\mathcal{H}$, where $\mathsf{mask} = 2^h - 1$ and $\circledcirc$ is the "bitwise-and" operator. We initialize $\mathcal{H} = [\bot, ..., \bot]$ and implement rolling Karp-Rabin fingerprinting (see Section 2.6) for the lengths L.

At position i, we set $\mathcal{H}[\phi(i, i + l_j - 1) \circledcirc \mathsf{mask} + 1] \leftarrow i$ for each $j \in [1, k]$ (see the left-hand side in Figure 4.2) and thereby possibly overwrite earlier occurrences of potentially

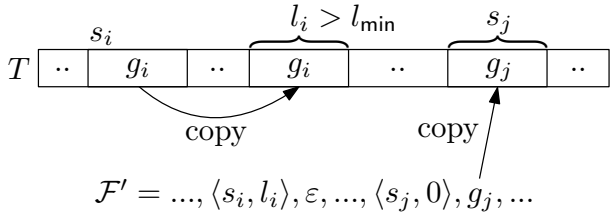


Figure 4.3: Illustration of the intermediary representation $\mathcal{F}'$. The i-th factor is a copy factor ($l_i > 0$), and $l_j = 0$ indicates that the gap g_j is non-empty.

different patterns (due to hash collisions and the modulo operation). Suppose we have factorized T and maintained $\mathcal{H}$ as described up to position $x - 1$. Now, we want to compute the longest previous occurrence of a substring of T starting at position x that we can find with $\mathcal{H}$. Then, we compute the set $\mathsf{Occ} = \{o = \mathcal{H}[\phi(x, x + l_j - 1) \circledcirc \mathsf{mask} + 1] \mid o \neq \bot \wedge j \in [1, k]\}$. By the incremental construction of $\mathcal{H}$, it holds $\forall o \in \mathsf{Occ} : o < x$, hence we output $\arg\max_{o \in \mathsf{Occ}}\{\mathsf{LCE}(o, x)\}$ (see the right-hand side in Figure 4.2).

4.2.2 Separately compressing gaps

Instead of factorizing the gaps, we can create an intermediary representation of the text, and then compress it using a standard compression algorithm. This also tackles the final "encoding" step of practical compression algorithms.

4.2.2.1 Variant 1

Let $g_0 f_1 g_1 f_2 g_2 ... f_{z'} g_{z'} = T$ be a gapped factorization resulting from Lemma 3.1.5, where f_i is the i-th (perfect) LPF phrase, and g_i is the gap after the i-th LPF phrase. Instead of closing the gaps using factors, we construct a string $\mathcal{G} = g_0 g_1 ... g_{z'}$ consisting of the concatenation of all gaps, and separately compress $\mathcal{G}$ using a suitably space-efficient compression algorithm. Let $\mathsf{comp}(\mathcal{G})$ be the compressed representation of $\mathcal{G}$. We then output $\langle \mathcal{F}_\mathcal{G}, \mathsf{comp}(\mathcal{G}) \rangle$, where $\mathcal{F}_\mathcal{G} = \langle 0, 0, g_0 \rangle, \langle |f_1|, s_1, |g_1| \rangle, ..., \langle |f_{z'}|, s_{z'}, |g_{z'}| \rangle$ and s_i is a source of the i-th factor.

To decompress this representation, we at first decompress $\mathsf{comp}(\mathcal{G})$. Then, we iterate over T, $\mathcal{G}$ and $\mathcal{F}_\mathcal{G}$ and maintain the current positions i in T, j in $\mathcal{G}$ and x in $\mathcal{F}_\mathcal{G}$. For each $\langle l_x, s_x, |g_x| \rangle$, we write $T[s_x, s_x + l_x)$ to $T[i, i + l_x)$, increment i by l_x, write $\mathcal{G}[j, j + |g_x|)$ to $T[i, i + |g_x|)$ and increment i and j by $|g_x|$.

4.2.2.2 Variant 2

We can also let the compression algorithm compress the LPF phrases. Again, let $g_0 f_1 g_1 f_2 g_2 ... f_{z'} g_{z'} = T$ be a gapped factorization resulting from Lemma 3.1.5. For each $g f g'$ in the gapped factorization, where f is a referencing factor that is shorter than l_{min}, we plainly

store f, i.e, we replace gfg' with a new gap $g'' = gfg'$. This can increase the compression ratio by a few percent in practice (we set $l_{\mathsf{min}} = 64$). Let $g'_0 f'_1 g'_1 f'_2 g'_2 ... f'_{z''} g'_{z''} = T$ be the resulting factorization.

Now, we construct the sequence $\mathcal{F}' = \langle s_1, l_1 \rangle, g_1, \langle s_2, l_2 \rangle, g_2, ..., \langle s_{z''}, l_{z''} \rangle, g_{z''}$, where for each $i \in [1, z'']$, either $l_i = 0$ indicates that the gap g_i of length $|g_i| = s_i > 0$ follows, or else ($l_i > 0$), $\langle d_i, l_i \rangle$ represents a referencing factor of length $l_i > 0$ with distance d_i, and $g_i = \varepsilon$ (see Figure 4.3). Finally, we output $\mathsf{comp}(\mathcal{F}')$.

To decompress this representation, we at first decompress $\mathsf{comp}(\mathcal{F}')$. Then, we iterate over $\mathcal{F}'$ and maintain the current positions i in T and x in $\mathcal{F}'$. For each $\langle |g_x|, 0 \rangle, g_x$ in $\mathcal{F}'$, we write g_x to $T[i, i + |g_x|)$ and increment i by $|g_x|$. For each $\langle d_x, l_x \rangle, \varepsilon$ in $\mathcal{F}'$, we write $T[i - d_x, i - d_x + l_x)$ to $T[i, i + l_x)$ and increment i by l_x.

4.2.2 Example. Let us continue Example 2.7.3. There, we computed the 3-synchronizing set $S = \{1, 4, 14, 15, 16, 19\}$. If we compute the LPF phrases as in the 3-Approximation (Section 3.1.2), and also admit LPF phrases that are shorter than 2τ, then we get $\mathcal{F}' = \langle 3, 0 \rangle, abc, \langle 3, 5 \rangle, \varepsilon, \langle 6, 0 \rangle, bbbbbb, \langle 1, 4 \rangle, \varepsilon, \langle 18, 8 \rangle, \varepsilon$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b

4.3 LPF Phrases

In practice, we want to compute an array of LPF phrase tuples $\mathsf{LPF}[1..N]$, where (**i**) $\mathsf{LPF}[i] = \langle b_i, e_i, s_i \rangle$, and $T[b_i, e_i) = T[s_i, s_i + b_i - e_i)$, (**ii**) the phrases do not overlap, i.e, $[b_i, e_i) \cap [b_j, e_j) = \varnothing$ holds for all $i, j \in [1, N]$ with $i \neq j$, and (**iii**) at the last position in LPF there is a sentinel entry $\mathsf{LPF}[N] = \langle n + 1, n + 1, n + 1 \rangle$. (**ii**) makes it easier for us to factorize (see Section 4.4) or compress (see Section 4.2.2) the gaps afterwards. In the following, we use the abbreviations $\mathsf{LPF}[i].b = b_i$, $\mathsf{LPF}[i].e = e_i$ and $\mathsf{LPF}[i].s = s_i$.

4.3.1 LZ77 3-Approximation

Algorithm 12 shows the computation of the LPF phrases during the 3-Approximation. Overall, we iterate with i once over S, and compute the phrases from left to right. We maintain that the last output LPF phrase ends at position $x - 1$.

Suppose we have computed an LPF phrase for $S[i]$. To compute the next LPF phrase, we find the last sample after $S[i]$ and up to x if it exists, or the first sample after x, else. After incrementing i in lines 3-4, now $S[i]$ is this sample. Now, we compute an LPF phrase $\langle b', e', s' \rangle$ for $S[i]$ using NSV_S and PSV_S (see lines 6-9). If $S[i] > x$, then we extend it as far as possible to the left down to x (see lines 10-13). If $S[i] < x$, then we shorten it from the left by $x - S[i]$ characters such that it does not overlap with the last output phrase

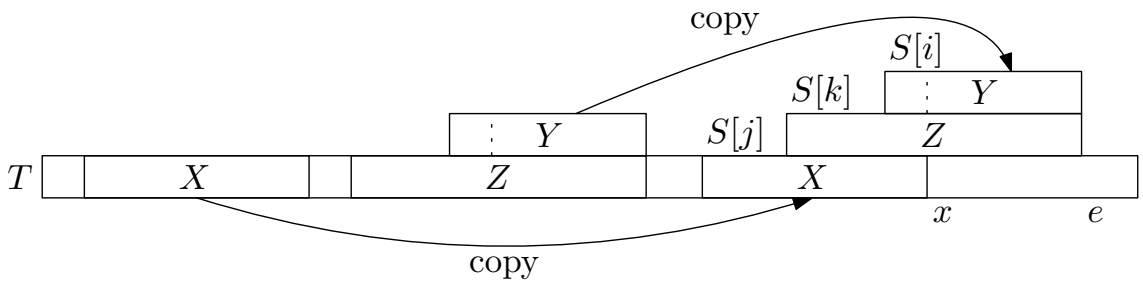


Figure 4.4: Illustration of Observation 4.3.1. Since each sample $S[k]$ with $j < k < i$ yields an LPF phrase that ends before e, we can skip it.

(see lines 14-16). Finally, we output the resulting LPF phrase, if it has length > 1, and update x (see lines 20-22).

Algorithm 12: compute-lpf

```
1  x, i ← 1;
2  while i ⩽ |S| do
3      while i + 1 ⩽ |S| ∧ S[i + 1] ⩽ x do
4          i ← i + 1;
5      ⟨b, e, s⟩ ← ⟨0, 0, 0⟩;
6      if PSV_S[SA_S^{-1}[i]] ≠ 0 then
7          s′ ← S[SA_S[PSV_S[SA_S^{-1}[i]]]];
8          b′ ← S[i];
9          e′ ← S[i] + LCE(s′, S[i]);
10         if S[i] > x then
11             lce_l ← LCE_{⩽S[i]−x}(s′ − 1, S[i] − 1);
12             s′ ← s′ − lce_l;
13             b′ ← b′ − lce_l;
14         else if S[i] < x then
15             b′ ← x;
16             s′ ← s′ + x − S[i];
17         if e′ − b′ > e − b then
18             ⟨b, e, s⟩ ← ⟨b′, e′, s′⟩;
19     Repeat lines 6-15 with NSV_S instead of PSV_S;
20     if e − b > 1 then
21         output ⟨b, e, s⟩;
22         x ← e;
```

If $S[i] \geqslant x$ holds in line 10, then we have not skipped any samples in line 4 during this iteration. Else ($S[i] < x$), then we have skipped the samples $S[j, i)$, where $S[j]$ is the sample that we used to compute the last LPF phrase. Now, we show that none of those skipped samples yields a longer LPF phrase than $S[i]$.

4.3.1 Observation. Let $e \geqslant S[i] + 2\tau$ be the end of the computed LPF phrase for sample $S[i]$, and suppose there was a skipped sample $S[k]$ with $j < k < i$ that yields an LPF phrase ending at some position $\hat{e} > e$. Since $T[S[k], \hat{e}]$ has a previous occurrence and fully contains $T[S[i], e]$, $T[S[i], \hat{e}]$ also has a previous occurrence. However, by the same argument as in Construction 2.4.3, $T[S[i], e]$ is the longest prefix of $T_{S[i]}$ that has a previous occurrence, which contradicts the assumption that such sample $S[k]$ exists.

Note that each computed LPF phrase (for some sample $S[i]$) is only then perfect if $T[S[i], S[i] + 2\tau)$ has a previous occurrence.

4.3.2 Example. In Example 2.7.3, we computed the 3-synchronizing set $S = \{1, 4, 14, 15, 16, 19\}$. In Example 4.2.2, we computed the phrases $\langle 4, 9, 1\rangle, \langle 15, 19, 14\rangle$ and $\langle 19, 27, 1\rangle$. Algorithm 12 instead extends the second phrase by 5 characters to the left, i.e, it instead outputs $\langle 9, 19, 8\rangle$. It additionally considers the sample $S[5] = 16$. However, the phrase it yields is identical to the last computed phrase $\langle 9, 19, 8\rangle$ (after extending it to the left). Note that all 3 computed phrases are part of the exact LZ77 factorization.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b

4.3.2 LZ77 LPF/LNF Approximation

We have also implemented a version of the 3-approximation that considers an additional set of phrases. In addition to computing LPF phrases in T using PSV_S and NSV_S, we use a string synchronizing set S' of $\mathsf{rev}(T)$ to compute a set of LNF phrases in $\mathsf{rev}(T)$ using $\mathsf{PGV}'_{S'}$ and $\mathsf{NGV}'_{S'}$ (see Definition 4.3.3).

4.3.2.1 Preliminaries

4.3.3 Definition. Let $S'[1..|S'|]$ be a synchronizing set of $\mathsf{rev}(T)$. Let $\mathsf{SA}'_{S'}$ be the sparse suffix array of $\mathsf{rev}(T)$ w.r.t. S', and let $\overline{\mathsf{SA}}^{-1}_{S'}[1..|S'|]$ be the inverse sparse suffix array, i.e, $\forall i \in [1, |S'|] : \overline{\mathsf{SA}}^{-1}_{S'}[\mathsf{SA}'_{S'}[i]] = i$. Let $\mathsf{PGV}'_{S'}[1..|S'|]$ be the sparse previous greater value array w.r.t. S', i.e, $\mathsf{PGV}'_{S'}[i] = 0$ if $\forall j \in [1, i) : \mathsf{SA}'_{S'}[j] < \mathsf{SA}'_{S'}[i]$ and $\mathsf{PGV}_S[i] = \max\{j \in [1, i) \mid \mathsf{SA}'_{S'}[j] > \mathsf{SA}'_{S'}[i]\}$, else. Analogously, we call the array $\mathsf{NGV}'_{S'}[1..|S'|]$ the sparse next greater value array w.r.t. S', where $\mathsf{NGV}'_{S'}[i] = 0$ if $\forall j \in (i, |S'|] : \mathsf{SA}'_{S'}[j] < \mathsf{SA}'_{S'}[i]$ and $\mathsf{NGV}'_{S'}[i] = \min\{j \in (i, |S'|] \mid \mathsf{SA}'_{S'}[j] > \mathsf{SA}'_{S'}[i]\}$, else. We call an array $\mathsf{LNF}_{S'}[1..|S'|]$ a *sparse longest next factor* array w.r.t. S' iff for each

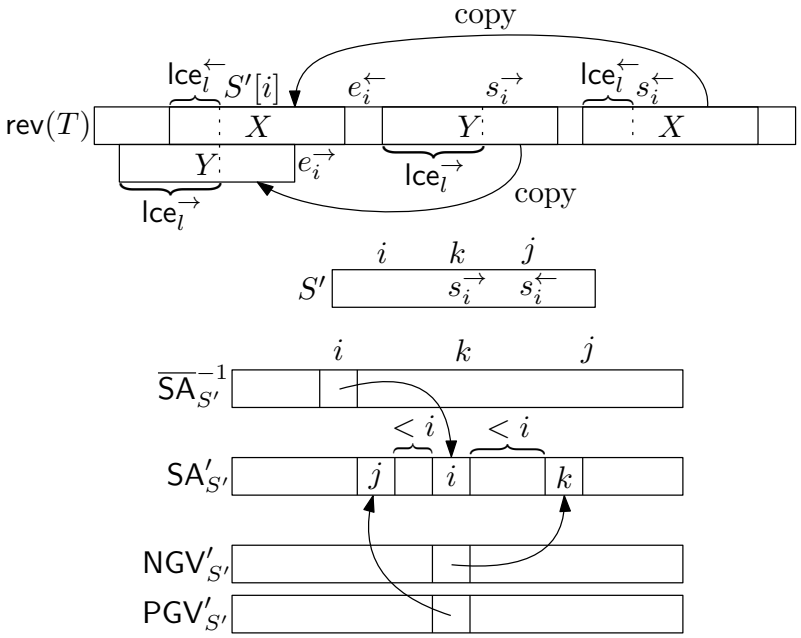


Figure 4.5: Illustration of Definition 4.3.5 and Observation 4.3.6.

$i \in [1, |S'|]$, either $\mathsf{LNF}_{S'}[i] = \bot$ if $\mathsf{rev}(T)[S'[i]]$ does not occur in $\mathsf{rev}(T)(S'[i], n]$, or $\mathsf{LNF}_{S'}[i] = \arg\max_{j \in (S'[i],n]}\{\mathsf{LCE}_{\mathsf{rev}(T)}(j, S'[i])\}$, else.

By the same argument as in Lemma 3.1.5, we can compute $\mathsf{LNF}_{S'}[i]$ in $\mathcal{O}(1)$ time using Theorem 2.8.4 (see Construction 4.3.4).

4.3.4 Construction. *Let $i \in [1, |S'|]$, let $i' = \overline{\mathsf{SA}}_{S'}^{-1}[i]$, let $j = \mathsf{PGV}'_{S'}[i']$ and let $k = \mathsf{NGV}'_{S'}[i']$. If $j = 0$ and $k = 0$, then $\mathsf{LNF}_{S'}[i'] = 0$. If either $j = 0$ or $k = 0$, then $\mathsf{LNF}_{S'}[i'] = S'[\mathsf{SA}'_{S'}[\mathsf{NGV}'_{S'}[i']]]$ or $\mathsf{LNF}'_{S'}[i] = S'[\mathsf{SA}'_{S'}[\mathsf{PGV}'_{S'}[i']]]$, respectively. Finally, if $j \neq 0$ and $k \neq 0$, then $\mathsf{LNF}'_{S'}[i] = S'[\mathsf{SA}'_{S'}[\mathsf{NGV}'_{S'}[i']]]$ if $\mathsf{LCE}_{\mathsf{rev}(T)}(S'[i'], S'[\mathsf{SA}'_{S'}[\mathsf{PGV}'_{S'}[i']]]) > \mathsf{LCE}_{\mathsf{rev}(T)}(S'[i'], S'[\mathsf{SA}'_{S'}[\mathsf{NGV}'_{S'}[i']]])$, or $\mathsf{LNF}_{S'}[i'] = S'[\mathsf{SA}'_{S'}[\mathsf{PGV}'_{S'}[i']]]$, else.*

In practice, we do not explicitly compute $\mathsf{LNF}_{S'}[i]$, but consider both of the phrases resulting from the sources $S'[\mathsf{SA}_{S'}[\mathsf{PGV}_{S'}[i']]]$ and $S'[\mathsf{SA}_{S'}[\mathsf{NGV}_{S'}[i']]]$. Additionally, we extend the resulting phrases to the left (see Figure 4.5).

4.3.5 Definition. Let $i \in [1, |S'|]$ and $i' = \overline{\mathsf{SA}}_{S'}^{-1}[i]$. If $\mathsf{PGV}'_{S'}[i'] \neq 0$, then let $s_i^{\leftarrow} = S'[\mathsf{SA}_{S'}[\mathsf{PGV}'_{S'}[i']]]$, $e_i^{\leftarrow} = S'[i] + \mathsf{LCE}_{\mathsf{rev}(T)}(s_i^{\leftarrow}, S'[i])$ and $\mathsf{lce}_l^{\leftarrow} = \mathsf{LCE}^{\mathsf{L}}_{\mathsf{rev}(T)}(S'[i], s_i^{\leftarrow})$. Similarly, if $\mathsf{NGV}'_{S'}[i'] \neq 0$, then let $s_i^{\rightarrow} = S'[\mathsf{SA}_{S'}[\mathsf{NGV}'_{S'}[i']]]$ and $e_i^{\rightarrow} = S'[i] + \mathsf{LCE}_{\mathsf{rev}(T)}(s_i^{\rightarrow}, S'[i])$, $\mathsf{lce}_l^{\rightarrow} = \mathsf{LCE}^{\mathsf{L}}_{\mathsf{rev}(T)}(S'[i], s_i^{\rightarrow})$.

4.3.6 Observation. Let $\hat{e}_i^{\leftarrow} = n - (S'[i] - \mathsf{lce}_l^{\leftarrow}) + 1$, $\hat{b}_i^{\leftarrow} = n - e_i^{\leftarrow} + 1$ and $\hat{s}_i^{\leftarrow} = n - (s_i^{\leftarrow} - \mathsf{lce}_l^{\leftarrow}) + 1$. Similarly, let $\hat{e}_i^{\rightarrow} = n - (S'[i] - \mathsf{lce}_l^{\rightarrow}) + 1$, $\hat{b}_i^{\rightarrow} = n - e_i^{\rightarrow} + 1$ and $\hat{s}_i^{\rightarrow} = n - (s_i^{\rightarrow} - \mathsf{lce}_l^{\rightarrow}) + 1$. Since $\mathsf{rev}(T)[b_i^{\leftarrow}, e_i^{\leftarrow}) = \mathsf{rev}(T)[s_i^{\leftarrow}, s_i^{\leftarrow} + e_i^{\leftarrow} - b_i^{\leftarrow})$ and $s_i^{\leftarrow} > b_i^{\leftarrow}$, we have $T[\hat{b}_i^{\leftarrow}, \hat{e}_i^{\leftarrow}) = T[\hat{s}_i^{\leftarrow}, \hat{s}_i^{\leftarrow} + \hat{e}_i^{\leftarrow} - \hat{b}_i^{\leftarrow})$ and $\hat{s}_i^{\leftarrow} < \hat{b}_i^{\leftarrow}$, hence we can use $\langle \hat{b}_i^{\leftarrow}, \hat{e}_i^{\leftarrow}, \hat{s}_i^{\leftarrow} \rangle$ as an LPF phrase tuple in T. Analogously, we can use $\langle \hat{b}_i^{\rightarrow}, \hat{e}_i^{\rightarrow}, \hat{s}_i^{\rightarrow} \rangle$ as a phrase tuple in T.

4.3.2.2 Algorithm

Our algorithm proceeds as follows. We begin by reversing T. Then, we compute a string synchronizing set S' for $\mathsf{rev}(T)$, $\mathsf{SA}'_{S'}$, $\overline{\mathsf{SA}}^{-1}_{S'}$, $\mathsf{PSV}'_{S'}$, and $\mathsf{NSV}'_{S'}$. Then, we compute for each $i \in [1, |S'|]$ the phrase tuples $\langle \hat{b}_i^{\leftarrow}, \hat{e}_i^{\leftarrow}, \hat{s}_i^{\leftarrow} \rangle$ and $\langle \hat{b}_i^{\rightarrow}, \hat{e}_i^{\rightarrow}, \hat{s}_i^{\rightarrow} \rangle$ as described in Observation 4.3.6 and append them to the array LPF.

However, we skip the computation of $\langle \hat{b}_i^{\leftarrow}, \hat{e}_i^{\leftarrow}, \hat{s}_i^{\leftarrow} \rangle$ if it is identical to the last computed phrase tuple $\langle \hat{b}_j^{\leftarrow}, \hat{e}_j^{\leftarrow}, \hat{s}_j^{\leftarrow} \rangle$, where $j \in [1, i)$ is maximal such that $\mathsf{PGV}_{S'}[\overline{\mathsf{SA}}^{-1}_{S'}[j]] \neq 0$. More precisely, if $b_i^{\leftarrow} \in [b_j^{\leftarrow}, e_j^{\leftarrow})$ and $s_i^{\leftarrow} - b_i^{\leftarrow} = s_j^{\leftarrow} - b_j^{\leftarrow}$, then $\langle b_i^{\leftarrow}, e_i^{\leftarrow}, s_i^{\leftarrow} \rangle = \langle b_j^{\leftarrow}, e_j^{\leftarrow}, s_j^{\leftarrow} \rangle$ and therefore $\langle \hat{b}_i^{\leftarrow}, \hat{e}_i^{\leftarrow}, \hat{s}_i^{\leftarrow} \rangle = \langle \hat{b}_j^{\leftarrow}, \hat{e}_j^{\leftarrow}, \hat{s}_j^{\leftarrow} \rangle$, hence we can skip computing $\langle \hat{b}_i^{\leftarrow}, \hat{e}_i^{\leftarrow}, \hat{s}_i^{\leftarrow} \rangle$. Similarly, we skip the computation of $\langle \hat{b}_i^{\rightarrow}, \hat{e}_i^{\rightarrow}, \hat{s}_i^{\rightarrow} \rangle$ if $b_i^{\rightarrow} \in [b_j^{\rightarrow}, e_j^{\rightarrow})$ and $b_i^{\rightarrow} - s_i^{\rightarrow} = b_j^{\rightarrow} - s_j^{\rightarrow}$, where $j \in [1, i)$ is maximal such that $\mathsf{NGV}_{S'}[\overline{\mathsf{SA}}^{-1}_{S'}[j]] \neq 0$.

Then, we reverse $\mathsf{rev}(T)$, build a string synchronizing set S for T, SA_S, SA_S^{-1}, PSV_S, and NSV_S, and use those to compute LPF phrases in the same way (without translating positions p in $\mathsf{rev}(T)$ to their corresponding positions $n - p + 1$ in T). We again append the computed phrases to the array LPF. Now, we sort the array $\mathsf{LPF}[1..N]$ such that $\forall i, j \in [1, N] : b_i < b_j \vee (b_i = b_j \wedge e_i > e_j)$. This enables us to remove duplicate phrases and cut overlapping phrases with one scan over $\mathsf{LPF}[1..N]$.

4.3.7 Example. In Example 2.7.3, we computed the 3-synchronizing set $S = \{1, 4, 14, 15, 16, 19\}$. Let $\mathsf{id}(i)$ be the lexicographical rank of $\mathsf{rev}(T)[i, i+\tau)$. Then, Theorem 2.7.2 yields $S' = \{2, 5, 8, 15, 16, 17, 20\}$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$\mathsf{rev}(T) =$	b	a	c	b	a	c	b	a	b	b	b	b	b	b	b	b	b	b	b	a	c	b	a	c	b	a
$\mathsf{id} =$	6	3	9	6	3	9	5	2	8	8	8	8	8	8	8	8	8	7	6	3	9	6	3	9	4	1

	1	2	3	4	5	6
$S =$	1	4	14	15	16	19
$\mathsf{SA}_S =$	2	6	1	5	4	3
$\mathsf{SA}_S^{-1} =$	3	1	6	5	4	2
$\mathsf{PSV}_S =$	0	1	0	3	3	3
$\mathsf{NSV}_S =$	3	3	0	5	6	0

	1	2	3	4	5	6	7
$S' =$	2	5	8	15	16	17	20
$\mathsf{SA}'_{S'} =$	3	2	7	1	6	5	4
$\overline{\mathsf{SA}}^{-1}_{S'} =$	4	2	1	7	6	5	3
$\mathsf{PGV}'_{S'} =$	0	1	0	3	2	5	6
$\mathsf{NGV}'_{S'} =$	3	3	0	5	0	0	0

Since $\mathsf{rev}(T)_x$ with $x = S'[\mathsf{SA}'_{S'}[\mathsf{NGV}'_{S'}[\overline{\mathsf{SA}}^{-1}_{S'}[1]]]] = 17$ yields no match with $\mathsf{rev}(T)_{S'[1]} = \mathsf{rev}(T)_2$, $\langle b_1^{\rightarrow}, e_1^{\rightarrow}, s_1^{\rightarrow} \rangle$ is undefined. However, $\mathsf{rev}(T)_x$ with $x = S'[\mathsf{SA}'_{S'}[\mathsf{PGV}'_{S'}[\overline{\mathsf{SA}}^{-1}_{S'}[1]]]] = 20$ yields a match of length 7 with $\mathsf{rev}(T)_2$, hence $\langle b_2^{\leftarrow}, e_2^{\leftarrow}, s_2^{\leftarrow} \rangle = \langle 2, 9, 20 \rangle$. Since $\mathsf{lce}_l^{\leftarrow} = 2$, we have $\langle \hat{b}_1^{\leftarrow}, \hat{e}_1^{\leftarrow}, \hat{s}_1^{\leftarrow} \rangle = \langle 19, 27, 1 \rangle$. The remaining computed LNF phrases

are $\langle \hat{b}_2^{\leftarrow}, \hat{e}_2^{\leftarrow}, \hat{s}_2^{\leftarrow} \rangle = \langle 22, 27, 19 \rangle$, $\langle \hat{b}_2^{\rightarrow}, \hat{e}_2^{\rightarrow}, \hat{s}_2^{\rightarrow} \rangle = \langle 19, 24, 4 \rangle$, $\langle \hat{b}_3^{\rightarrow}, \hat{e}_3^{\rightarrow}, \hat{s}_3^{\rightarrow} \rangle = \langle 19, 21, 7 \rangle$ and $\langle \hat{b}_4^{\leftarrow}, \hat{e}_4^{\leftarrow}, \hat{s}_4^{\leftarrow} \rangle = \langle 9, 19, 8 \rangle$. $\langle \hat{b}_5^{\leftarrow}, \hat{e}_5^{\leftarrow}, \hat{s}_5^{\leftarrow} \rangle$ is not computed, because $b_5^{\leftarrow} = 17 \in [16, 20) = [b_4^{\leftarrow}, e_4^{\leftarrow})$ and $s_5^{\leftarrow} - b_5^{\leftarrow} = 17 - 16 = 16 - 15 = s_4^{\leftarrow} - b_4^{\leftarrow}$ imply that $\langle \hat{b}_5^{\leftarrow}, \hat{e}_5^{\leftarrow}, \hat{s}_5^{\leftarrow} \rangle = \langle 9, 19, 8 \rangle$ is identical to $\langle \hat{b}_4^{\leftarrow}, \hat{e}_4^{\leftarrow}, \hat{s}_4^{\leftarrow} \rangle = \langle 9, 19, 8 \rangle$.

The algorithm computes the LPF phrases $\langle 4, 9, 1 \rangle$,$\langle 9, 19, 8 \rangle$ and $\langle 19, 27, 1 \rangle$ like the algorithm from Section 4.3.1. After sorting LPF, we have

$$\mathsf{LPF} = [\langle 4, 9, 1 \rangle, \langle 9, 19, 8 \rangle, \underline{\langle 9, 19, 8 \rangle}, \langle 19, 27, 1 \rangle, \underline{\langle 19, 27, 1 \rangle}, \langle 19, 24, 4 \rangle, \langle 19, 21, 7 \rangle, \langle 22, 27, 19 \rangle].$$

Duplicate phrases are underlined. Finally, the algorithm chooses the phrases $\langle 4, 9, 1 \rangle$, $\langle 9, 19, 8 \rangle$ and $\langle 19, 27, 1 \rangle$. No phrase is cut.

4.4 Factorizing Gaps

Now, we discuss the case where we want to compress the gaps using factors. In practice, the rolling hash index $\mathcal{H}$ described in Section 4.2.1.3 provided the best compression ratio and throughput. Another advantage of it compared with the other indexes is that its size is predictable and independent of the chosen pattern lengths.

4.4.1 Tuning the Rolling Hash Index in Practice

To choose its size and the pattern lengths, we at first compute the following statistics with one scan over $\mathsf{LPF}[1..N]$.

$$g = b_1 + \sum_{i=1}^{N} b_{i+1} - e_i \tag{4.1}$$

$$\overline{\mathsf{gap}} = g/N \tag{4.2}$$

$$\overline{\mathsf{lpf}} = \frac{\sum_{i=1}^{N} e_i - b_i}{N} \tag{4.3}$$

$$\rho = \min(\overline{\mathsf{gap}}, \overline{\mathsf{lpf}}, 8 \cdot 128^{1-g/n}) \tag{4.4}$$

$$L = \begin{cases} \{2,3,4,5,6\} & \text{if } \rho \in (0,6] \\ \{2,3,4,6,8\} & \text{if } \rho \in (6,8] \\ \{2,3,4,8,12\} & \text{if } \rho \in (8,12] \\ \{2,4,6,9,16\} & \text{if } \rho \in (12,16] \\ \{2,4,6,10,20\} & \text{if } \rho \in (16,32] \\ \{2,4,7,12,28\} & \text{if } \rho \in (32,64] \\ \{2,4,8,16,36\} & \text{if } \rho \in (64,128] \\ \{2,5,10,20,42\} & \text{if } \rho \in (128,256] \\ \{2,6,12,24,48\} & \text{if } \rho \in (256,512] \\ \{2,8,16,32,64\} & \text{if } \rho \in (512,\infty] \end{cases} \tag{4.5}$$

g is the length of the gaps, $\overline{\mathsf{gap}}$ is the average gap length and $\overline{\mathsf{lpf}}$ is the average LPF phrase length. Recall that we have $|\mathcal{H}| = 2^h$. We choose a target size $t = \max(n/12, g/3)$ for $\mathcal{H}$. Then, we choose h such that the deviation $||\mathcal{H}| - t| = |2^h - t|$ of $|\mathcal{H}|$ from its target size t is minimal. We choose 5 pattern lengths $L = \{l_1, l_2, l_3, l_4, l_5\}$ (see Equation (4.5)) for $\mathcal{H}$ based on a pattern length guess ρ (see Equation (4.4)).

We chose ρ in this way because when factorizing the gaps, we will rarely need to compute phrases that are longer than $\overline{\mathsf{gap}}$. It is also unlikely that we will find a previous occurrence of a substring within a gap with length $\geqslant \overline{\mathsf{lpf}}$, because else, this substring would likely be part of some LPF phrase. Finally, it is possible that T is unrepetitive, but there are only few and long LPF phrases. In this case, $\min(\overline{\mathsf{gap}}, \overline{\mathsf{lpf}})$ becomes too large. However, since in this case, the relative length $1 - g/n$ of the LPF phrases is small, limiting ρ to $8 \cdot 128^{1-g/n}$ mitigates the issue.

4.4.2 Rolling Hash Index Interface

With $\mathcal{H}$, we store the current position $p \in [1, n]$, the chosen pattern lengths $L = \{l_1, ..., l_M\}$, and the fingerprint $\Phi_i = \phi(p, p + l_i - 1)$ at position p, for each $i \in [1, M]$. $\mathcal{H}$ provides the following interface.

- $\mathcal{H}.\mathsf{pos}()$ returns the current position p ($\mathcal{O}(1)$ time).
- $\mathcal{H}.\mathsf{init}(j)$ sets $p \leftarrow j$ and recomputes $\Phi_i \leftarrow \phi(p, p + l_i - 1)$ for each $i \in [1, M]$ ($\mathcal{O}(\sum_{i=1}^{M} l_i)$ time).
- $\mathcal{H}.\mathsf{roll}(i)$ rolls Φ_i such that $\Phi_i = \phi(p + 1, p + l_i)$ ($\mathcal{O}(1)$ time).
- $\mathcal{H}.\mathsf{roll}()$ calls $\mathcal{H}.\mathsf{roll}(i)$ for each $i \in [1, M]$ and then increments $p \leftarrow p + 1$ to maintain $\forall i \in [1, M] : \Phi_i = \phi(p, p + l_i - 1)$ ($\mathcal{O}(M)$ time).
- $\mathcal{H}.\mathsf{advance}(i)$ sets $\mathcal{H}[\Phi_i \circledwedge \mathsf{mask} + 1] \leftarrow p$, and calls $\mathcal{H}.\mathsf{roll}(i)$ ($\mathcal{O}(1)$ time).
- $\mathcal{H}.\mathsf{advance}()$ calls $\mathcal{H}.\mathsf{advance}(i)$ for each $i \in [1, M]$, and then increments $p \leftarrow p + 1$ to maintain $\Phi_i = \phi(p, p + l_i - 1)$ ($\mathcal{O}(M)$ time).
- $\mathcal{H}.\mathsf{prev\text{-}occ}(l_{\mathsf{max}})$ (see Algorithm 13) returns an approximate LZ factor $\langle s, l \rangle$ with $l \leqslant l_{\mathsf{max}}$ for position p; has the same effect as $\mathcal{H}.\mathsf{advance}()$ ($\mathcal{O}(M)$ time).

In Algorithm 13, we start by initializing $\langle s, l \rangle = \langle 0, T[p] \rangle$ such that we return a literal factor if we do not find any previous occurrence. Then, we consider each pattern length l_i in L (starting with the longest) and try to find a previous occurrence s' of $T[p, p + l_i)$ using $\mathcal{H}$ and the fingerprint Φ_i of $T[p, p + l_i)$ (see line 4). Then, we update $\mathcal{H}$ as in $\mathcal{H}.\mathsf{advance}(i)$ (see line 5). If $s' \neq \bot$, and $l' = \min(\mathsf{LCE}(p, s'), l_{\mathsf{max}}) \neq 0$, then $\langle s', l' \rangle$ is a referencing LZ factor for position p, and we set $\langle s, l \rangle \leftarrow \langle s', l' \rangle$ in line 9. As soon as we have found such a referencing LZ factor, we will call $\mathcal{H}.\mathsf{advance}(j)$ in line 11 for each remaining pattern length index $j \in [1, i)$ and finally return $\langle s', l' \rangle$ in line 12.

Algorithm 13: prev-occ(l_{max})

```
1  ⟨s, l⟩ ← ⟨T[p], 0⟩;
2  for i from M down to 1 do
3      if l = 0 then
4          s' ← H[Φ_i ⊙ mask + 1];
5          H.advance(i);
6          if s' ≠ ⊥ then
7              l' ← min(LCE(p, s'), l_max);
8              if l' ≠ 0 then
9                  ⟨s, l⟩ ← ⟨s', l'⟩;
10     else
11         H.advance(i);
12 p ← p + 1;
13 return ⟨s, l⟩;
```

4.4.3 Naive Algorithm

Now, we discuss how we use the rolling hash index to factorize the gaps (see Algorithm 14). We maintain the current position $i \in [1, n]$ in T and the index j of the next LPF phrase starting at or after i. At first, we initialize the rolling hash index $\mathcal{H}$ to position $i = 1$ (see line 2).

At the start of each iteration, we advance with $\mathcal{H}$ up to position i (see lines 4-5). Then, we factorize the gap $T[i, \mathsf{LPF}[j].b)$ by iteratively calling prev-occ($\mathsf{LPF}[j].b - i$), outputting the factor it returns, and advancing with $\mathcal{H}$ to the end of this factor (see lines 7-11). If now $i = n + 1$ holds, then we have completely factorized T and return (see lines 12-13). Else, we output either the next (j-th) LPF factor $\langle s, l\rangle$ (line 14) or the factor $\langle s', l'\rangle$ returned by prev-occ($\mathsf{LPF}[j + 1].b - i$) if it is longer, i.e, $l' > l$ (see line 17). Note that $l' > l$ can hold because each computed LPF phrase (for some sample $S[i]$) is only then perfect if $T[S[i], S[i] + 2\tau)$ has a previous occurrence. We still consider potentially non-perfect LPF phrases, because it improves the compression ratio in practice. Finally, we find the next LPF phrase starting at or after i (see lines 20-21).

4.4.1 Example. Suppose we compute the LPF phrases naively as in the 3-Approximation (see Section 3.1.2). This yields $\mathsf{LPF} = [\langle 4, 9, 1\rangle, \langle 15, 19, 14\rangle, \langle 19, 27, 1\rangle]$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b

Algorithm 14: factorize-gaps-naive()

```
1  i ← 1; j ← 1;
2  H.init(1);
3  while true do
4      while H.pos() < i do
5          H.advance();
6      while i < LPF[j].b do
7          ⟨s, l⟩ ← prev-occ(LPF[j].b − i);
8          output ⟨s, l⟩;
9          i ← i + max(1, l);
10         while H.pos() < i do
11             H.advance();
12     if i = n + 1 then
13         return;
14     ⟨s, l⟩ ← ⟨LPF[j].s, LPF[j].e − LPF[j].b⟩;
15     ⟨s′, l′⟩ ← prev-occ(LPF[j + 1].b − i);
16     if l′ > l then
17         ⟨s, l⟩ ← ⟨s′, l′⟩;
18     output ⟨s, l⟩;
19     i ← i + max(1, l);
20     while LPF[j].b < i do
21         j ← j + 1;
```

We have $g = 9$, $\overline{\mathsf{gap}} = g/N = 9/3 = 3$, $\overline{\mathsf{lpf}} = \frac{5+4+8}{3} = 5.\overline{6}$ and $\rho = \min(3, 5.\overline{6}, 8 \cdot 128^{1-9/26}) = \min(3, 5.\overline{6}, 190.89) = 3$, hence $L = \{2, 3, 4, 5, 6\}$. It holds $t = \max(26/12, 6/3) = \max(2.1\overline{6}, 2) = 2.1\overline{6}$, so $h = 1$ minimizes the deviation $||2^h| - t| = 0.1\overline{6}$. Finally, we have $\mathsf{mask} = 2^h - 1 = 1$. For simplicity, we use $L = \{2, 3\}$ ($M = 2$) and set the base for the Karp-Rabin fingerprints to $\sigma + 1 = 4$ in this example. Let $a = 1$, $b = 2$ and $c = 3$.

Initialization. We start by initializing $\mathcal{H}$ to position 1 in line 2, which sets $p = 1$, $\Phi_1 = \phi(1, 2) = 4^1 \cdot a + 4^0 \cdot b = 4 + 2 = 6$ and $\Phi_2 = \phi(1, 3) = 4^2 \cdot a + 4^1 \cdot b + 4^0 \cdot c = 8 + 8 + 3 = 19$.

Iteration 1. In the first iteration, we have $i = 1 < 4 = \mathsf{LPF}[j].b$ (see line 6), so we call prev-occ(3) in line 7. There, we initialize $s = T[p] = a$ and $l = 0$. For $i = M = 2$, we get $s' = \mathcal{H}[\Phi_2 \circledcirc \mathsf{mask} + 1] = \mathcal{H}[19 \bmod 2^h + 1] = \mathcal{H}[1 + 1] = \perp$ and set $\mathcal{H}[2] \leftarrow p = 1$ and $\Phi_1 \leftarrow \phi(2, 3) = 8 + 3 = 11$. For $i = 1$, we get $s' = \mathcal{H}[\Phi_1 \circledcirc \mathsf{mask}] = \mathcal{H}[6 \bmod 2^h + 1] = \mathcal{H}[0 + 1] = \perp$ and set $\mathcal{H}[1] \leftarrow p = 1$ and $\Phi_2 \leftarrow \phi(2, 4) = 16 + 12 + 1 = 29$. Now, we increment p by 1 and return and output the literal factor $\langle a, 0 \rangle$ in line 8 of Algorithm 14.

After setting $i \leftarrow 2$ in line 9, we call $\mathsf{prev\text{-}occ}(2)$ in line 7, and output the literal factor $\langle b, 0\rangle$. Finally, we set $i \leftarrow 3$, call $\mathsf{prev\text{-}occ}(3)$, output $\langle c, 0\rangle$ and set $i \leftarrow 4$. Now, we have $\mathcal{H} = [3, 3]$. Since $i = 4 \not< 4 = \mathsf{LPF}[1].b$ in line 6, we exit the **while**-loop (lines 6-11). In line 14, we set $\langle s, l\rangle = \langle 4, 5\rangle$. Since $\mathcal{H} = [3, 3]$ and $T[i] = b \neq T[3]$, $\mathsf{prev\text{-}occ}(15 - 4)$ (line 15) returns a literal factor $\langle s', l'\rangle = \langle a, 0\rangle$, and we output $\langle 4, 5\rangle$ in line 18. Finally, we set $i \leftarrow 9$ and $j \leftarrow 2$.

Iteration 2. In the second iteration, we call $\mathcal{H}.\mathsf{advance}()$ $i - p = 9 - 5 = 4$ times in line 5. Now, it holds $\mathcal{H} = [8, 6]$ and $p = i = 9$. In line 7, we call $\mathsf{prev\text{-}occ}(15 - 9)$, which returns $\langle 8, 6\rangle$, because for $i = 2$, we have $\Phi_i = \phi(9, 11) = 42$ and $s' = \mathcal{H}[\Phi_i \circledcirc \mathsf{mask}] = \mathcal{H}[42 \bmod 2^h + 1] = \mathcal{H}[0 + 1] = 8$ and $l' = \min(\mathsf{LCE}(p, s'), l_{\mathsf{max}}) = \min(\mathsf{LCE}(9, 8), 6) = 6$. After line 15, it holds $\langle s, l\rangle = \langle s', l'\rangle = \langle 14, 4\rangle$, and we output $\langle 14, 4\rangle$.

Iteration 3. At the beginning of the last iteration, we have $i = 19$, $p = 15$ and $j = 3$. After calling $\mathcal{H}.\mathsf{advance}()$ $i - p = 4$ times in lines 5, we output the final factor $\langle 1, 8\rangle$ in line 18.

4.4.4 Optimized Algorithm

Algorithm 15 shows our optimized gap factorization algorithm. It builds upon Algorithm 14 in two ways. The first is that we do not roll over LPF phrases. Intuitively, this does not reduce the compression ratio, because the gaps do not contain substrings that occur within LPF phrases.

If we have $\mathcal{H}.\mathsf{pos}() < i$ in line 5, then $T[\mathcal{H}.\mathsf{pos}(), i)$ is part of an LPF phrase, hence we want to skip it. To do so, we can call $\mathcal{H}.\mathsf{init}(i)$. However, it is possible that the cost $\theta = \sum_{j=1}^{M} l_j/M$ per pattern length for calling $\mathcal{H}.\mathsf{init}(i)$ is larger than the cost $i - \mathcal{H}.\mathsf{pos}()$ per pattern length for rolling up to position i with $\mathcal{H}$. In line 6, we check which one of these two options is cheaper.

Recall that in line 7 and 15 of Algorithm 14, we limit the length of the computed phrases such that they end at or before the start of the respective next LPF phrases (as in Algorithm 8). Since the computed LPF phrases are not necessarily perfect, it is possible that the factors output by prev-occ in lines 12 and 32 reach beyond multiple of the following LPF phrases. Then, we possibly have to skip some LPF phrases (see lines 14-23 and 37-38). If the computed phrase reaches into the next LPF phrase ($i > \mathsf{LPF}[j].b$, see line 14), then we check if it ends within this LPF phrase ($i \leqslant \mathsf{LPF}[j].e$, see line 14). If this is the case, then we cut it in lines 16-17 such that it ends at $\mathsf{LPF}[b].j$. Else ($i > \mathsf{LPF}[j].e$), then we skip over the following LPF phrases until i lies within the next (j-th) LPF phrase, i.e, it holds $i < \mathsf{LPF}[j].e$ (see lines 19-20). In any case, we roll with $\mathcal{H}$ over the just computed phrase (see lines 21-23 and 25-26).

Since we do not limit the phrase length in $\mathsf{prev\text{-}occ}(n)$, we generally have $i \in [\mathsf{LPF}[j].b, \mathsf{LPF}[j].e)$ in line 29, so that we may have to shorten the j-th LPF phrase from the left by

Algorithm 15: factorize-gaps()

```
1  i ← 1; j ← 1;
2  H.init(1);
3  while true do
4      if i < LPF[j].b then
5          if H.pos() < i then
6              if i − H.pos() ⩽ θ then
7                  while H.pos() < i do
8                      H.roll();
9              else
10                 H.init(i);
11         while i < LPF[j].b do
12             ⟨s, l⟩ ← prev-occ(n);
13             i ← i + max(1, l);
14             if i > LPF[j].b then
15                 if i ⩽ LPF[j].e then
16                     l ← l − (i − LPF[j].b);
17                     i ← LPF[j].b;
18                 else
19                     while i ⩾ LPF[j].e do
20                         j ← j + 1;
21             do
22                 H.advance();
23             while H.pos() < LPF[j].b;
24         output ⟨s, l⟩;
25         while H.pos() < i do
26             H.advance();
27     if i = n + 1 then
28         return;
29     x ← i − LPF[j].b;
30     ⟨s, l⟩ ← ⟨LPF[j].s + x,
         LPF[j].e − LPF[j].b − x⟩;
31     if H.pos() = LPF[j].b then
32         ⟨s′, l′⟩ ← prev-occ(n);
33         if l′ > l then
34             ⟨s, l⟩ ← ⟨s′, l′⟩;
35     output ⟨s, l⟩;
36     i ← i + max(1, l);
37     while LPF[j].b < i do
38         j ← j + 1;
```

the excess $x = i - \mathsf{LPF}[j].b$ of the last computed phrase (see lines 29-30). Due to the fact that we do not roll over LPF phrases with $\mathcal{H}$, it is possible that $\mathcal{H}.\mathsf{pos}() < i$ holds in line 31. However, the factor output by $\mathsf{prev\text{-}occ}(n)$ is only valid if $\mathcal{H}.\mathsf{pos}() = i$ holds, hence we would have to call $\mathcal{H}.\mathsf{init}(i)$ beforehand, which is expensive considering the fact that the resulting factor is likely shorter than the LPF phrase. Therefore, we only consider it only if $\mathcal{H}.\mathsf{pos}() = i$ holds (see lines 31-34).

4.4.2 Example. If we run this algorithm on the text from Example 4.4.1, then we get the exact LZ77 factorization, because compared with Algorithm 14, in iteration 2, the length of the factor $\langle 8, 6\rangle$ is not limited to $l_{\mathsf{max}} = 6$, i.e, we instead output $\langle 8, 10\rangle$ and skip the LPF phrase $\langle 15, 19, 14\rangle$.

4.5 LZ77 Exact Algorithm

The running time (see Theorem 3.2.10) of the exact LZ77 Algorithm described in Section 3.2 is dominated by the insertions into the data structure for insertion-only orthogonal range one reporting (see Definition 3.2.1). In practice, however, the running time is dominated by the computation of sparse prefix- and suffix array intervals, and orthogonal range queries.

Therefore, we focus on optimizing those aspects. In Section 4.5.1, we improve the practical running time of sparse prefix- and suffix array interval searches. In Section 4.5.2, we discuss a data structure for quickly computing the Karp-Rabin fingerprint of any substring of T, which we will use in Section 4.5.3. There, we aim to speed up sparse prefix- and suffix array interval searches by sampling all intervals for a set of pattern lengths. In Section 4.5.4, we present practical data structures for static and dynamic (insertion-only) orthogonal range one-reporting. Finally, we employ the discussed optimizations in the exact LZ77 algorithm (see Section 4.5.5).

4.5.1 Sparse Prefix- and Suffix Array Interval Search

In Lemma 3.2.5, we discussed how to find the sparse prefix and suffix array intervals of a substring of T. In practice, the algorithm can be improved in several ways (see Algorithm 33).

The algorithm $\mathsf{ssa\text{-}interval}([i, i+l), [b, e])$ computes $[y_1, y_2] = \mathsf{siv}_S(T[i, i+l))$, and requires $[y_1, y_2] \subseteq [b, e]$. If we have $[b', e'] = \mathsf{siv}_S(T[i, i+l'))$ with $l' < l$ at hand, then we can call $\mathsf{ssa\text{-}interval}([i, i+l), [b', e'])$. Instead of performing a binary search over $[1, |S|]$ to find y_1 as in Algorithm 9, $\mathsf{ssa\text{-}interval}([i, i+l), [b, e])$ performs a binary search over $[b, e]$ to find y_1 and uses each lexicographical comparison during this search to iteratively shrink an interval $[\hat{b}, \hat{e}] \subseteq [b, e]$ such that $y_2 \in [\hat{b}, \hat{e}]$. Then, it performs a binary search over $[\hat{b}, \hat{e}]$ to find y_2.

We start with $[\hat{b}, \hat{e}] = [b, e]$. Compared with Algorithm 9, we additionally maintain $\mathsf{lce}_b = \mathsf{LCE}(S[\mathsf{SA}_S[b]], i)$, $\mathsf{lce}_e = \mathsf{LCE}(S[\mathsf{SA}_S[e]], i)$, $\mathsf{lce}_{\hat{b}} = \mathsf{LCE}(S[\mathsf{SA}_S[\hat{b}]], i)$ and $\mathsf{lce}_{\hat{e}} = \mathsf{LCE}(S[\mathsf{SA}_S[\hat{e}]], i)$. Let $[b, e]$ be the current search interval for y_1. Then, we compute $\mathsf{lce}_m = \mathsf{LCE}(S[\mathsf{SA}_S[m]], i)$ for the candidate position $m = \lceil (b+e)/2 \rceil$ by

$$\mathsf{lce}_m = \mathsf{LCE}(S[\mathsf{SA}_S[m]], i) = \mathsf{lce}_{\mathsf{min}} + \mathsf{LCE}(S[\mathsf{SA}_S[m]] + \mathsf{lce}_{\mathsf{min}}, i + \mathsf{lce}_{\mathsf{min}}), \tag{4.6}$$

where $\mathsf{lce}_{\mathsf{min}} = \min(\mathsf{lce}_b, \mathsf{lce}_e)$. This reduces the running time of short LCE queries, because it rises with the computed LCE values. Now, we consider 3 cases (similar to Figure 3.6).

- **Case 1**: $\mathsf{lce}_m \geqslant l$. Then, $y_1 \in [b, m]$ and $y_2 \in [m, e]$, hence we set $e \leftarrow m$ and $\mathsf{lce}_e \leftarrow \mathsf{lce}_m$. If $m > \hat{b}$, then we set $\hat{b} \leftarrow m$ and $\mathsf{lce}_{\hat{b}} \leftarrow \mathsf{lce}_m$.
- **Case 2**: $\mathsf{lce}_m < l$ and $T_{S[\mathsf{SA}_S[m]]} \prec T_i$. Then, $y_1 \in [m, e]$ and $y_2 \in [m, e]$, hence we set $b \leftarrow m$ and $\mathsf{lce}_b \leftarrow \mathsf{lce}_m$. If $m > \hat{b}$, then we set $\hat{b} \leftarrow m$ and $\mathsf{lce}_{\hat{b}} \leftarrow \mathsf{lce}_m$.

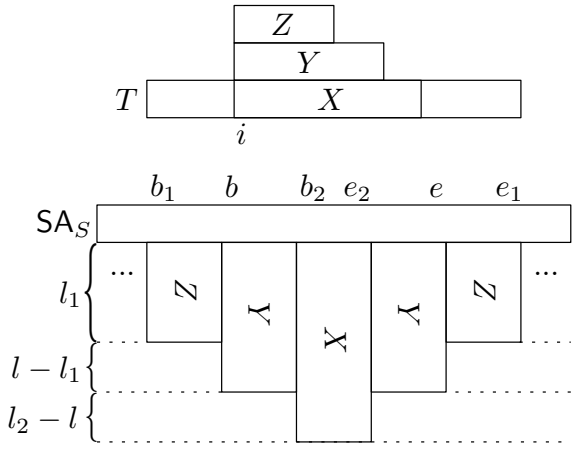


Figure 4.6: Illustration of Observation 4.5.1.

- **Case 3**: $\mathsf{lce}_m < l$ and $T_{S[\mathsf{SA}_S[m]]} \succ T_i$. Then, $y_1 \in [b, m]$ and $y_2 \in [b, m]$, hence we set $e \leftarrow m$ and $\mathsf{lce}_e \leftarrow \mathsf{lce}_m$. If $m < \hat{e}$, then we set $\hat{e} \leftarrow m$ and $\mathsf{lce}_{\hat{e}} \leftarrow \mathsf{lce}_m$.

We stop the binary search as soon as $|[b, e]| = 2$. If $\max(\mathsf{lce}_b, \mathsf{lce}_e) < l$, then $[y_1, y_2] = \varnothing$, and we return $\varnothing$. Else, if $\mathsf{lce}_b \geqslant l$, then $y_1 = b$, and $y_1 = e$, else.

Now, we compute y_2 using a binary search over $[\hat{b}, \hat{e}]$. We set $[b, e] \leftarrow [\hat{b}, \hat{e}]$, $\mathsf{lce}_b \leftarrow \mathsf{lce}_{\hat{b}}$ and $\mathsf{lce}_e \leftarrow \mathsf{lce}_{\hat{e}}$. For a candidate position $m \in [b, e]$, we again compute $\mathsf{lce}_m = \mathsf{LCE}(S[\mathsf{SA}_S[m]], i)$ according to Equation (4.6). However, since $\mathsf{lce}_b \geqslant l$ always holds, we have to consider only 2 cases.

- **Case 1**: $\mathsf{lce}_m \geqslant l$. Then, $y_2 \in [m, e]$, hence we set $b \leftarrow m$ and $\mathsf{lce}_b \leftarrow \mathsf{lce}_m$.
- **Case 2**: $\mathsf{lce}_m < l$. Then, $y_2 \in [b, m]$, hence we set $e \leftarrow m$ and $\mathsf{lce}_e \leftarrow \mathsf{lce}_m$.

We stop the binary search as soon as $|[b, e]| = 2$. If $\mathsf{lce}_e \geqslant l$, then $y_2 = e$, and $y_2 = b$, else.

The algorithm $\mathsf{spa\text{-}interval}([i, i+l), [b, e])$ analogously computes $\mathsf{piv}_S(T[i, i+l)) \subseteq [b, e]$.

4.5.1.1 Interpolating Intervals

In the exact LZ77 algorithm (see Theorem 3.2.9), we perform exponential searches to find the maximum possible length l, such that we can combine the tail $T[i, i+l)$ with the head.

4.5.1 Observation. Let $[b, e] = \mathsf{siv}_S(T[i, i+l))$, $[b_1, e_1] = \mathsf{siv}_S(T[i, i+l_1))$ and $[b_2, e_2] = \mathsf{siv}_S(T[i, i+l_2))$ with $l_1 \leqslant l \leqslant l_2$. Then, $b_1 \leqslant b \leqslant b_2$ and $e_2 \leqslant e \leqslant e_1$ (see Figure 4.6). This implies that we can compute b and e using binary searches over $[b_1, b_2]$ and $[e_2, e_1]$, respectively.

The algorithm $\mathsf{interpolate\text{-}ssa}([i, i+l), \langle [b_1, e_1], l_1\rangle, \langle [b_2, e_2], l_2\rangle)$ (see Algorithm 32) uses Observation 4.5.1 to compute $\mathsf{siv}_S(T[i, i+l))$. It uses the same optimization as Algorithm 33 to reduce the running time of the LCE queries. The algorithm $\mathsf{interpolate\text{-}spa}([i, i+l), \langle [b_1, e_1], l_1\rangle, \langle [b_2, e_2], l_2\rangle)$ works analogously for sparse prefix array intervals.

4.5.2 Karp-Rabin Fingerprint Sampling

In order to speed up the computation of sparse prefix- and suffix array intervals, we will sample all intervals for a set of pattern lengths. To this end, we need a data structure that allows us to quickly compute the Karp-Rabin fingerprint of any substring of T. In this section, we discuss how we can achieve this.

4.5.2.1 Data Structure for Combining Fingerprints in Constant Time

Since we will only sample a subset of all possible fingerprints, we need a data structure that allows us to quickly combine the fingerprints of two substrings.

4.5.2 Lemma. *We can construct in $\mathcal{O}(\sqrt{n})$ time and space a data structure of size $\mathcal{O}(\sqrt{n})$ that allows us to compute $\phi(x, z)$ given $\phi(x, y)$ and $\phi(y + 1, z)$ in $\mathcal{O}(1)$ time.*

Proof. With Definition 2.6.1, we have

$$\begin{aligned}
\phi(x,z) &= \left(\sum_{k=x}^{z} T[k]\cdot b^{z-k}\right) \bmod q \\
&= \left(\sum_{k=x}^{y} T[k]\cdot b^{z-k} + \sum_{k=y+1}^{z} T[k]\cdot b^{z-k}\right) \bmod q \\
&= \left(b^{z-y}\cdot\left(\sum_{k=x}^{y} T[k]\cdot b^{y-k}\right) \bmod q + \left(\sum_{k=y+1}^{z} T[k]\cdot b^{z-k}\right) \bmod q\right) \bmod q \\
&= \big(b^{z-y}\cdot\phi(x,y) + \phi(y+1,z)\big) \bmod q \\
&= \big(\big(b^{z-y} \bmod q\big)\cdot\phi(x,y) + \phi(y+1,z)\big) \bmod q,
\end{aligned} \tag{4.7}$$

where the mod operation that we added in the last equality prevents an integer overflow (we use 128-bit registers). Now it remains to compute $b^{z-y} \bmod q$ in $\mathcal{O}(1)$ time.

We store two arrays $\mathcal{X}[1..\lceil\sqrt{n}\rceil]$ and $\mathcal{Y}[1..\lceil\sqrt{n}\rceil]$, where $\mathcal{X}[i] = b^i \bmod q$ and $\mathcal{Y}[i] = b^{i\lceil\sqrt{n}\rceil} \bmod q$. Suppose we want to compute $b^i \bmod q$, where $i \in [1, n]$, and let $y = \lfloor i/\lceil\sqrt{n}\rceil\rfloor$ and $x = i - y\lceil\sqrt{n}\rceil$. Then, we have $b^i = b^{y\lceil\sqrt{n}\rceil}\cdot b^x$ and $i, y \in [1, \lceil\sqrt{n}\rceil]$, hence we can compute

$$b^i \bmod q = ((b^{y\lceil\sqrt{n}\rceil} \bmod q)(b^x \bmod q)) \bmod q = (\mathcal{X}[y]\cdot\mathcal{Y}[x]) \bmod q, \tag{4.8}$$

which takes $\mathcal{O}(1)$ time. $\mathcal{X}$ and $\mathcal{Y}$ can be constructed in $\mathcal{O}(\sqrt{n})$ time and space in one scan (see Algorithm 16).

Algorithm 17 summarizes how we combine the fingerprints f_l and f_r of a left and a right substring, where l_r is the length of the right substring. In lines 1-3, we compute $b^{l_\mathsf{r}} \bmod q$ according to Equation (4.8), and in line 4, we return the combined fingerprint according to Equation (4.7). □

Algorithm 16: build-X-Y()

1 $\mathcal{X}[0] \leftarrow 1; \mathcal{X}[1] \leftarrow b;$
2 **for** i **from** 2 **to** $\lceil\sqrt{n}\rceil$ **do**
3 $\quad \mathcal{X}[i] \leftarrow (\mathcal{X}[i-1] \cdot b) \bmod q;$
4 $\mathcal{Y}[0] \leftarrow 1; \mathcal{Y}[1] \leftarrow \mathcal{X}[\lceil\sqrt{n}\rceil];$
5 **for** i **from** 2 **to** $\lceil\sqrt{n}\rceil$ **do**
6 $\quad \mathcal{Y}[i] \leftarrow (\mathcal{Y}[i-1] \cdot \mathcal{Y}[1]) \bmod q;$

Algorithm 17: concat-fps($f_\mathsf{l}, f_\mathsf{r}, l_\mathsf{r}$)

1 $y \leftarrow \lceil l_\mathsf{r}/\lceil\sqrt{n}\rceil\rceil;$
2 $x \leftarrow l_\mathsf{r} - y\lceil\sqrt{n}\rceil;$
3 $b' \leftarrow (\mathcal{X}[x] \cdot \mathcal{Y}[y]) \bmod q;$
4 **return** $(f_\mathsf{l} \cdot b' + f_\mathsf{r}) \bmod q;$

4.5.2.2 Main Data Structure

With Lemma 4.5.2, we can now discuss the main data structure.

4.5.3 Theorem. *Let $s \geqslant 1$ be an integer. Then, we can construct in $\mathcal{O}(n)$ time and $\mathcal{O}(n/s + \sqrt{n})$ space a data structure of size $\mathcal{O}(n/s + \sqrt{n})$ that allows us to compute the Karp–Rabin fingerprint of any length-l substring of T in $\mathcal{O}(\min(l, \log(l/s) + s))$ time.*

Proof. **Data Structures.** We use the data structure from Lemma 4.5.2. Additionally, we store a binary tree of height $h_\mathsf{max} = \lceil\log_2\lceil n/s\rceil\rceil + 1$, where

1. for each $h \in [1, h_\mathsf{max}]$, the array $\mathcal{F}[h][1..w_h]$ stores the $w_h = \lceil\lceil n/s\rceil/2^{h-1}\rceil$ nodes at height h, i.e, $\mathcal{F}[h][i]$ with $i \in [1, w_h]$ represents the i-th node (from the left) at height h,
2. each leaf $\mathcal{F}[1][i]$ with $i \in [1, w_1]$ stores $\phi((i-1) \cdot s + 1, \min(n, i \cdot s))$,
3. for each $h \in [2, h_\mathsf{max}]$ and $i \in [1, w_h]$,
   (a) the internal node $\mathcal{F}[h][i]$ has left child $\mathcal{F}[h-1][2i-1]$, and
   (b) if $2i \leqslant w_{h-1}$, then
      i. $\mathcal{F}[h][i]$ has right child $\mathcal{F}[h-1][2i]$, and
      ii. $\mathcal{F}[h][i] = \phi(x, z)$, where $\mathcal{F}[h-1][2i-1] = \phi(x, y)$ and $\mathcal{F}[h-1][2i] = \phi(y+1, z)$,
   (c) and else ($2i > w_{h-1}$), then $\mathcal{F}[h][i] = \mathcal{F}[h-1][2i-1]$ has no right child.

More generally, for each $h \in [1, h_\mathsf{max}]$ and $i \in [1, w_h]$, we have $\mathcal{F}[h][i] = \phi((i-1) \cdot s \cdot 2^h + 1, \min(n, i \cdot s \cdot 2^h))$.

Construction. Algorithm 18 shows the construction. We compute the fingerprint of each leaf naively in lines 3-4. In lines 5-17, we build $\mathcal{F}[2..h_\mathsf{max}]$. For each height $h \in [2, h_\mathsf{max}]$, we compute w_h in line 6. In line 8, we compute the length $l_\mathsf{ch} = s \cdot 2^{h-2}$ of the fingerprints at the child height $h-1$. Since for $i \in [1, w_h)$, the nodes $\mathcal{F}[h][i]$ are guaranteed to have two children, and the length of the fingerprints of their right children

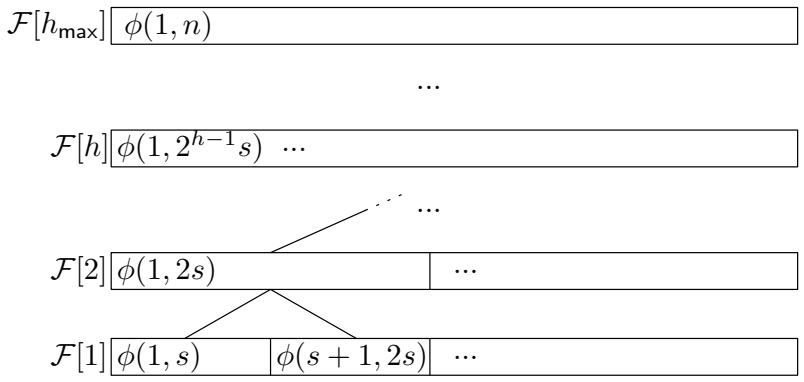


Figure 4.7: Illustration of $\mathcal{F}$ in Theorem 4.5.3.

$\mathcal{F}[h-1][2i]$ are guaranteed to be l_{ch}, we can compute all of them in the same way in lines 9-10. For the last (w_h-th) node $\mathcal{F}[h][w_h]$ at height h, this does not necessarily hold. In lines 18-19, we compute the positions $p_{\mathsf{lc}} = 2w_h + 1$ and $p_{\mathsf{rc}} = p_{\mathsf{lc}} + 1$ in $\mathcal{F}[h-1]$ of its left and its (potential) right child. $\mathcal{F}[h][w_h]$ has a right child iff $p_{\mathsf{rc}} \cdot l_{\mathsf{ch}} < n$ (the fingerprint of its left child ends before position n). If $\mathcal{F}[h][w_h]$ has no right child, then it has the same fingerprint as its left child, i.e, $\mathcal{F}[h][w_h] = \mathcal{F}[h-1][2i-1]$ (see lines 13-14). Else, the fingerprint of its right child is the last fingerprint at height $h-1$, hence it starts at the end $p_{\mathsf{rc}} \cdot l_{\mathsf{ch}}$ of $\mathcal{F}[h][w_h]$'s left child and goes up to the end of T, i.e, position n. Therefore, its length is $l_{\mathsf{rc}} = n + 1 - p_{\mathsf{rc}} \cdot l_{\mathsf{ch}}$ (see line 16), and we can compute $\mathcal{F}[h][p_{\mathsf{rc}}]$ in line 17 similarly to line 10.

Construction Time and Space. The tree has $< 2^{h_{\mathsf{max}}+1} = 2^{\lceil \log_2 \lceil n/s \rceil \rceil + 2} = \mathcal{O}(n/s)$ nodes, hence we can store $\mathcal{F}$ in $\mathcal{O}(n/s)$ space. Computing the fingerprints of the leaves takes $\mathcal{O}(s \cdot n/s) = \mathcal{O}(n)$ time. Computing the fingerprint of one of the $\mathcal{O}(n/s)$ internal nodes takes $\mathcal{O}(1)$ time with Lemma 4.5.2, hence this takes overall $\mathcal{O}(n/s)$ time. Overall, the construction takes $\mathcal{O}(n + \sqrt{n}) = \mathcal{O}(n)$ time and $\mathcal{O}(n/s + \sqrt{n})$ space.

Queries. Suppose we want to compute the fingerprint $\phi(p, p+l-1)$ of the substring $T[p, p+l)$. Now we use the samples in $\mathcal{F}$ (see Algorithm 19). If $l < s$, then we cannot use any samples and naively compute $\phi(p, p+l-1)$ in line 2. Else, we compute the longest sampled length $l_{\mathsf{h}} = s \cdot 2^{h-1}$ and its corresponding height $h = \lceil \log_2 \lceil l/s \rceil \rceil + 1$ in $\mathcal{F}$ that could fit into the substring $[p, p+l)$ in lines 3-4. In line 5, we determine the index $i = \lceil p/l_h \rceil$ in $\mathcal{F}[h]$ of the first sample with length l_h that starts at or after position p. If this sample ends after the end $e = p+l$ of the substring, i.e, it holds $(i+1) \cdot l_h > e$ (see line 7), then we cannot use it. If $h = 1$, then we cannot use any sample in $\mathcal{F}$, and compute $\phi(p, p+l-1)$ naively (see line 9). Else, we can use a shorter sample, so we decrement h by one, halve the sample length l_h, and adjust i (see lines 10-12).

Algorithm 18: build-fp-sampling()

```
1  h_max ← ⌈log_2⌈n/s⌉⌉ + 1;
2  F ← new array[1..h_max];
3  for i from 1 to ⌈n/s⌉ do
4      F[1][i] ← φ((i − 1) · s + 1, min(n, i · s));
5  for h from 2 to h_max do
6      w_h ← ⌈⌈n/s⌉/2^{h−1}⌉;
7      F[h] ← new array[1..w_h];
8      l_ch ← s · 2^{h−2};
9      for i from 1 to w_h − 1 do
10         F[h][i] ← concat-fps(
             F[h − 1][2i − 1],
             F[h − 1][2i], l_ch);
11     p_lc ← 2w_h + 1;
12     p_rc ← p_lc + 1;
13     if p_rc · l_ch > n then
14         F[h][w_h] ← F[h − 1][p_lc];
15     else
16         l_rc ← n + 1 − p_rc · l_ch;
17         F[h][w_h] ← concat-fps(
             F[h − 1][p_lc],
             F[h − 1][p_rc], l_rc);
```

Algorithm 19: fp-substr($[p, p + l)$)

```
1  if l < s then
2      return φ(p, p + l − 1);
3  h ← ⌈log_2⌈l/s⌉⌉ + 1;
4  l_h ← s · 2^{h−1};
5  i ← ⌈p/l_h⌉;
6  e ← p + l;
7  if (i + 1) · l_h > e then
8      if h = 1 then
9          return φ(p, p + l − 1);
10     h ← h − 1;
11     l_h ← l_h/2;
12     i ← ⌈p/l_h⌉;
13 f ← F[h][i];
14 p_m ← i · l_h;
15 p_r ← p_m + l_h;
16 if p_m > p then
17     f_l ← fp-substr(p, p_m − p);
18     f ← concat-fps(f_l, f, l_h);
19 if p_r < e then
20     l_r ← e − p_r;
21     f_r ← fp-substr(p_r, l_r);
22     f ← concat-fps(f, f_r, l_r);
23 return f;
```

Since now $l \geqslant s \cdot 2^h$ holds, the i-th sample of length $s \cdot 2^{h-1}$ must be fully contained in $[p, p + l)$. Now, we divide $T[p, p + l)$ up into three substrings $T[p_l, l_l)T[p_m, l_m)T[p_r, l_r) = T[p, p+l)$, where $p_m = i \cdot l_h$ and $l_m = l_h$. We start by computing the fingerprint $f = \mathcal{F}[h][i]$ of the middle substring $T[p_m, l_m)$ in line 13. If the left substring $T[p_l, l_l)$ is not empty ($p_m > p$ holds in line 16), then we recursively compute its fingerprint in line 17 and combine it with that of $T[p_m, l_m)$ in line 18 using Lemma 4.5.2. Therefore, $f = \phi(p_l, p_r - 1)$ holds after line 18. Finally, we recursively compute and combine the fingerprint of the right substring with f in lines 20-22 analogously to lines 17-18 if it is not empty ($p_r < e$). Now $f = \phi(p_l, p_r + l_r - 1) = \phi(p, p + l - 1)$ holds, and we return f.

Query Running Time. Let p'_l, p'_m, p'_r and p''_l, p''_m, p''_r be the starting positions of the left, middle, and right substrings in its left and right recursive calls within fp-substr($[p, p + l)$), respectively (see Figure 4.8). Since the middle substring $T[p'_m, p'_r)$ in the left recursive

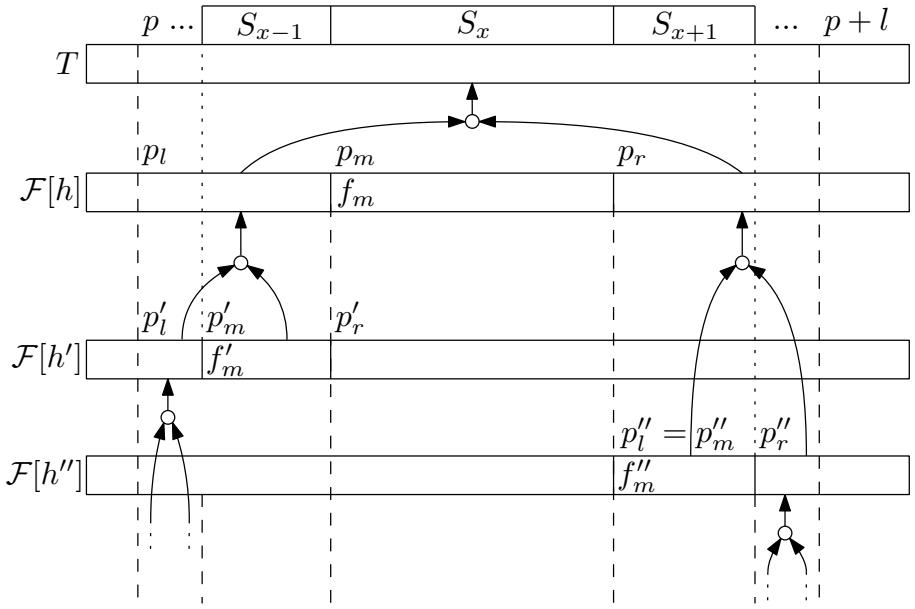


Figure 4.8: Illustration of the recursion tree of fp-substr($[p, p+l)$).

call is shorter than the middle substring $T[p_m, p_r)$, $T[p'_m, p'_r)$ ends at p_m, i.e, $p'_r = p_m$, hence the left recursive call does not issue a right recursive call. This argument can be continued inductively downwards the left-hand side of the recursion tree in order to show that each recursive call on the left branch makes at most one recursive call. Similarly, it holds $p''_l = p_r$ in the right recursive call, and due to the same argument, each recursive call in the right branch of the recursion tree makes at most one recursive call.

Let $T(l')$ be the running time of one such call fp-substr($[p', p'+l')$). By the choice of h, the middle substring $T[p'_{\mathsf{m}}, l'_{\mathsf{m}})$ has length $\geqslant l'/3$, hence we get the recurrence

$$T(l') = \begin{cases} T(2l'/3) + \mathcal{O}(1) & \text{if } l' \geqslant s, \\ \mathcal{O}(1) & \text{else,} \end{cases} \tag{4.9}$$

which solves to $T(l') = \mathcal{O}(\log(l'/s))$. Now it remains to assess the overall time spent in lines 2 and 9. Let $S_1 ... S_k = T[p, p+l)$ be the decomposition of $T[p, p+l)$ such that each S_i with $i \in [1, k]$ is either the middle substring $S_i = T[p'_{\mathsf{m}}, p'_{\mathsf{m}} + l'_{\mathsf{m}})$ in some recursive call fp-substring($[p', p'+l')$) within fp-substring($[p, p+l)$), or a string whose fingerprint is naively computed (in line 2 or 9) in another such recursive call (see Figure 4.8). Since we always choose the middle substring $T[p'_{\mathsf{m}}, p'_{\mathsf{m}} + l'_{\mathsf{m}})$ as large as possible, we have $|S_1| < ... \leqslant |S_x| \geqslant ... > |S_k|$, where $S_x = T[p_{\mathsf{m}}, p_{\mathsf{m}} + l_{\mathsf{m}})$, and $\forall y \in [2, k) : |S_y| \geqslant s$. Therefore, there are at most 2 instances where we compute the fingerprint of a substring naively (S_1 and S_k), and each such substring has length $< s$, hence lines 2 and 9 take overall $\mathcal{O}(s)$ time. Since the running time is $\mathcal{O}(l)$ if $l < s$, this results in an overall running time of $\mathcal{O}(\min(l, \log(l/s) + s))$. □

4.5.3 Sparse Prefix- and Suffix Array Interval Sampling

To speed up the computation of the sparse suffix array interval of a given length-λ pattern, we use a hash table storing the sparse suffix array intervals of all length-λ substrings $T[i, i+\lambda)$ in T starting at a sampled position.

4.5.3.1 Preliminaries

4.5.4 Definition. Let $\lambda \in [1, n]$ be a pattern length. Let SIV_S^λ be the set containing the sparse suffix array intervals of all length-λ substrings of T w.r.t. S, i.e, $\mathsf{SIV}_S^\lambda = \{\mathsf{siv}_S(T[S[i], S[i]+\lambda)) \mid i \in S\}$. Let S_S^λ be a function, where $\mathsf{S}_S^\lambda([b, e]) = \phi(S[\mathsf{SA}_S[b]], S[\mathsf{SA}_S[b]] + \lambda - 1)$. Finally, let $\mathsf{H}(\mathsf{SIV}_S^\lambda)$ be a hash table storing SIV_S^λ and using S_S^λ as the hash function. PIV_S^λ, P_S^λ and $\mathsf{H}(\mathsf{PIV}_S^\lambda)$ are defined analogously for sparse prefix array intervals.

4.5.5 Lemma. *Let $\lambda \in [1, n]$ be a pattern length, let $i \in [1, n-\lambda)$, and let $[b, e] = \mathsf{siv}_S(T[i, i+\lambda))$. Given $\mathsf{H}(\mathsf{SIV}_S^\lambda)$, a data structure for $\mathcal{O}(1)$ time LCE queries and $\phi(i, i+\lambda-1)$, we can compute $[b, e]$ (or report $[b, e] = \varnothing$) in $\mathcal{O}(1)$ expected time.*

Proof. Let B be the bucket in $\mathsf{H}(\mathsf{SIV}_S^\lambda)$ that the hash $\phi(i, i+\lambda-1)$ points to. For each $[b', e'] \in B$, we compute $l = \mathsf{LCE}(S[\mathsf{SA}_S[b']], i)$. If $l \geqslant \lambda$, then we return $[b', e']$. After iterating over B completely, we return $\varnothing$ to indicate that $[b, e]$ does not exist.

If $[b, e]$ does not exist, then there is no interval $[b', e'] \in \mathsf{SIV}_S^\lambda$ such that $\mathsf{LCE}(S[\mathsf{SA}_S[b']], i) \geqslant \lambda$, hence we correctly return $\varnothing$. If $[b, e]$ exists, then $T[S[\mathsf{SA}_S[b]], S[\mathsf{SA}_S[b]] + \lambda) = T[i, i+\lambda)$ implies $\mathsf{S}_S^\lambda([b, e]) = \phi(S[\mathsf{SA}_S[b]], S[\mathsf{SA}_S[b]] + \lambda - 1) = \phi(i, i+\lambda-1)$, hence $[b, e] \in B$, and because $\mathsf{LCE}(S[\mathsf{SA}_S[b]], i) \geqslant \lambda$, we correctly return $[b, e]$. Since the expected size of B is $\mathcal{O}(1)$, this takes $\mathcal{O}(1)$ expected time using $\mathcal{O}(1)$ time LCE queries. □

4.5.6 Observation. Let $\lambda \in [1, n]$ be a pattern length, let $\chi(\lambda) = |\{i \in [1, |S|] \mid \mathsf{LCP}_S[i] < \lambda\}|$, let $i \in [1, n-\lambda)$, and let $[b, e] = \mathsf{siv}_S(T[i, i+\lambda))$. Then, $\mathsf{LCP}_S[i] \geqslant \lambda$ for $i \in (b, e]$, and $\mathsf{LCP}_S[b] < \lambda$. Therefore, the total number of sparse suffix array intervals for length-λ patterns is at most $\chi(\lambda)$, i.e, $|\mathsf{SIV}_S^\lambda| \leqslant \chi(\lambda)$. Let $\mathsf{LCP}'_S[1..s]$ be the array LCP_S sorted in ascending order. We have $\chi(\lambda) = \max\{i \in [1, |S|] \mid \mathsf{LCP}'_S[i] < \lambda\}$, which we can compute in $\mathcal{O}(\log |S|)$ time using a binary search over LCP'_S. This gives us an upper bound for the size of $\mathsf{H}(\mathsf{SIV}_S^\lambda)$.

Since we have $\sigma \leqslant 256 = 2^8$ in practice, the sparse prefix- and suffix array for patterns of length 1 and 2 can be stored in arrays of length 2^8 and 2^{16}, respectively. Therefore, Lemma 4.5.5 is only useful for $\lambda \geqslant 3$. If we have a maximum wanted pattern length λ_{max} and a maximum wanted number of sampled intervals θ, then we can use Theorem 4.5.7 to compute a set of pattern lengths between 3 and λ_{max} and hash maps according to Definition 4.5.4 for each of those pattern lengths such that (**i**) at most θ intervals are

sampled, (**ii**) $\Theta(\theta)$ are sampled in expectation, and (**iii**) the number of sampled intervals rises linearly with the pattern length index (not the pattern length itself).

To see why we chose (**iii**), suppose that the majority of values in $\mathsf{LCP}_{\mathcal{C}}$ lie in a small range $[x, y]$. Then the majority of phrases in the LZ77 factorization have length $l \in [x, y]$. In this case, (**iii**) ensures that many of the chosen pattern lengths lie in the range $[x, y]$ and thus speeds up the search for l.

4.5.3.2 Data Structure

4.5.7 Theorem. *Let S be a sampling of text positions, let $\lambda_{\mathsf{max}} \geqslant 3$ with $\lambda_{\mathsf{max}} = \mathcal{O}(1)$ be the maximum pattern length, let $\theta \geqslant |S|$ be the maximum number of samples, let $\mathcal{D}$ be a data structure for computing the Karp-Rabin fingerprint of a substring of T, let $q_{\mathcal{D}}$ and $c_{\mathcal{D}}$ be its query- and construction time, respectively, and let $s_{\mathcal{D}}$ be its size. Then, we can compute in $\mathcal{O}(n/\log_\sigma n + |S| \log |S| + c_{\mathcal{D}} + \theta \cdot q_{\mathcal{D}})$ expected time and $\mathcal{O}(n/\log_\sigma n + \theta + s_{\mathcal{D}})$ expected space a set of pattern lengths $\Lambda = \{\lambda_1, ..., \lambda_M\}$ and hash tables $\mathsf{H}(\mathsf{SIV}_S^{\lambda_1}), ..., \mathsf{H}(\mathsf{SIV}_S^{\lambda_M})$ with $\lambda_1 < ... < \lambda_M = \lambda_{\mathsf{max}}$ such that for $i \in [1, M]$,*

(**i**) $\sum_{i=1}^{M} |\mathsf{H}(\mathsf{SIV}_S^{\lambda_i})| \leqslant \theta$, *and*

(**ii**) $\sum_{i=1}^{M} |\mathsf{H}(\mathsf{SIV}_S^{\lambda_i})| = \Theta(\theta)$ *in expectation, and*

(**iii**) $|\mathsf{H}(\mathsf{SIV}_S^{\lambda_i})| - \chi(3) = \Theta(i)$ *in expectation.*

Proof. We start by computing $\mathsf{SA}_S[1..|S|]$ in $\mathcal{O}(n/\log_\sigma + |S| \log |S|)$ time and $\mathcal{O}(n/\log_\sigma n + |S|)$ space using Lemma 3.2.4. Now, we compute $\mathsf{LCP}_S[1..|S|]$ in $\mathcal{O}(|S|)$ time according to Definition 3.2.2 using the $\mathcal{O}(1)$ time LCE data structure from Theorem 2.5.2. We compute LCP'_S in $\mathcal{O}(|S| \log |S|)$ time.

Let $\mu = \chi(3)$ and $\nu = \chi(\lambda_{\mathsf{max}})$. We choose

$$\forall\ i \in [1, M) : \lambda_i = \mathsf{LCP}'_S[\lfloor \mu + \frac{i}{M}(\nu - \mu) \rfloor] \tag{4.10}$$

(see Figure 4.9), which implies

$$\forall\ i \in [1, M] : |\mathsf{H}(\mathsf{SIV}_S^{\lambda_i})| \leqslant \chi(\lambda_i) \leqslant \mu + \frac{i}{M}(\nu - \mu). \tag{4.11}$$

However, we have not yet set M. If we insert Equation (4.11) into (**i**), then we get

$$\begin{gathered} \sum_{i=1}^{M} |\mathsf{H}(\mathsf{SIV}_S^{\lambda_i})| \leqslant \theta \Leftrightarrow \sum_{i=1}^{M} (\mu + \frac{i}{M}(\nu - \mu)) \leqslant \theta \Leftrightarrow \\ M\mu + \frac{\nu - \mu}{M} \sum_{i=1}^{M} i \leqslant \theta \Leftrightarrow M\mu + \frac{\nu - \mu}{M} \frac{M(M+1)}{2} \leqslant \theta \Leftrightarrow \\ M(\mu + \frac{\nu - \mu}{2}) + \frac{\nu - \mu}{2} \leqslant \theta \Leftrightarrow M \leqslant \frac{2\theta + \mu - \nu}{\mu + \nu}. \end{gathered} \tag{4.12}$$

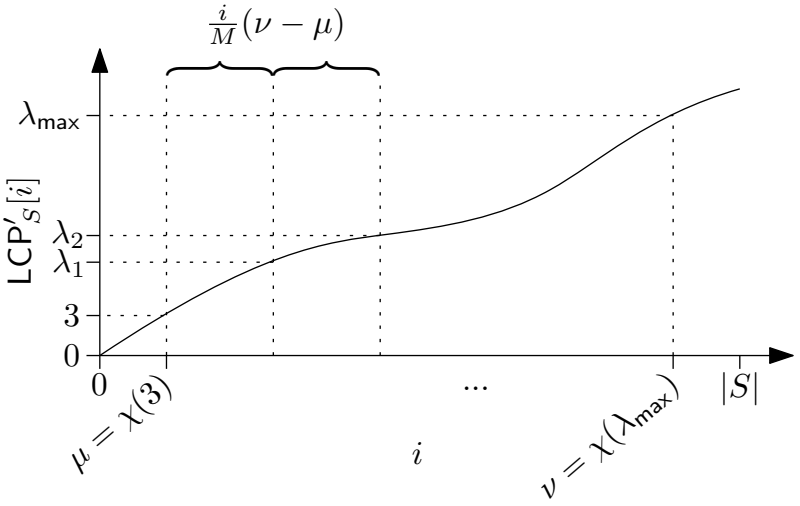


Figure 4.9: Illustration of the choice of Λ in Theorem 4.5.7. The graph shows $\mathsf{LCP}'_S[i]$ at position i.

Therefore, setting (**iv**) $M = \lfloor (2\theta + \mu - \nu)/(\mu + \nu) \rfloor$ and choosing λ_i according to Equation (4.10) ensures (**i**). Now, we show that $M \geqslant 1$ always holds.

$$M \geqslant 1 \Leftrightarrow \frac{2\theta + \mu - \nu}{\mu + \nu} \geqslant 1 \Leftrightarrow 2\theta + \mu - \nu \geqslant \mu + \nu \Leftrightarrow \theta \geqslant \nu \tag{4.13}$$

Equation (4.13) always holds, because we have $\nu \in [1, |S|]$ and $\theta \geqslant |S|$. If we assume that in expectation,

$$\begin{aligned} &(\mathbf{v}) \quad \chi(\lambda_i) = \mu + \Theta\left(\frac{i}{M}(\nu - \mu)\right) \text{ and} \\ &(\mathbf{vi}) \quad |\mathsf{SIV}_S^{\lambda_i}| = \Theta(\chi(\lambda_i)) \end{aligned} \tag{4.14}$$

hold for $i \in [1, M]$, then we get

$$\begin{aligned} \chi(\lambda_i) = \mu + \Theta\left(\frac{i}{M}(\nu - \mu)\right) &\Leftrightarrow \chi(\lambda_i) - \mu = \Theta\left(\frac{i}{M}(\nu - \mu)\right) \Leftrightarrow \\ \chi(\lambda_i) - \chi(3) = \Theta\left(\frac{i}{M}(\nu - \mu)\right) &\Leftrightarrow |\mathsf{H}(\mathsf{SIV}_S^{\lambda_i})| - \chi(3) = \Theta\left(\frac{i}{M}(\nu - \mu)\right) \Leftrightarrow \\ &|\mathsf{H}(\mathsf{SIV}_S^{\lambda_i})| - \chi(3) = \Theta(i) \end{aligned} \tag{4.15}$$

for $i \in [1, M]$, which is (**iii**). Furthermore, (**iv**), (**v**) and (**vi**) yield

$$\begin{aligned} \sum_{i=1}^{M} |\mathsf{H}(\mathsf{SIV}_S^{\lambda_i})| &= \sum_{i=1}^{M} \Theta(\chi(\lambda_i)) = \\ \sum_{i=1}^{M} \Theta\left(\mu + \frac{i}{M}(\nu - \mu)\right) &= \Theta\left(M\mu + \frac{\nu - \mu}{M} \sum_{i=1}^{M} i\right) = \\ \Theta\left(M\mu + \frac{\nu - \mu}{M} \frac{M(M+1)}{2}\right) &= \Theta\left(M\frac{\mu + \nu}{2} + \frac{\nu - \mu}{2}\right) = \\ \Theta\left(\left\lfloor \frac{2\theta + \mu - \nu}{\mu + \nu} \right\rfloor \frac{\mu + \nu}{2} + \frac{\nu - \mu}{2}\right) &= \Theta(\theta), \end{aligned} \tag{4.16}$$

which is (**ii**). Lines 1-7 in Algorithm 20 summarize the computation of Λ. In lines 1-2, we compute μ and ν according to Observation 4.5.6 in $\mathcal{O}(\log|S|)$ time. In line 3, we preliminarily choose M according to Equation (4.12). Then, we choose λ_i for $i \in [1, M]$ according to Equation (4.10) in lines 5-6. In general, we may have chosen some pattern length twice, i.e, $\lambda_i = \lambda_{i-1}$ may hold for some $i \in [2, M]$. In this case, $|\Lambda| < M$ holds. To account for this, we set $M \leftarrow |\Lambda|$ in line 4.

Now, it remains to compute the hash tables. Overall, we iterate once over $\mathsf{LCP}_S[2..|S|]$, i.e, we consider all samples and therefore also all sparse suffix array intervals in lexicographical order. We maintain an array B that stores for each pattern length λ_j at position j the start of the sparse suffix array interval of the lexicographically next largest not yet considered length-λ_j substring of T that starts at a sampled position. More formally, let $i \in [2, |S|+1]$. Let $j \in [1, M]$, let $p_j = S[\mathsf{SA}_S[i-1]]$, and let $[b_j, e_j] = \mathsf{siv}_S(T[p_j, p_j + \lambda_j))$. At the start of the $i-1$st iteration, the following holds for each $j \in [1, M]$. If $p_j + \lambda_j > n+1$, then $B[j] = i - 1$, and $B[j] = b_j$, else (see Figure 4.10).

At the start of the first iteration ($i = 2$), $B[1..M] = [1, ..., 1]$ does not violate the invariant. Suppose we encounter an LCP value $\mathsf{LCP}_S[i] < \lambda_j$ at position i. If $p_j + \lambda_j \leqslant n+1$, then $[B[j], i) \in \mathsf{SIV}_S^{\lambda_j}$, hence we compute $\mathsf{S}_S^{\lambda}([B[j], i)) = \phi(p_j, p_j + \lambda_j - 1)$ in line 15 and use it to insert $[B[j], i)$ into $\mathsf{H}(\mathsf{SIV}_S^{\lambda_j})$ in line 16 (see Definition 4.5.4). In any case, setting $B[j] \leftarrow i$ in line 17 maintains the invariant. Finally, we add a sentinel value $\mathsf{LCP}_S[|S|+1] = 0$ to LCP_S to ensure that we consider all intervals ending at $|S|$ in the last iteration.

For each $\lambda_j \in \Lambda$, the if-clause in line 14 fails iff $S[\mathsf{SA}_S[i-1]] > n - \lambda_j$ holds. Since $\lambda_1, ..., \lambda_M \leqslant \lambda_{\mathsf{max}} = \mathcal{O}(1)$, this happens at most

$$\sum_{j=1}^{M} |S \cap (n - \lambda_j, n]| \leqslant \sum_{j=1}^{M} \lambda_j = \sum_{j=1}^{M} \mathcal{O}(1) = \mathcal{O}(M) = \mathcal{O}(\theta) \tag{4.17}$$

times. For every successful evaluation of the if-clause in line 14, we insert one interval into some hash table. Therefore, the if-clause succeeds $\sum_{j=1}^{M} |\mathsf{H}(\mathsf{SIV}_S^{\lambda_j})| = \Theta(\theta)$ times. Since it fails $\mathcal{O}(\theta)$ times and succeeds $\Theta(\theta)$ times, we reach line 14 $\Theta(\theta)$ times. Furthermore, we

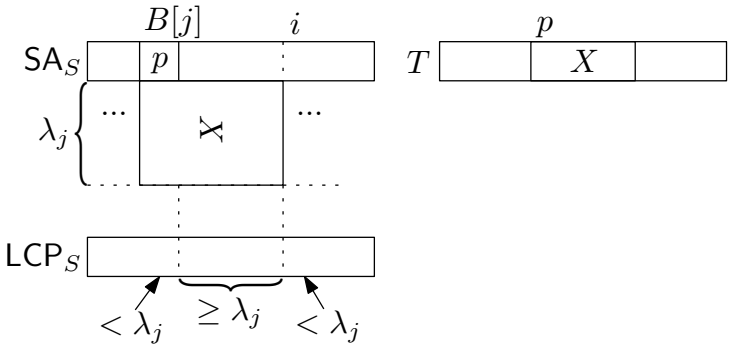


Figure 4.10: Illustration of Algorithm 20.

Algorithm 20: build-ssa-interval-sampling()

```
1  μ ← max{i ∈ [1, |S|] | LCP'_S[i] < 3};
2  ν ← max{i ∈ [1, |S|] | LCP'_S[i] < λ_max};
3  M ← ⌊(2θ + μ − ν)/(μ + ν)⌋;
4  Λ ← {λ_max};
5  for i from 1 to M − 1 do
6      Λ ← Λ ∪ {LCP'_S[⌊μ + (i/M)(ν − μ)⌋]};
7  M ← |Λ|;
8  B ← new array[1..M];
9  B ← [1, ..., 1];
10 for i from 2 to |S| + 1 do
11     for j from M to 1 do
12         if LCP_S[i] ⩾ λ_j then
13             break;
14         if S[SA_S[i − 1]] + λ_j ⩽ n + 1 then
15             f ← fp-substr(S[SA_S[i − 1]], λ_j);
16             H(SIV_S^{λ_j})[f] ← [B[j], i);
17         B[j] ← i;
```

break out of the inner for-loop (lines 11-17) at most $|S|$ times. Therefore, we run through $\Theta(\theta + |S|) = \Theta(\theta)$ iterations of the inner for-loop. Since we compute exactly one Karp-Rabin fingerprint per inserted interval, the outer for-loop takes $\Theta(\theta \cdot q_{\mathcal{D}})$ expected time. Overall, the construction takes $\mathcal{O}(n/\log_\sigma n + |S| \log |S| + c_{\mathcal{D}} + \theta \cdot q_{\mathcal{D}})$ expected time and $\mathcal{O}(n/\log_\sigma n + |S| + \theta + s_{\mathcal{D}}) = \mathcal{O}(n/\log_\sigma n + \theta + s_{\mathcal{D}})$ expected space. □

4.5.4 Orthogonal Range One-Reporting

The insertion-only orthogonal range one-reporting (IO-OROR) problem (see Definition 3.2.1) can be reduced (see Lemma 4.5.9) to the static weighted orthogonal range one-reporting (SW-OROR) problem (see Definition 4.5.8) if the insertion order is known beforehand.

4.5.8 Definition. Let π be a permutation of $[1, N]$, let $\mathcal{U} = \{(i, \pi(i)) \mid i \in [1, N]\}$ be the set of valid points, and let $w((i, \pi))$ be the weight of the point $(i, \pi(i)) \in \mathcal{U}$. The problem of *static weighted orthogonal range one-reporting* (SW-OROR) is, given a query rectangle $\mathcal{Q} = [x_1, x_2] \times [y_1, y_2]$ and weight w', to output a point $p \in \mathcal{Q} \cap \mathcal{U}$ with $w(p) \leqslant w'$, or report that there is no such point. We call $\langle \pi, w \rangle$ an instance of the SW-OROR problem.

4.5.9 Lemma. *Let π be a permutation of $[1, N]$, let $\mathcal{U} = \{(i, \pi(i)) \mid i \in [1, N]\}$ be the set of valid points, let $I = \langle (i_1, \pi(i_1)), ..., (i_M, \pi(i_M)) \rangle$ be the sequence of inserted points in an*

instance $\langle \pi, I \rangle$ of the IO-OROR problem with fixed insertion order. Then, this instance can be reduced to an instance of the SW-OROR problem.

Proof. Let $\mathcal{A}_j = \{(i_k, \pi(i_k)) \mid k \in [1, j]\}$ be the set of inserted points after j insertions, and let $\langle \pi, w \rangle$ be an instance for the SW-OROR problem, where $w((i_j, \pi(i_j))) = j$ for each $j \in [1, M]$, and $w(p) = \infty$ for each $p \in \mathcal{U} \backslash \mathcal{A}_M$.

We will now show that we can solve $\langle \pi, I \rangle$ by solving $\langle \pi, w \rangle$. Let $\mathcal{Q} = [x_1, x_2] \times [y_1, y_2]$ be a query rectangle for the instance $\langle \pi, I \rangle$, and let $(i_j, \pi(i_j))$ with $j \in [1, M]$ be the last inserted point. For each $p \in \mathcal{U}$, we have $p \in \mathcal{A}_j \cap \mathcal{Q} \Leftrightarrow p \in \mathcal{Q} \wedge w(p) \leqslant j$, hence we can answer the query to $\langle \pi, I \rangle$ by querying $\langle \pi, w \rangle$ with the same rectangle $\mathcal{Q}$ and weight $w' = j$. □

In the algorithm for the exact LZ77 factorization, the insertion order fixed (see Observation 3.2.3). Therefore, we will also discuss data structures for SW-OROR (see Section 4.5.4.2).

4.5.4.1 Dynamic Data Structures

We begin with the Dynamic Square Grid, which is a very simple data structure for IO-OROR. Then, we discuss the Semi-Dynamic Square Grid, which is a slightly optimized version for the case, where the set of inserted points is known beforehand.

4.5.4.1.1 Dynamic Square Grid (D-SG) Here, we split the overall range $[1, N] \times [1, N]$ into $\approx (N/m)^2$ equally-sized square cells with width $m \geqslant 1$. Thus, we have a $g \times g$ grid of square cells, where $g = \lceil N/m \rceil$ (see Figure 4.11).

Data Structure. The data structure consists of $g^2 = \mathcal{O}((N/m)^2)$ dynamic arrays $C_1, ..., C_{g^2}$, where each point $p = (x, y) \in \mathcal{A}_M$ will be inserted into $C_{c(p)}$, where $c(p) = g\lfloor y/m \rfloor + \lfloor x/m \rfloor + 1$ is the cell index of p.

Pre-Processing. The pre-processing consists of the initialization of the arrays $C_1, ..., C_{g^2}$. This takes $\mathcal{O}(g^2) = \mathcal{O}((N/m)^2)$ time and space.

Size. The overall size of this data structure is $\mathcal{O}(g^2 + N) = \mathcal{O}((N/m)^2 + N)$. Storing one point needs 8 bytes if $N < 2^{32}$, and 16 bytes, else. Storing one dynamic array (ignoring the points) needs 16 bytes (8 for the size, and 8 for the pointer to the data). Storing the g^2 dynamic arrays $C_1, ..., C_{g^2}$ therefore requires $16 \cdot g^2 = 16 \cdot \lceil N/m \rceil^2$ bytes. Thus, this data structure needs $8 \cdot N + 16 \cdot \lceil N/m \rceil^2$ bytes if $N < 2^{32}$, and $16 \cdot N + 16 \cdot \lceil N/m \rceil^2$ bytes, else.

Queries. Algorithm 21 shows the query algorithm. To answer a query with rectangle $\mathcal{Q} = [x_1, x_2] \times [y_1, y_2]$, we compute the leftmost, rightmost, lowest and highest cell ranks $x'_1 = \lfloor x_1/m \rfloor$, $x'_2 = \lfloor x_2/m \rfloor$, $y'_1 = \lfloor y_1/m \rfloor$ and $y'_2 = \lfloor y_2/m \rfloor$, respectively. Then, we iterate for each $(x', y') \in [x'_1, x'_2] \times [y'_1, y'_2]$ over the array C_i, where $i = g \cdot y' + x' + 1$ is the cell index of the cell with x-rank x' and y-rank y'. If we encounter a point $p \in \mathcal{Q}$, then we return p. Else, we return $\perp$.

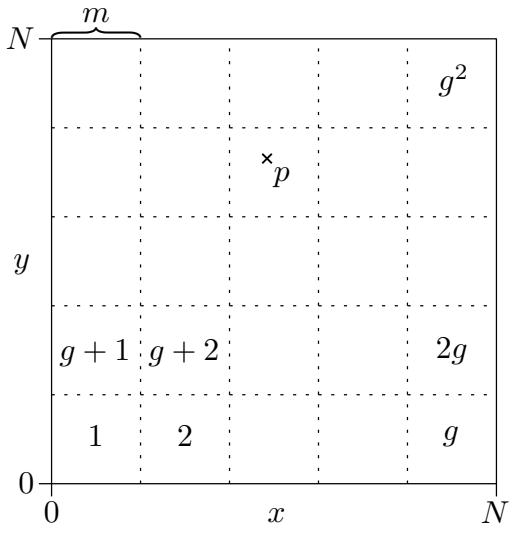


Figure 4.11: Illustration of the grid-based data structures. Each cell is labeled with its cell index. In this case, the cell index of the point $p = (x, y)$ is $c(p) = g\lfloor y/m \rfloor + \lfloor x/m \rfloor + 1 = 3g + 3 = 18$, where $g = 5$.

Query Time. $\mathcal{Q}$ fully contains $|\mathcal{Q}|/m^2$ cells. If we find a point in one such cell, we can directly return it, hence we spend $\mathcal{O}(|\mathcal{Q}|/m^2)$ time for the fully contained cells. $\mathcal{Q}$ cuts $< 4g = \mathcal{O}(N/m)$ boundary cells. For each point in a boundary cell, we have to check whether it lies in $\mathcal{Q}$. However, since the points have distinct x- and y coordinates, there are $\leqslant 4m$ such points in total, hence we need overall $\mathcal{O}(N/m + m)$ time for the boundary cells. In total, we need $\mathcal{O}(|\mathcal{Q}|/m^2 + N/m + m)$ time.

4.5.4.1.2 Semi-Dynamic Square Grid (SD-SG)

If we know the insertion order I beforehand (see Lemma 4.5.9), we can reduce the space of the D-SG.

Data Structure. Instead of storing g^2 dynamic arrays, we store 3 arrays $P[1..N]$, $L[1..g^2]$ and $X[1..g^2]$, where

- X is initialized such that $X[i] = |\{p \in \mathcal{A}_M \mid c(p) < i\}| + 1$ is the number of points in $\mathcal{A}_M$ with cell indices smaller than i, for each $i \in [1, g^2]$, and after j insertions,
- $L[i] = |\{p \in \mathcal{A}_j \mid c(p) = i\}|$ stores the number of already inserted points with cell index i, for each $i \in [1, g^2]$, and
- each $p = (x, y) \in \mathcal{A}_j$ is stored in $P[X[c(p)], X[c(p)] + L[c(p)]]$.

Intuitively, $P[X[i], X[i+1]]$ is the range in P where all points in the cell with index i will be stored, and $P[X[i], X[i] + L[i]]$ is the range containing the already inserted points in the cell with index i.

Size. As with the D-SG, the overall size of this data structure is $\mathcal{O}(g^2 + N) = \mathcal{O}((N/m)^2 + N)$. If we assume that $m \leqslant 2^{16}$, then we can store each entry in L with

Algorithm 21: query-D-SG($\mathcal{Q}$)

```
x'_1 = ⌊x_1/m⌋;
x'_2 = ⌊x_2/m⌋;
y'_1 = ⌊y_1/m⌋;
y'_2 = ⌊y_2/m⌋;
for x' from x'_1 to x'_2 do
    for y' from y'_1 to y'_2 do
        i ← g · y' + x' + 1;
        for p ∈ C_i do
            if p ∈ Q then
                return p;
return ⊥;
```

Algorithm 22: build-SW-SG()

```
L ← [0, ..., 0];
for p = (x, y) ∈ U do
    j ← c(p) = g⌊y/m⌋ + ⌊x/m⌋ + 1;
    L[j] ← L[j] + 1;
X[1] ← 1;
for i from 2 to g^2 do
    X[i] ← X[i − 1] + L[i − 1];
L ← [0, ..., 0];
for p = (x, y) ∈ U do
    j ← c(p) = g⌊y/m⌋ + ⌊x/m⌋ + 1;
    P[X[j] + L[j]] ← ⟨p, w(p)⟩;
    L[j] ← L[j] + 1;
```

2 bytes. If $N < 2^{32}$, then we store each entry in X with 4 bytes. Else, we need 8 bytes. Thus, this data structure needs $8 \cdot N + 6 \cdot \lceil N/m \rceil^2$ bytes if $N < 2^{32}$, and $16 \cdot N + 10 \cdot \lceil N/m \rceil^2$ bytes, else.

Pre-Processing. We start by initializing $L \leftarrow [0, ..., 0]$. Then, we increment $L[c(p)]$ by 1, for each $p \in A_M$. Now, we set $X[1] \leftarrow 1$, and for i from 2 to g^2, we set $X[i] \leftarrow X[i-1] + L[i-1]$. Finally, we reset $L \leftarrow [0, ..., 0]$. The pre-processing takes $\mathcal{O}(g^2) = \mathcal{O}((N/m)^2)$ time and space.

Queries. The query procedure is almost identical to that of the D-SG, but here, we iterate for each cell index i with j over the range $P[X[i], X[i] + L[i])$. Again, if we encounter a point $P[j] \in \mathcal{Q}$, then we return $P[j]$. Else, we return $\perp$.

Query Time. This data structure has the same query time as D-SG. In general, a query takes $\mathcal{O}(|\mathcal{Q}|/m^2 + N/m + m)$ time.

4.5.4.2 Static Weighted Data Structures

As argued in Lemma 4.5.9, we can implement a data structure for SW-OROR instead of IO-OROR if the insertion order is known beforehand. We will now discuss several data structures for the SW-OROR problem.

4.5.4.2.1 Static Weighted Square Grid (SW-SG) This is the static version of the SD-SG.

Data Structure. We store two arrays $D[1..N]$ and $X[1..g^2]$, where

- $X[i] = |\{p \in \mathcal{U} \mid c(p) < i\}| + 1$ is the number of points in $\mathcal{U}$ with cell indices smaller than i, for each $i \in [1, g^2]$, and

- $\langle p, w(p)\rangle \in D[X[c(p)], X[c(p)+1])$, for each $p \in \mathcal{U}$.

$L[1..g^2]$ is not needed anymore, because we do not allow insertions. However, we now also have to store the weight for each point in D.

Size. As with the SD-SG, the overall size of this data structure is $\mathcal{O}(g^2 + N) = \mathcal{O}((N/m)^2 + N)$. However, we do not store L, but additionally store the weight of each point, which needs 4 bytes if $N < 2^{32}$, and 8 bytes, else. Overall, this data structure needs $12 \cdot N + 4 \cdot \lceil N/m \rceil^2$ bytes if $N < 2^{32}$, and $24 \cdot N + 8 \cdot \lceil N/m \rceil^2$ bytes, else.

Construction. We start by constructing X and L as with the SD-SG (see lines 1-8 in Algorithm 22). Then, we write each point $p \in \mathcal{U}$ and its weight $w(p)$ to the next free position $P[X[j] + L[j]] \leftarrow \langle p, w(p)\rangle$ in its cell $j = c(p)$, and increment the number $L[j]$ of already inserted points in its cell (see lines 9-12). The construction takes $\mathcal{O}(g^2 + N) = \mathcal{O}((N/m)^2 + N)$ time and space.

Queries. The query procedure is identical to that of the SD-SG, with the difference that we have to check if the weight $w(P[j])$ of a considered point does not exceed the query weight w'.

Query Time. The query time is similar to that of the D-SG and the SD-SG, with the difference that we cannot directly return a point in a fully contained cell. However, since there are at most $\min(|[x_1, x_2]|, |[y_1, y_2]|) \leqslant \sqrt{|\mathcal{Q}|}$ points in $\mathcal{Q}$, we need overall $\mathcal{O}(|\mathcal{Q}|/m^2 + N/m + m + \sqrt{|\mathcal{Q}|})$ time.

4.5.4.2.2 Static Weighted Striped Square (SW-SS)

Instead of dividing the overall range $[1, N] \times [1, N]$ into squares, we now divide it into $\lceil N/s \rceil$ vertical segments with width s, where $s \geqslant 1$ is an integer, and sort the points inside each segment by their y-coordinates. By the definition of $\mathcal{U}$, there is exactly one point for each x- and y-coordinate, hence each segment contains exactly s points (except for possibly the last segment). This simplifies the data structure, its construction, and queries.

Data Structure. We store the array $S[1..N]$, where

- for each point $p = (x, y) \in \mathcal{U}$, $\langle p, w(p)\rangle$ is stored in $S[b, e]$, where $b = \lfloor x/s \rfloor s + 1$, $e = \min(N, \lfloor x/s \rfloor (s+1))$, and
- the points stored in $P[b, e]$ are sorted in ascending order by their y-coordinates, for each $[b, e] \in \mathcal{S} = \{[i \cdot s + 1, \min(N, (i+1) \cdot s)] \mid i \in [0, \lfloor N/s \rfloor]\}$.

Size. This data structure needs $\mathcal{O}(N)$ space. Since we only plainly store each point and its weight, we need $12 \cdot N$ bytes if $N < 2^{32}$, and $24 \cdot N$ bytes, else.

Construction. At first, we place each point into the correct segment in S (similar to radix-sort). To do so, we maintain that in the array $R[1..\lceil N/s \rceil]$, we store at position i the number of already inserted points in the i-th segment + 1 (R is similar to L from the SD-DG and the SW-SG). For each $p = (x, y) \in \mathcal{U}$, we write $S[\lfloor x/s \rfloor + R[\lfloor x/s \rfloor]] \leftarrow \langle p, w(p)\rangle$

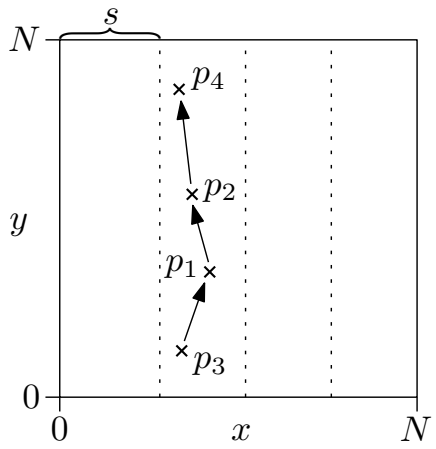


Figure 4.12: Illustration of the SW-SS. In this case, we have $s = 4$, and the points $p_1, ..., p_4$ in the second segment are stored in $S[s+1, 2s] = [p_3, p_1, p_2, p_4]$.

Algorithm 23: query-SW-SQ$(\mathcal{Q}, w')$

```
1  x'_1 ← ⌊x_1/s⌋;
2  x'_2 ← ⌊x_2/s⌋;
3  for x' from x'_1 to x'_2 do
4      b ← x' · s + 1;
5      t ← min(N + 1, b + s);
6      while b ≠ t do
7          c ← (b + t)/2;
8          if S[c].p.y < y_1 then
9              b ← c + 1;
10         else if S[c].p.y < y_2 then
11             t ← m;
12         break;
13     if [b, t) = ∅ then
14         continue;
15     for i from c to t do
16         if S[i].p.y > y_2 then
17             break;
18         if S[i].w ∈ Q ∧ S[i].w < w' then
19             return S[i].p;
20     for i from c − 1 down to b do
21         if S[i].p.y < y_1 then
22             break;
23         if S[i].w ∈ Q ∧ S[i].w < w' then
24             return S[i].p;
25 return ⊥;
```

and increment $R[\lfloor x/s \rfloor]]$ by 1. Finally, we sort the points in $P[b, e)$ by their y-coordinates, for each $[b, e) \in \mathcal{S}$. This takes $\mathcal{O}(N + (N/s)(s \log s)) = \mathcal{O}(N \log s)$ time.

In practice, we use another method to construct S. We set $S \leftarrow \mathcal{U}$, and then sort S such that for each $i, j \in [1, N]$, it holds

$$i < j \Leftrightarrow (\lfloor S[i].p.x/s \rfloor < \lfloor S[j].p.x/s \rfloor \vee \tag{4.18}$$

$$(\lfloor S[i].p.x/s \rfloor = \lfloor S[j].p.x/s \rfloor \wedge S[i].p.y < S[j].p.y)). \tag{4.19}$$

This takes $\mathcal{O}(N \log N)$ time and does not require the array $R[1..\lceil N/s \rceil]$.

Queries. Algorithm 23 shows the query algorithm. We start by computing the leftmost and rightmost segment ranks x'_1 and x'_2. For each x'-th cell with $x' \in [x'_1, x'_2]$, we perform a binary search for those points in this segment that lie in $\mathcal{Q}$ if there are any (see lines 3-12). If the point at the center position c in the current search range $[b, t)$ lies below $\mathcal{Q}$ ($S[c].p.y < y_1$ holds in line 8), then we continue the search above $S[c].p$, i.e, set $b \leftarrow c + 1$ (see line 9). Similarly, if it lies above $\mathcal{Q}$, then we continue the search below it (see line 11). Else, $P[c].p$ lies inside, left or right of $\mathcal{Q}$, and we abort the binary search (see line 12). If there is no such point, then $[b, t) = \varnothing$ holds in line 13, and we continue with the next segment (see line 14). Else, we scan with i from $S[c]$ up to $S[t-1]$, and then from $S[c-1]$ down to $S[b]$ (see lines 15-24). If there is a point $S[i].p \in \mathcal{Q}$ with $S[j].w \leqslant w'$, then we return p. Once we see a point above $\mathcal{Q}$ (see line 16) or below $\mathcal{Q}$ (see line 21), then we can stop scanning, because the points in each segment are sorted by their y-coordinates. Finally, we return $\perp$ in line 25 if we have not found any point. The algorithm is correct, because we consider every segment that is cut by $\mathcal{Q}$, and only filter out points above and below $\mathcal{Q}$.

Query Time. Let $\bar{x} = |[x_1, x_2]|$ and $\bar{y} = |[y_1, y_2]|$. $\mathcal{Q}$ cuts $\lceil \bar{x}/s \rceil$ segments, hence we spend $\mathcal{O}((\bar{x}/s) \log s)$ time for the binary searches in total. In lines 15-24 we scan over at most all the $\min(\bar{x}, \bar{y}) \leqslant \sqrt{|\mathcal{Q}|}$ points in $\mathcal{Q} \cap \mathcal{U}$, and $< 2s$ additional points that lie left and right of $\mathcal{Q}$ in the leftmost and rightmost segments that are cut by $\mathcal{Q}$. Thus, we scan over $\mathcal{O}(\sqrt{|\mathcal{Q}|} + s)$ points in total. The overall query time is $\mathcal{O}((\bar{x}/s) \log s + s + \sqrt{|\mathcal{Q}|})$. If $\bar{x} = \mathcal{O}(s/\log s \sqrt{|\mathcal{Q}|})$, then one query takes $\mathcal{O}(((s/\log s \sqrt{|\mathcal{Q}|})/s) \log s + \sqrt{|\mathcal{Q}|} + s) = \mathcal{O}(\sqrt{|\mathcal{Q}|} + s)$ time.

4.5.4.2.3 Static Weighted K-D Tree (SW-KDT)

A k-d tree with $k = 2$ divides a point set recursively alternately according to the x- and y-coordinates into 2 subsets of approximately equal size.

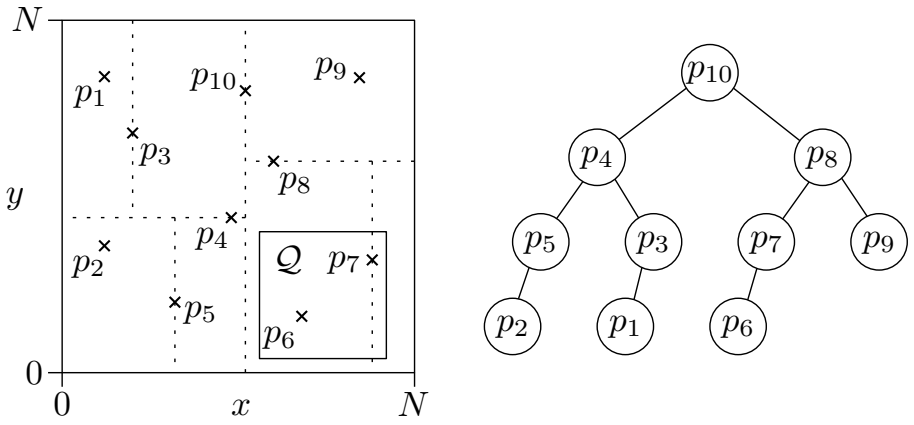
4.5.10 Definition. Let $P = \{(x_1, y_1), ..., (x_n, y_n)\} \subseteq \mathbb{R}^2$. Then, the x-2-d tree of P is defined as either a single node $\langle (x_1, y_1) \rangle$ if $n = 1$, or the tree with root $\langle (x_m, y_m) \rangle$, where

- $m \in [1, n]$ such that $|P_<| = \lceil n/2 \rceil$, where
  $P_< = \{(x, y) \in P \mid x < x_m\}$ and
  $P_> = \{(x, y) \in P \mid x > x_m\}$, and

- the left child of $\langle (x_m, y_m) \rangle$ is the root of a y-2-d tree of $P_<$, and
- the right child of $\langle (x_m, y_m) \rangle$ is the root of a y-2-d tree of $P_>$ if $P_> \neq \varnothing$.

The y-2-d tree of P is defined analogously to the x-2-d tree of P, with the difference that the point set is divided into 2 subsets based on the y-coordinates, and that the children of the root of a y-2-d tree are roots of x-2-d trees.

4.5.11 Example. The dotted lines on the left-hand side indicate the recursive subdivisions of the point set. The 2-d tree of the point set from the left-hand side is shown on the right-hand side.



Data Structure. We store the points of the x-2-d tree of $\mathcal{U}$ in an array $D[1..N]$, which is recursively defined as follows.

- $D[1]$ stores the root of the x-2-d tree of $\mathcal{U}$, and
- If $D[i]$ stores the root of an x/y-2-d tree of $P = \{(x_1, y_1), ..., (x_n, y_n)\}$, then $D[i] = \langle p, w(p), w_{\mathsf{min}} \rangle$, where
  - $w_{\mathsf{min}} = \min\{w(q) \mid q \in P\}$ is the minimum weight of any point in P,
  - $p = (x_m, y_m)$ is the median point in P, i.e, we have $|P_<| = \lceil n/2 \rceil$, where $P_< = \{(x, y) \in P \mid x < x_m\}$ and $P_> = \{(x, y) \in P \mid x > x_m\}$, and
  - $l = i + 1$ is the index in D of its left child,
  - $r = i + |P_<| + 1$ is the index in D of its right child,
  - $D[l]$ stores the root of the y/x-2-d tree of $P_<$ if $n > 1$, and
  - $D[r]$ stores the root of the y/x-2-d tree of $P_>$ if $P_> \neq \varnothing$.

Size. At each node, we store 4 integers in the range $[1, N]$, hence we need $\mathcal{O}(N)$ space. More precisely, we need $16 \cdot N$ bytes if $N < 2^{32}$, and $32 \cdot N$ bytes, else.

Construction. If we use the *quickselect* [23] algorithm to find the median point in a point set in $\mathcal{O}(N)$ time, then the definition of the array D yields an $\mathcal{O}(N \log N)$ time and space construction algorithm for D.

Queries. Algorithm 24 shows the query algorithm. Let $\mathcal{T}$ be a subtree in the overall 2-d-tree, let $D[b..e]$ be the range that stores exactly the nodes in $\mathcal{T}$ (due to the construction, such interval $[b, e]$ exists for every possible subtree), and let $v =$ **true** iff the root of $\mathcal{T}$ splits the point set vertically. Then, query-KDT$(\mathcal{Q}, w', b, e, v)$ either returns a point stored in $\mathcal{T}$ that lies in $\mathcal{Q}$ and has weight $\leqslant w'$ if it exists, or returns $\perp$, else.

Algorithm 24: query-KDT$(\mathcal{Q}, w', b, e, v)$

```
1  if D[b].p ∈ Q ∧ D[b].w ⩽ w' then
2      return D[b].p;
3  l ← b + 1;
4  r ← l + ⌈(e − b)/2⌉;
5  if e > b ∧ D[l].w_min ⩽ w' ∧ (v ∧ x_1 < D[b].p.x ∨ ¬v ∧ y_1 < D[b].p.y) then
6      p ← query-KDT(Q, w', l, r, ¬v);
7      if p ≠ ⊥ then
8          return p;
9  if e − b > 1 ∧ D[r].w_min ⩽ w' ∧ (v ∧ x_2 > D[b].p.x ∨ ¬v ∧ y_2 > D[b].p.y) then
10     p ← query-KDT(Q, w', r, e, ¬v);
11     if p ≠ ⊥ then
12         return p;
13 return ⊥;
```

Due to the construction, the root of $\mathcal{T}$ is stored in $D[b]$, its left and right children are stored in $D[l]$ and $D[r]$, where $l = b+1$ and $r = l + \lceil (e-b)/2 \rceil$. If $D[b].p$ lies in $\mathcal{Q}$ and has weight $D[b].w \leqslant w'$, then we return it. Else, we recursively search in the left subtree iff

- $D[b]$ has a left child, i.e, $e > b$, and
- not all points in the left subtree are heavier than w', i.e, $D[l].w_{\mathsf{min}} \leqslant w'$, and
- either $D[b]$ splits the point set vertically, and not all points to the left of $D[b].p.x$ lie to the left of $\mathcal{Q}$, i.e, $v \wedge x_1 < D[b].p.x$, or
- $D[b]$ splits the point set horizontally, and not all points below $D[b].p.y$ lie below $\mathcal{Q}$, i.e, $\neg v \wedge y_1 < D[b].p.y$.

If all of those criteria are met (see line 5), then we recursively search in the left subtree $D[l, r]$ (see line 6). If query-KDT$(\mathcal{Q}, w', l, r, \neg v)$ returns a point, then we return it. Else,

Data Structure	Problem	Constr./Prepr. Time	Space [bytes]
D-SG	IO-OROR	$\mathcal{O}(N + (N/m)^2)$	$8 \cdot N + 16 \cdot \lceil N/m \rceil^2$
SD-SG	IO-OROR	$\mathcal{O}(N + (N/m)^2)$	$8 \cdot N + 6 \cdot \lceil N/m \rceil^2$
SW-SG	SW-OROR	$\mathcal{O}(N + (N/m)^2)$	$12 \cdot N + 4 \cdot \lceil N/m \rceil^2$
SW-SS	SW-OROR	$\mathcal{O}(N \log s)$	$12 \cdot N$
SW-KDT	SW-OROR	$\mathcal{O}(N)$	$16 \cdot N$

Table 4.1: Construction/preprocessing times and space requirements of the OROR data structures.

Data Structure	Problem	Query Time
D-SG	IO-OROR	$\mathcal{O}(\lvert\mathcal{Q}\rvert/m^2 + N/m + m)$
SD-SG	IO-OROR	$\mathcal{O}(\lvert\mathcal{Q}\rvert/m^2 + N/m + m)$
SW-SG	SW-OROR	$\mathcal{O}(\lvert\mathcal{Q}\rvert/m^2 + N/m + m + \sqrt{\lvert\mathcal{Q}\rvert})$
SW-SS	SW-OROR	$\mathcal{O}((\bar{x}/s) \log s + s + \sqrt{\lvert\mathcal{Q}\rvert})$
SW-KDT	SW-OROR	$\mathcal{O}(\sqrt{N})$

Table 4.2: Query times of the OROR data structures.

we potentially query the right subtree (see lines 9-12). Finally, we return $\perp$ in line 13, if we have not found a point. To query the overall tree, we call $\mathsf{query\text{-}KDT}(\mathcal{Q}, w', 1, N, \mathbf{true})$.

4.5.12 Example. Let us continue Example 4.5.11.

$$\begin{array}{rcccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ D = & p_{10} & p_4 & p_5 & p_2 & p_3 & p_1 & p_8 & p_7 & p_6 & p_9 \end{array}$$

We have $w(p_7) = 1$ and $w(p) = \infty$ for $p \in P \backslash \{p_7\}$. To query the 2-d tree with $\mathcal{Q} = [x_1, x_2] \times [y_1, y_2]$ and query weight $w' = 1$, we call $\mathsf{query\text{-}KDT}(\mathcal{Q}, 1, 1, 10, \mathbf{true})$. Since $D[b].p = p_{10} \notin \mathcal{Q}$, we do not return p_{10} in line 2. Then, we set $l = 2$ and $r = 7$ in lines 3-4. Since $D[l].w_{\mathsf{min}} = \infty \nleqslant 1 = w'$, the **if**-clause in line 5 fails. However, we have $e - b = 9 > 1$, $D[r].w_{\mathsf{min}} = 0 \leqslant 1 = w'$ and $v \wedge x_2 > D[b].p.x$, hence we recursively call $\mathsf{query\text{-}KDT}(\mathcal{Q}, 1, 7, 10, \mathbf{false})$. There, we continue to search only in the left subtree, i.e, call $\mathsf{query\text{-}KDT}(\mathcal{Q}, 1, 8, 9, \mathbf{true})$. Now, we have $D[b].p = p_7 \in \mathcal{Q}$ and $D[b].w = 1 \leqslant 1 = w'$, so we return $D[b].p = p_7$.

Query Time. Let v be the number of visited nodes during the query. We have $v = \mathcal{O}(\sqrt{N})$ [1]. Since each recursive call visits exactly one of these visited nodes, there are $\leqslant v = \mathcal{O}(\sqrt{N})$ recursive calls. Each recursive call takes $\mathcal{O}(1)$ time. Thus, one query takes $\mathcal{O}(\sqrt{N})$ time.

Table 4.1 and Table 4.2 give an overview of the OROR data structures.

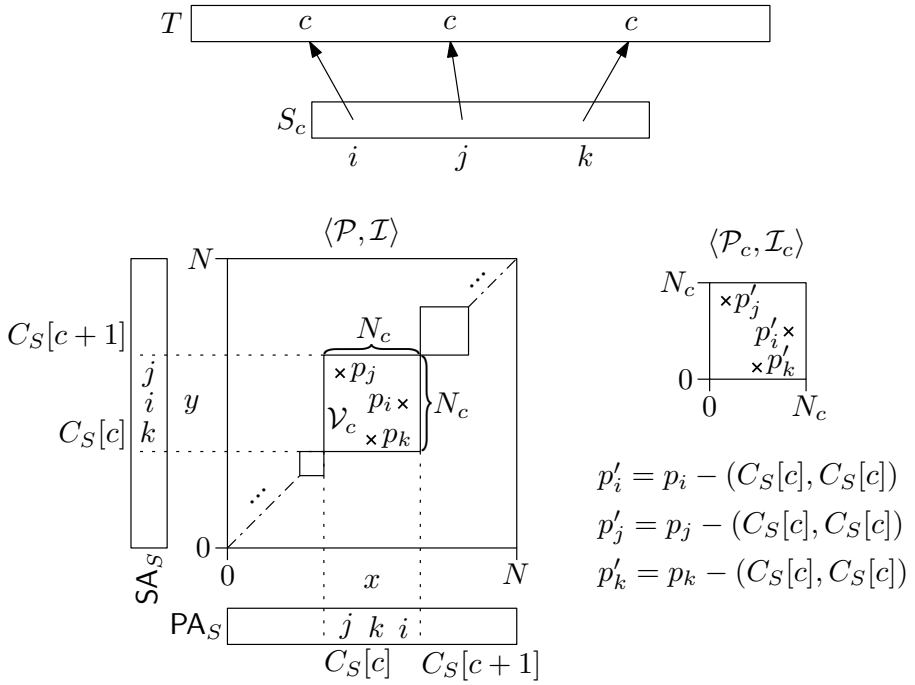


Figure 4.13: Illustration of an instance $\langle \mathcal{P}_c, \mathcal{I}_c \rangle$ from Definition 4.5.13, where $N_c = C_S[c+1] - C_S[c]$. The left-hand side shows the range $\mathcal{V}_c$, which contains all points $p = (x, y)$ with $T[S[\mathsf{PA}_S[x]]] = T[S[\mathsf{SA}_S[y]]] = c$. Note that its center lies on the diagonal. On the left-hand side, the instance $\langle \mathcal{P}_c, \mathcal{I}_c \rangle$ is shown.

4.5.4.3 Decomposed Orthogonal Range Reporting

The instance $\langle \mathcal{P}, \mathcal{I} \rangle$ of the IO-OROR problem from Observation 3.2.3 has the property that we can divide the overall point range $[1, N] \times [1, N]$ into σ smaller ranges such that each point is contained in one of those ranges, and the points contained in each one of these ranges form a new instance of the IO-OROR problem (see Lemma 4.5.14). Similarly, the range of each query issued in Theorem 3.2.9 is fully contained in one of those σ ranges (see Section 4.5.5.6). Combining these two insights allows us to reduce $\langle \mathcal{P}, \mathcal{I} \rangle$ to σ smaller instances of the IO-OROR problem (see Theorem 4.5.15). We at first formally show how this can be done, and then explain why it is valuable in practice.

4.5.13 Definition. Let $C_S[1..\sigma + 1]$ be the *sparse C-array* w.r.t. S, where $C_S[c] = \{i \in [1, |S|] \mid T[S[i]] < c\}$. Let $S_1, ..., S_\sigma$ be arrays, where $|S_c| = C_S[c + 1] - C_S[c]$ and $S_c[i] = \mathsf{select}_c(S, i)$. Let $\mathcal{P}_1, ..., \mathcal{P}_\sigma$ be point sets, where $\mathcal{P}_c = \{(\mathsf{PA}_S^{-1}[i] - C_S[c], \mathsf{SA}_S^{-1}[i] - C_S[c]) \mid i \in S_c\}$. Let $\mathcal{I}_1, ..., \mathcal{I}_\sigma$ be sequences, where $\mathcal{I}_c = (\mathsf{PA}_S^{-1}[S_c[1]] - C_S[c], \mathsf{SA}_S^{-1}[S_c[1]] - C_S[c]), ..., (\mathsf{PA}_S^{-1}[S_c[|S_c|]] - C_S[c], \mathsf{SA}_S^{-1}[S_c[|S_c|]] - C_S[c])$. Finally, let $\mathcal{V}_1, ..., \mathcal{V}_\sigma$ be ranges, where $\mathcal{V}_c = (C_S[c], C_S[c + 1]] \times (C_S[c], C_S[c + 1]]$.

4.5.14 Lemma. *Let $c \in \Sigma$ such that $S_c \neq \varnothing$. Then, $\langle \mathcal{P}_c, \mathcal{I}_c \rangle$ is an instance of the IO-OROR problem with fixed insertion order.*

Proof. $\langle \mathcal{P}_c, \mathcal{I}_c \rangle$ is an instance of the IO-OROR problem with fixed insertion order iff (**i**) there is a permutation π_c of $[1, N]$ for some integer $N \geqslant 1$ such that $\mathcal{P}_c = \{(i, \pi(i)) \mid i \in [1, N]\}$, and (**ii**) $\mathcal{I}_c$ contains distinct points from $\mathcal{P}_c$.

Let $\pi_c(\mathsf{PA}_S^{-1}[i] - C_S[c]) = \mathsf{SA}_S^{-1}[i] - C_S[c]$ for each $i \in S_c$, and let $[b, e] = \mathsf{siv}_S(c)$, i.e, we have $[b, e] = (C_S[c], C_S[c+1]]$. Since $\mathsf{PA}_S[b, e]$ and $\mathsf{SA}_S[b, e]$ contain exactly the values S_c, we have

$$\begin{aligned} &\{\mathsf{SA}_S^{-1}[i] \mid i \in S_c\} = \{\mathsf{PA}_S^{-1}[i] \mid i \in S_c\} = [b, e] \\ \Leftrightarrow &\{\mathsf{SA}_S^{-1}[i] - C_S[c] \mid i \in S_c\} = \{\mathsf{PA}_S^{-1}[i] - C_S[c] \mid i \in S_c\} = [1, |S_c|], \end{aligned} \tag{4.20}$$

hence (**i**) holds with $N = |S_c|$. Since SA_S^{-1} and PA_S^{-1} are permutations, and $(\mathsf{PA}_S^{-1}[i] - C_S[c], \mathsf{SA}_S^{-1}[i] - C_S[c]) \in \mathcal{P}_c$ for each $i \in S_c$, it holds (**ii**). □

4.5.15 Theorem. *If each query rectangle $\mathcal{Q}$ in the instance $\langle \mathcal{P}, \mathcal{I} \rangle$ from Observation 3.2.3 is annotated with the character $c \in \Sigma$ such that $\mathcal{Q} \subseteq \mathcal{V}_c$, then we can solve $\langle \mathcal{P}, \mathcal{I} \rangle$ by solving $\langle \mathcal{P}_1, \mathcal{I}_1 \rangle, ..., \langle \mathcal{P}_\sigma, \mathcal{I}_\sigma \rangle$.*

Proof. Let $\mathcal{R}_1, ..., \mathcal{R}_\sigma$ be data structures, where $\mathcal{R}_c$ solves $\langle \mathcal{P}_c, \mathcal{I}_c \rangle$.

Invariant. We maintain that after the i-th insertion in $\langle \mathcal{P}, \mathcal{I} \rangle$, $\mathcal{R}_c$ holds exactly the points $\mathcal{A}_c = \{(\mathsf{PA}_S^{-1}[j] - C_S[c], \mathsf{SA}_S^{-1}[j] - C_S[c]) \mid j \in S_c \cap [1, i]\}$ for each $c \in [1, \sigma]$.

Insertions. Given the next point $(\mathsf{PA}_S^{-1}[i], \mathsf{SA}_S^{-1}[i])$ to insert, we instead insert $(\mathsf{PA}_S^{-1}[i] - C_S[c], \mathsf{SA}_S^{-1}[i] - C_S[c])$ into $\mathcal{R}_c$, where $c = T[S[i]]$. This maintains the invariant.

Queries. Let i be the number of already inserted points in $\langle \mathcal{P}, \mathcal{I} \rangle$, i.e, $\mathcal{R}$ holds the points $\mathcal{A} = \{(\mathsf{PA}_S^{-1}[j], \mathsf{SA}_S^{-1}[j]) \mid j \in [1, i]\}$. Suppose we receive a query with the character $c \in \Sigma$ and the rectangle $\mathcal{Q} = [x_1, x_2] \times [y_1, y_2] \subseteq V_c$, and let

$$\mathcal{Q}_c = [x_1', x_2'] \times [y_1', y_2'] = [x_1 - C_S[c], x_2 - C_S[c]] \times [y_1 - C_S[c], y_2 - C_S[c]]. \tag{4.21}$$

By the invariant, we have

$$\begin{aligned} \forall j \in [1, i] : &(\mathsf{PA}_S^{-1}[j], \mathsf{SA}_S^{-1}[j]) \in \mathcal{A} \cap \mathcal{Q} \\ \Leftrightarrow &(\mathsf{PA}_S^{-1}[j] - C_S[c], \mathsf{SA}_S^{-1}[j] - C_S[c]) \in \mathcal{A}_c \cap \mathcal{Q}_c. \end{aligned} \tag{4.22}$$

□

Therefore, we can answer this query by querying $\mathcal{R}_c$ with $\mathcal{Q}_c$ (see Algorithm 25). In lines 1-2, we compute $\mathcal{Q}_c$ from $\mathcal{Q}$. If $\mathcal{R}_c$ returns a point $p = (\tilde{x}, \tilde{y})$, then we can return $(\tilde{x} + C_S[c], \tilde{y} + C_S[c])$ by Equation (4.22). Else (if $p = \perp$), then the query with rectangle $\mathcal{Q}$ to $\mathcal{R}$ would have also returned $\perp$ by Equation (4.22), hence we correctly return $\perp$.

4.5.16 Example. Consider the sampling from Example 3.2.11.

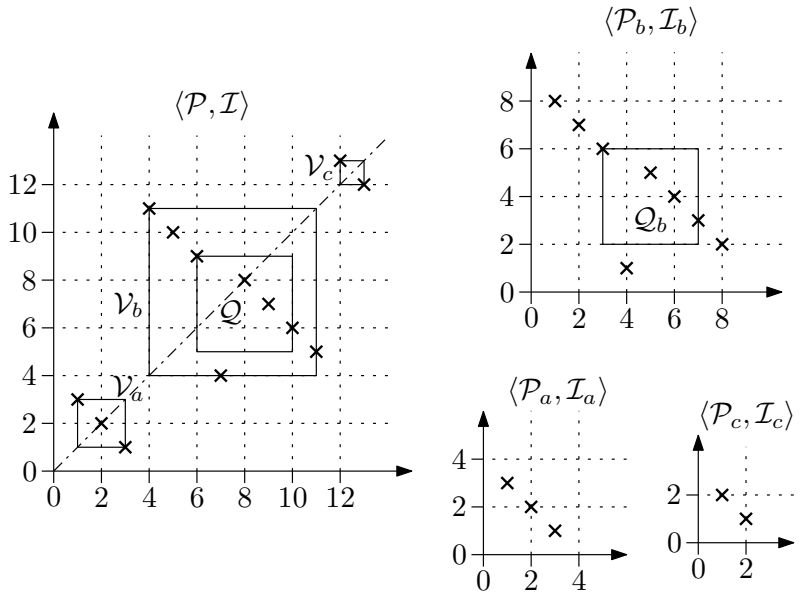
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
$T =$	a	b	c	a	b	c	a	b	b	b	b	b	b	b	b	b	b	b	a	b	c	a	b	c	a	b

Algorithm 25: $\mathcal{R}.\mathsf{query}(c, [x_1, x_2], [y_1, y_2])$

$[x'_1, x'_2] \leftarrow [x_1 - C_S[c], x_2 - C_S[c]]$;
$[y'_1, y'_2] \leftarrow [y_1 - C_S[c], y_2 - C_S[c]]$;
$p = (\tilde{x}, \tilde{y}) \leftarrow \mathcal{R}_c.\mathsf{query}([x'_1, x'_2], [y'_1, y'_2])$;
if $p \neq \perp$ **then**
    $(\tilde{x}, \tilde{y}) \leftarrow (\tilde{x} + C_S[c], \tilde{y} + C_S[c])$;
return p;

	1	2	3	4	5	6	7	8	9	10	11	12	13
$\mathcal{C} =$	1	2	3	4	5	8	9	13	17	18	21	25	26
$\mathsf{PA}_{\mathcal{C}} =$	1	4	12	2	5	6	13	7	8	9	10	3	11
$\mathsf{PA}_{\mathcal{C}}^{-1} =$	1	4	12	2	5	6	8	9	10	11	13	3	7
$\mathsf{SA}_{\mathcal{C}} =$	12	4	1	13	10	9	8	7	6	5	2	11	3
$\mathsf{SA}_{\mathcal{C}}^{-1} =$	3	11	13	2	10	9	8	7	6	5	12	1	4

We have $C_{\mathcal{C}} = [0, 3, 11, 13]$, $S_a = [1, 4, 25]$, $S_b = [2, 5, 8, 9, 13, 17, 18, 26]$, $S_c = [3, 21]$, $\mathcal{V}_a = [1, 3] \times [1, 3]$, $\mathcal{V}_b = [4, 11] \times [4, 11]$, $\mathcal{V}_c = [12, 13] \times [12, 13]$, $\mathcal{I}_a = \langle (1, 3), (2, 2), (3, 1) \rangle$, $\mathcal{I}_b = \langle (1, 8), (2, 7), (3, 6), (4, 1), (5, 5), (6, 4), (7, 3), (8, 2) \rangle$ and $\mathcal{I}_c = \langle (1, 2), (2, 1) \rangle$. The instances $\langle \mathcal{P}_a, \mathcal{I}_a \rangle$, $\langle \mathcal{P}_b, \mathcal{I}_b \rangle$ and $\langle \mathcal{P}_c, \mathcal{I}_c \rangle$ are shown below.



Suppose we have already inserted all points, and consider the query $\mathcal{Q} = [6, 10] \times [5, 9]$, which is annotated with the character b, i.e, it holds $\mathcal{Q} \subseteq \mathcal{V}_b = [4, 11] \times [4, 11]$. Then, we have $\mathcal{Q}_b = [x_1 - C_{\mathcal{C}}[b], x_2 - C_{\mathcal{C}}[b]] \times [y_1 - C_{\mathcal{C}}[b], y_2 - C_{\mathcal{C}}[b]] = [6-3, 10-3] \times [5-3, 9-3] = [3, 7] \times [2, 6]$. The query $\mathcal{Q}_b$ to $\langle \mathcal{P}_b, \mathcal{I}_b \rangle$ can be answered with $p = (5, 5)$. To answer the query $\mathcal{Q}$ to $\langle \mathcal{P}, \mathcal{I} \rangle$, we return $(5 + C_{\mathcal{C}}[b], 5 + C_{\mathcal{C}}[b]) = (8, 8) \in \mathcal{Q}$.

Now, we discuss why solving $\langle \mathcal{P}_1, \mathcal{I}_1 \rangle, ..., \langle \mathcal{P}_\sigma, \mathcal{I}_\sigma \rangle$ instead of $\langle \mathcal{P}, \mathcal{I} \rangle$ is valuable in practice. If we assume that the characters in T are evenly distributed across the samples, i.e, we have $C_S[c] = (c-1)|S|/\sigma$ for each $c \in \Sigma$, then the overall point range

$$\left| \bigcup_{c \in \Sigma} \mathcal{V}_c \right| = \bigcup_{c \in \Sigma} |\mathcal{V}_c| = \bigcup_{c \in \Sigma} |(C_S[c], C_S[c+1]] \times (C_S[c], C_S[c+1]]| \tag{4.23}$$

$$= \bigcup_{c \in \Sigma} (C_S[c+1] - C_S[c])^2 \tag{4.24}$$

$$= (c \cdot |S|/\sigma - (c-1) \cdot |S|/\sigma)^2 = (|S|/\sigma)^2 \tag{4.25}$$

of the instances $\langle \mathcal{P}_1, \mathcal{I}_1 \rangle, ..., \langle \mathcal{P}_\sigma, \mathcal{I}_\sigma \rangle$ is

$$\frac{(|S|/\sigma)^2}{|[1, |S|] \times [1, |S|]|} = \frac{(|S|/\sigma)^2}{|S|^2} = \sigma^2 \tag{4.26}$$

times smaller than the point range $[1, |S|] \times [1, |S|]$ of the instance $\langle \mathcal{P}, \mathcal{I} \rangle$. In practice ($\sigma \leqslant 256$), this reduces the overall point range by a factor up to $\sigma^2 = 256^2 = 65536$. Conversely, this implies that the points can be distributed very unevenly in $\langle \mathcal{P}, \mathcal{I} \rangle$.

With grid-based data structures, this enables us to either decrease the cell width m while not increasing the size of the data structure, or reducing the size of the data structure while leaving m unchanged.

4.5.5 Algorithm

Now, we discuss our optimized implementation of the exact LZ77 algorithm from Section 3.2. We start by showing how the size of the sample set $\mathcal{C}$ can be reduced in practice (see Section 4.5.5.1). Then, we discuss the framework algorithm in Section 4.5.5.3. Finally, we employ the previously discussed optimizations in the algorithm for computing a perfect phrase (see Sections 4.5.5.5 and 4.5.5.6).

4.5.5.1 Sample-Set Construction

Again, let $\mathcal{F} = \langle s_1, l_1\rangle, ..., \langle s_{z'}, l_{z'}\rangle$ be the LZ-like factorization resulting from Theorem 3.1.8. Instead of constructing $\mathcal{C}$ as in Lemma 3.2.7, we construct $\mathcal{C}$ with one scan over $\mathcal{F}$ from left to right (see Algorithm 26). Since we add at most $\lceil n/\delta \rceil$ samples to $\mathcal{E}$ in order to achieve δ-density, $\mathcal{C}$ cannot exceed size $|\mathcal{F}| + \lceil n/\delta \rceil$. Thus, initializing $\mathcal{C}$ as an array of this size suffices (see line 1).

Now, we iterate with i over $\mathcal{C}$ and with j over $\mathcal{F}$. We maintain that after line 4 of the j-th iteration (for $j > 1$), p is the $j-1$-th and e is the j-th largest sample in $\mathcal{E}$. If p and e have distance $> \delta$, then we add samples $p + \delta$, $p + 2\delta$,... until the last added sample has distance $\leqslant \delta$ to e (see lines 5-8). Finally, we add e to $\mathcal{C}$ (see line 9). Since the last added sample is $\mathcal{C}[i-1]$ in line 11, we return $\mathcal{C}[1, i)$.

Since each 2 consecutive samples have distance $\leqslant \delta$, and we have added every sample from $\mathcal{E}$, $\mathcal{C}$ is δ-dense and leftmost-substring-covering. The algorithm runs in $\Theta(|\mathcal{F}|+n/\delta) = \Theta(z + n/(n/z)) = \Theta(z)$ time and uses $\mathcal{O}(1)$ additional space.

This algorithm produces a $\approx 30\%$ smaller sampling than Lemma 3.2.7 in practice, because we add extra samples between 2 consecutive samples of $\mathcal{E}$ only if they have distance $< \delta$.

Algorithm 26: build-C()

```
1  C ← new array[1..|F| + ⌈n/δ⌉];
2  i ← 1; p ← 0;
3  for j from 1 to |F| do
4      e ← p + max(1, l_j);
5      while e − p > δ do
6          p ← p + δ;
7          C[i] ← p;
8          i ← i + 1;
9      C[i] ← e;
10     p ← e;
11 return C[1, i);
```

Algorithm 27: lz77-exact()

```
1  i ← 1;
2  v ← 0;
3  while i ⩽ n do
4      if R.is-dynamic() then
5          while v < N ∧ C[v + 1] < i do
6              v ← v + 1;
7              R.insert(PA_C^{-1}[v], SA_C^{-1}[v]);
8      ⟨s, l⟩ ← perfect-factor(i, v);
9      i ← i + max(1, l);
10     output ⟨s, l⟩;
```

4.5.5.2 Auxiliary Data Structures

4.5.5.2.1 Orthogonal Range One-Reporting Data Structure We use Theorem 4.5.15 to split the instance $\langle \mathcal{P}, \mathcal{I} \rangle$ up into σ instances $\langle \mathcal{P}_1, \mathcal{I}_1 \rangle, ..., \langle \mathcal{P}_\sigma, \mathcal{I}_\sigma \rangle$, and solve each of those using one of the data structures presented in Section 4.5.4. If this is a data structure for the IO-OROR problem (see Section 4.5.4.1), then the function $\mathcal{R}.\mathsf{is\text{-}dynamic}()$ returns **true**. Similarly, $\mathcal{R}.\mathsf{is\text{-}static}()$ returns **true** iff this is a data structure for the SW-OROR problem (see Section 4.5.4.2).

4.5.5.2.2 Sparse Suffix Array Interval Samples We use Theorem 4.5.7 for the sample-set $\mathcal{C}$, implement $\mathcal{D}$ with Theorem 4.5.3, and set $\theta = 2|\mathcal{C}|$ and $\lambda_{\mathsf{max}} = \lfloor n/z \rfloor$. If we set the sampling rate in Theorem 4.5.3 to $s = \Theta(n/z)$, then the construction takes $\mathcal{O}(n/\log_\sigma n + |\mathcal{C}| \log |\mathcal{C}| + n + 2|\mathcal{C}| \log((n/(n/z)) + n/z)) = \mathcal{O}(n + z \log z)$ time and $\mathcal{O}(n/\log_\sigma n + 2|\mathcal{C}| + n/(n/z)) = \mathcal{O}(z)$ space. In practice, however, we set $s = 64$ if $n/|\mathcal{C}| \geqslant 256$. Else, we do not implement Theorem 4.5.3, and instead compute the fingerprints naively. Note that during the construction of Theorem 4.5.3 and the computation of the exact LZ77 factorization, we compute fingerprints with length $\lessapprox n/z \approx n/|\mathcal{C}| \leqslant 256$. In this case, computing fingerprints naively is faster than using Theorem 4.5.3, even when decreasing the sample rate s below 64. We implement Lemma 4.5.2 in any case, in order to concatenate fingerprints. Let Λ_{suf} be the set of pattern lengths resulting from Theorem 4.5.7.

4.5.5.2.3 Sparse Prefix Array Interval Samples In practice, we set $\delta = \min(256, n/z)$, because increasing δ beyond 256 does neither decrease the running time, nor the peak memory usage (when setting $\tau = 512$). We also implement Theorem 4.5.7 for sparse prefix array intervals w.r.t. $\mathcal{C}$. Since we only compute sparse prefix array intervals of length-$\leqslant \delta$ substrings of T, we set $\lambda_{\mathsf{max}} = \delta$. We use the data structure from Paragraph 4.5.5.2.2 for computing fingerprints. Let Λ_{pref} be the set of resulting pattern lengths.

4.5.5.3 Framework Algorithm

Algorithm 27 shows the framework algorithm. We maintain the starting position i in T of the next phrase to compute, and the index v of the last sample $\mathcal{C}[v]$ before i (or $v = 0$ if no such sample exists). Additionally, we store a globally defined array $\Phi[1..\delta]$ for temporarily storing fingerprints. In lines 5-7, we maintain the invariant from Theorem 3.2.9 that $\mathcal{R}$ holds exactly the points $\{(\mathsf{PA}_\mathcal{C}^{-1}[1], \mathsf{SA}_\mathcal{C}^{-1}[1]), ..., (\mathsf{PA}_\mathcal{C}^{-1}[v], \mathsf{SA}_\mathcal{C}^{-1}[v])\}$. However, we only do this if $\mathcal{R}$ is dynamic (see Section 4.5.4.1). If $\mathcal{R}$ is static, then we instead dynamically adjust the query weight for the queries to $\mathcal{R}$ (which is equivalent to inserting points into and removing points from $\mathcal{R}$ if it is dynamic, see Lemma 4.5.9). This also eliminates the need to consider close sources separately. We will explain this in more detail in Section 4.5.5.6. As

in Algorithm 11, we iteratively compute perfect factors (see line 8) until we have factorized T completely.

4.5.5.4 Computing a Perfect Factor

Algorithm 28 summarizes the computation of a perfect factor. As in Algorithm 10, we initialize $\langle s, l\rangle$ as a literal factor for position i in line 1. s, l and i are globally defined variables within $\mathsf{perfect\text{-}factor}(i, v)$.

4.5.5.4.1 Close Sources If $\mathcal{R}$ is dynamic, then we consider each close source $p \in [i-\delta, i)$ in lines 3-7 as in Algorithm 10. If $\mathcal{R}$ is static, then we instead consider all close sources during Paragraph 4.5.5.4.2.

4.5.5.4.2 Far Sources

4.5.17 Definition. Let $i \in [1, n]$, $t \in [1, n-i+1]$, $k \in [1, \min(\delta, n-i+1)]$, $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k))$ and $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(T[i+k-1, i+t))$. Let $v \in [0, |\mathcal{C}|]$ such that either $v = 0$ if $\mathcal{C}[1] > i$, or $v \geqslant 1$ is maximal such that $\mathcal{C}[v] < i$. Then, we say that $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$ intersect before i iff $\mathsf{PA}_{\mathcal{C}}[x_1, x_2] \cap \mathsf{SA}_{\mathcal{C}}[y_1, y_2] \cap [1, v] \neq \varnothing$.

Here, we want to compute t' and a source $s' < i$ of $T[i, i+t')$, where t' is the maximum t such that there are k, $[x_1, x_2]$ and $[y_1, y_2]$ as defined in Definition 4.5.17. $\mathsf{extend\text{-}right}(k, [x_1, x_2])$ (see Algorithm 29) computes, given some k and $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k)) \neq \varnothing$, the maximum t such that $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$ intersect before i. If such t exists, then it also finds a source $s' < i$ for $T[i, i+t)$, and updates $\langle s, l\rangle \leftarrow \langle s', t\rangle$. Therefore, calling $\mathsf{extend\text{-}right}(k, [x_1, x_2])$ for each possible k (with $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k))$) ensures $l = t'$ and that s is a source of $T[i, i+t')$. We will discuss Algorithm 29 in the next section.

To speed up the computation of the prefix array intervals, we compute the fingerprint of each possible head, and store it in $\Phi[1..\delta]$, i.e, $\Phi[k] = \phi(i, i+k-1)$ if $i+k-1 \leqslant n$. By Definition 2.6.1, we have $\Phi[1] = \phi(i, i) = T[i]$, and for each $k \in [2, \min(\delta, n-i+1)]$, we have

$$\begin{aligned}
\Phi[k] &= \phi(i, i+k-1) && (4.27)\\
&= \left(\sum_{j=i}^{i+k-1} T[k] \cdot b^{i+k-1-j}\right) \bmod q && (4.28)\\
&= \left(\left(\sum_{j=i}^{i+k-2} T[j] \cdot b^{i+k-2-j}\right) \cdot b + T[i+k-1] \cdot b^0\right) \bmod q && (4.29)\\
&= (\phi(i, i+k-2) \cdot b + T[i+k-1]) \bmod q && (4.30)\\
&= (\Phi[k-1] \cdot b + T[i+k-1]) \bmod q, && (4.31)
\end{aligned}$$

Algorithm 28: perfect-factor(i, v)

$\langle s, l \rangle = \langle 0, T[i] \rangle$;
if R.is-dynamic() **then**
  for p **from** $\max(1, i - \delta)$ **to** $i - 1$ **do**
    $\mathsf{lce}_p \leftarrow \mathsf{LCE}(p, i)$;
    if $\mathsf{lce}_p > l$ **then**
      $s \leftarrow p$;
      $l \leftarrow \mathsf{lce}_p$;
$\Phi[1] \leftarrow T[i]$;
for k **from** 2 **to** $\min(\delta, n - i + 1)$ **do**
  $\Phi[k] \leftarrow (\Phi[k-1] \cdot b + T[i + k - 1]) \bmod q$;
for $k \in \Lambda_{\mathsf{pref}}$ **do**
  $[x_1, x_2] \leftarrow \mathsf{H}(\mathsf{PIV}_{\mathcal{C}}^{k})[\Phi[k]]$;
  if $[x_1, x_2] \neq \varnothing$ **then**
    extend-right$(k, [x_1, x_2])$;
for k **from** 1 **to** $\min(\delta, n - i + 1)$ **do**
  if $k \in \Lambda_{\mathsf{pref}}$ **then**
    continue;
  $\gamma \leftarrow \max\{\lambda \in \Lambda_{\mathsf{pref}} \cup \{0\} \mid \lambda < k\}$;
  $\xi \leftarrow \min\{\lambda \in \Lambda_{\mathsf{pref}} \cup \{\infty\} \mid \lambda > k\}$;
  if $\gamma = 0$ **then**
    $[x_1^{\gamma}, x_2^{\gamma}] \leftarrow [1, |\mathcal{C}|]$;
  else
    $[x_1^{\gamma}, x_2^{\gamma}] \leftarrow \mathsf{H}(\mathsf{PIV}_{\mathcal{C}}^{\gamma})[\Phi[\gamma]]$;
  if $\xi = \infty$ **then**
    $[x_1, x_2] \leftarrow$ spa-interval$([i, i + k), [x_1^{\gamma}, x_2^{\gamma}])$;
  else
    $[x_1^{\xi}, x_2^{\xi}] \leftarrow \mathsf{H}(\mathsf{PIV}_{\mathcal{C}}^{\xi})[\Phi[\xi]]$;
    $[x_1, x_2] \leftarrow$ interpolate-spa$([i, i + k), \langle [x_1^{\gamma}, x_2^{\gamma}], \gamma \rangle, \langle [x_1^{\xi}, x_2^{\xi}], \xi \rangle)$;
  if $[x_1, x_2] \neq \varnothing$ **then**
    extend-right$(k, [x_1, x_2])$;
return $\langle s, l \rangle$;

hence we can compute $\Phi[1..\delta]$ from left to right as in lines 8-10. This takes $\mathcal{O}(\delta)$ time.

Sampled Heads. At first, we consider each $k \in [1, \delta] \cap \Lambda_{\mathsf{pref}}$. With $\Phi[1..\delta]$, we can look up $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k))$ for each such k by $[x_1, x_2] \leftarrow \mathsf{H}(\mathsf{PIV}^k_{\mathcal{C}})[\Phi[k]]$ in $\mathcal{O}(1)$ expected time (see line 12). If $[x_1, x_2] \neq \varnothing$, then we call $\mathsf{extend\text{-}right}(k, [x_1, x_2])$ in line 14.

Unsampled Heads. In lines 15-29, we consider the remaining values $k \in [1, \delta] \backslash \Lambda_{\mathsf{pref}}$. For each such k, we cannot use a hash table to directly look up $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k))$. However, we can still use the hash tables to speed up its computation. We compute the maximum sampled pattern length $\gamma \in \Lambda_{\mathsf{pref}}$ that is smaller than k if it exists (else $\gamma = 0$, see line 19), and the minimum sampled pattern length $\xi \in \Lambda_{\mathsf{pref}}$ that is greater than k if it exists (else $\xi = \infty$, see line 19). Now, we compute $[x_1^\gamma, x_2^\gamma] = \mathsf{piv}_{\mathcal{C}}(T[i, i+\gamma))$. If $\gamma = 0$, then $[x_1^\gamma, x_2^\gamma] = [1, |\mathcal{C}|]$ (see line 21). Else, we compute it by $[x_1^\gamma, x_2^\gamma] \leftarrow \mathsf{H}(\mathsf{PIV}^\gamma_{\mathcal{C}})[\Phi[\gamma]]$ (see line 23).

If $\xi \neq \infty$, then $\gamma < k < \xi$ implies $[x_1^\xi, x_2^\xi] \subseteq [x_1, x_2] \subseteq [x_1^\gamma, x_2^\gamma]$ by Observation 4.5.1. Thus, we can compute $[x_1, x_2]$ in line 28 as described in Section 4.5.1.1. Else (if $\xi = \infty$), then we compute $[x_1, x_2]$ using only $[x_1^\gamma, x_2^\gamma]$ in line 25 as described in Section 4.5.1. Finally, if $[x_1, x_2] \neq \varnothing$, then we call $\mathsf{extend\text{-}right}(k, [x_1, x_2])$ in line 30 analogously to line 14.

4.5.5.5 Exponential Search

Recall that $\mathsf{extend\text{-}right}(k, [x_1, x_2])$ (see Algorithm 29) computes, given k and $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k)) \neq \varnothing$, the maximum t such that $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$ intersect before i (see Definition 4.5.17). If such t exists, then it also finds a source $s' < i$ for $T[i, i+t)$, and updates $\langle s, l \rangle \leftarrow \langle s', t \rangle$. More precisely, we implicitly search for t by instead searching for the maximum possible tail length $t' = t - k + 1$. At first, we perform an exponential search over the sampled pattern lengths in Λ_{suf} in order to reduce the search interval for t' efficiently (Paragraph 4.5.5.5.1). Then, we search in the remaining interval (see Paragraph 4.5.5.5.2).

4.5.5.5.1 Search over Sampled Pattern Lengths Overall, we perform an exponential search for t' over $\Lambda'_{\mathsf{suf}} = \Lambda_{\mathsf{suf}} \cap [\lambda_{\mathsf{min}}, \lambda_{\mathsf{max}}]$. λ_{min} is either the maximum sampled pattern length that does not exceed $l - k + 2$ if it exists, or the smallest sampled pattern length overall, else (see line 1). Considering sampled pattern lengths shorter than λ_{min} would not be beneficial, because we have already found an LZ factor with length l. λ_{max} is the maximum sampled pattern length such that $T[m, m + \lambda_{\mathsf{max}})$ is a substring of T, where $m = i + k - 1$ is the end of the current head $T[i, i+k)$ and the start of each possible tail (see line 2).

Algorithm 29: extend-right$(k, [x_1, x_2])$

$m \leftarrow i + k - 1$;
$\lambda_{\mathsf{min}} \leftarrow \max(\min \Lambda_{\mathsf{suf}}, \max\{\lambda \in \Lambda_{\mathsf{suf}} \cup \{0\} \mid \lambda \leqslant l - k + 2\})$;
$\lambda_{\mathsf{max}} \leftarrow \max\{\lambda \in \Lambda_{\mathsf{suf}} \mid \lambda \leqslant n - m + 1\}$;
$\varrho, f_\varrho \leftarrow 0$;
$\zeta \leftarrow \infty$;
$[y_1^\varrho, y_2^\varrho] \leftarrow [1, |\mathcal{C}|]$;
$[y_1^\zeta, y_2^\zeta] \leftarrow \varnothing$;
exp-search for max $\lambda \in \Lambda_{\mathsf{suf}} \cap [\lambda_{\mathsf{min}}, \lambda_{\mathsf{max}}]$ **such that**
    $f' \leftarrow$ fp-substr$(m + \varrho, \lambda - \varrho)$;
    $f_\lambda \leftarrow$ concat-fps$(f_\varrho, f', \lambda - \varrho)$;
    $[y_1, y_2] \leftarrow \mathsf{H}(\mathsf{SIV}_{\mathcal{C}}^{\lambda})[f_\lambda]$;
    if $[y_1, y_2] \neq \varnothing$ **then**
        if intersect$(k, m, \lambda, [x_1, x_2], [y_1, y_2])$ **then**
            $[y_1^\varrho, y_2^\varrho] \leftarrow [y_1, y_2]$;
            $\varrho \leftarrow \lambda$;
            $f_\varrho \leftarrow f_\lambda$;
            report true;
        else
            $[y_1^\zeta, y_2^\zeta] \leftarrow [y_1, y_2]$;
    $\zeta \leftarrow \lambda$;
    report false;
if $\zeta = \infty$ **then**
    exp-search for max $\iota \in [\varrho, \zeta)$ **such that** extend-step;
else
    bin-search for max $\iota \in [\varrho, \zeta)$ **such that** extend-step;

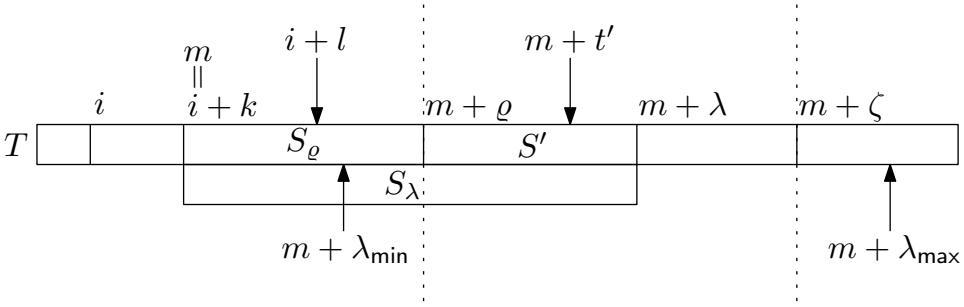


Figure 4.14: Illustration of a step in the search over Λ'_{suf}, where $\zeta \neq \infty$. The dotted lines indicate the current search interval $[\varrho, \zeta)$ for t'. In this case, we have $t' < \lambda$, hence we continue to search for t' in $[\varrho, \lambda)$.

Invariant. During the search over Λ'_{suf}, we maintain

$$\forall \lambda \in \Lambda'_{\mathsf{suf}} \cap [1, \varrho] : \lambda \leqslant t' \tag{4.32}$$

$$\forall \lambda \in \Lambda'_{\mathsf{suf}} \cap [\zeta, \infty] : \lambda > t' \tag{4.33}$$

$$f_\varrho = \phi(m, m + \varrho - 1) \tag{4.34}$$

$$[y_1^\varrho, y_2^\varrho] = \mathsf{siv}_{\mathcal{C}}(T[m, m + \varrho)) \neq \varnothing \tag{4.35}$$

$$[y_1^\zeta, y_2^\zeta] = \mathsf{siv}_{\mathcal{C}}(T[m, m + \zeta)) \tag{4.36}$$

Equations (4.32–4.33) state that the current search interval $[\varrho, \zeta)$ contains t'. Additionally, we maintain the fingerprint of $T[m, m + \varrho)$ (see Equation (4.34)) and the sparse suffix array intervals of $T[m, m + \varrho)$ and $T[m, m + \zeta)$ (see Equations (4.34–4.36)).

We aim to find ϱ_{max} and ζ_{min}, where ϱ_{max} is the maximum $\lambda \in \Lambda'_{\mathsf{suf}} \cup \{0\}$ such that Equation (4.32) holds. Similarly, ζ_{min} is the minimum $\lambda \in \Lambda'_{\mathsf{suf}} \cup \{\infty\}$ such that Equation (4.33) holds. Initializing the globally defined variables $\varrho, f_\varrho, \zeta, [y_1^\varrho, y_2^\varrho]$ and $[y_1^\zeta, y_2^\zeta]$ as in lines 4-7 does not violate Equations (4.32–4.36). During the search for ϱ_{max}, we increase ϱ and decrease ζ until $\varrho = \varrho_{\mathsf{max}}$ and $\zeta = \zeta_{\mathsf{min}}$.

Search. Let $\lambda \in \Lambda'_{\mathsf{suf}} \cap (\varrho, \zeta)$ be a candidate length during the search, let $S_\lambda = T[m, m + \lambda)$, $S_\varrho = T[m, m + \varrho)$ and $S' = T[m + \varrho, m + \lambda)$ (see Figure 4.14). We start by computing the fingerprint f' of S' (see line 9). Since $S_\lambda = S_\varrho S'$, we can compute the fingerprint f_λ of S_λ by concatenating f_ϱ and f' in line 10. With f_λ, we look up $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(S_\lambda)$ using $\mathsf{H}(\mathsf{SIV}_{\mathcal{C}}^\lambda)$ and f_λ (see line 11). If $[y_1, y_2] = \varnothing$, then $t' < \lambda$ and therefore $\varrho_{\mathsf{max}} < \lambda$, hence we report back to the search that $\varrho_{\mathsf{max}} \in [\varrho, \lambda)$ must hold (see line 21).

Then, we call $\mathsf{intersect}(k, m, \lambda, [x_1, x_2], [y_1, y_2])$ (see Algorithm 31), which returns **true** iff $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$ intersect before i. We will discuss Algorithm 31 in the next section. If $\mathsf{intersect}(k, m, \lambda, [x_1, x_2], [y_1, y_2])$ returns **true**, then $t' \geqslant \lambda$ and therefore $\varrho_{\mathsf{max}} \geqslant \lambda$, hence setting $\varrho \leftarrow \lambda$ $f_\varrho \leftarrow f_\lambda$ and $[y_1^\varrho, y_2^\varrho] \leftarrow [y_1, y_2]$ maintains the invariant, and we report back to the search that $\varrho_{\mathsf{max}} \in [\lambda, \zeta)$ must hold (see line 17). Else

($\mathsf{intersect}(k, m, \lambda, [x_1, x_2], [y_1, y_2])$ returns **false**), then $t' < \lambda$ and therefore $\varrho_{\mathsf{max}} < \lambda$, hence setting $[y_1^\zeta, y_2^\zeta] \leftarrow [y_1, y_2]$, $\zeta \leftarrow \lambda$ and $f_\zeta \leftarrow f_\lambda$ maintains the invariant, and we report back to the search that $\varrho_{\mathsf{max}} \in [\varrho, \lambda)$ must hold (see line 21). The functionality of the exponential search (see Section 2.1.3) ensures that we have $\varrho = \varrho_{\mathsf{max}}$ and $\zeta = \zeta_{\mathsf{min}}$ after line 21.

Algorithm 30: $\mathsf{extend\text{-}step}(\iota)$

```
1  if [y_1^ζ, y_2^ζ] = ∅ then
2      [y_1, y_2] ← ssa-interval([p, p + ι), [y_1^ϱ, y_2^ϱ]);
3  else
4      [y_1, y_2] ← interpolate-ssa([p, p + ι), ⟨[y_1^ϱ, y_2^ϱ], ϱ⟩, ⟨[y_1^ζ, y_2^ζ], ζ⟩);
5  if [y_1, y_2] ≠ ∅ ∧ intersect(k, m, ι, [x_1, x_2], [y_1, y_2]) then
6      ϱ ← ι;
7      [y_1^ϱ, y_2^ϱ] ← [y_1, y_2];
8      report true;
9  else
10     ζ ← ι;
11     [y_1^ζ, y_2^ζ] ← [y_1, y_2];
12     report false;
```

4.5.5.5.2 Remaining Search Equations (4.32) and (4.33) yield $\rho \leqslant t' < \zeta$. Thus, we can find t' by continuing to search in the interval $[\varrho, \zeta)$. Consider the cases where $t' > \max \Lambda'_{\mathsf{suf}}$ (and therefore $\zeta_{\mathsf{min}} = \infty$), but $t' - \max \Lambda'_{\mathsf{suf}} = \mathcal{O}(1)$ and $n - t' = \Theta(n)$. Here, a binary search over $[\varrho, \zeta)$ runs through $\mathcal{O}(\log n)$ iterations. However, an exponential search runs through only $\mathcal{O}(1)$ iterations. Therefore, we perform an exponential search in this case (see line 23). Conversely, if $\zeta_{\mathsf{min}} \neq \infty$, then the search over Λ'_{suf} in lines 8-21 enters the binary search phase before terminating (see Section 2.1.3), and we want to continue the last used search scheme. Thus, we perform binary search (see line 25).

During this search, we maintain the invariant that $t' \in [\varrho, \zeta)$ and Equations (4.35) and (4.36) hold. Similar to Paragraph 4.5.5.5.1, ϱ is increased and ζ is decreased until $\varrho = t'$ and either $\zeta = t' + 1$ or $\zeta = \infty$.

Let $\iota \in [\varrho, \zeta)$ be a candidate length during the search. At each search step, we call $\mathsf{extend\text{-}step}(\iota)$, which reports **true** iff $\mathsf{PA}_\mathcal{C}[x_1, x_2]$ and $\mathsf{SA}_\mathcal{C}[y_1, y_2]$ with $[y_1, y_2] = \mathsf{siv}_\mathcal{C}(T[m, m + \iota))$ intersect before i. If $[y_1^\zeta, y_2^\zeta] = \varnothing$, then we compute $[y_1, y_2]$ using only $[y_1^\varrho, y_2^\varrho]$ (see line 2). Else, we can use both $[y_1^\varrho, y_2^\varrho]$ and $[y_1^\zeta, y_2^\zeta]$ (see line 4). As in the last section, we continue to search for t' in $[\iota, \zeta)$ if $\mathsf{PA}_\mathcal{C}[x_1, x_2]$ and $\mathsf{SA}_\mathcal{C}[y_1, y_2]$ intersect before i (see lines 6-8). Else, we continue to search for t' in $[\varrho, \iota)$ (see lines 10-12).

4.5.5.6 Range Queries

Now it remains to describe $\mathsf{intersect}(k, m, t, [x_1, x_2], [y_1, y_2])$ (see Algorithm 31), which returns **true** iff $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$ intersect before i, where $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k))$ and $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(T[m, m+t))$. Recall from Definition 4.5.17 that we need to know the sample index v in order to determine whether $\mathsf{PA}_{\mathcal{C}}[x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y_1, y_2]$ intersect before i, where either $v = 0$ if $\mathcal{C}[1] > i$, or $v \geqslant 1$ is maximal such that $\mathcal{C}[v] < i$.

If $\mathcal{R}$ is dynamic, then we have already set v correctly in lines 5-7 of the framework algorithm (see Algorithm 27) and maintain that $\mathcal{R}$ holds exactly the points $\{(\mathsf{PA}_{\mathcal{C}}^{-1}[1], \mathsf{SA}_{\mathcal{C}}^{-1}[1]), ..., (\mathsf{PA}_{\mathcal{C}}^{-1}[v], \mathsf{SA}_{\mathcal{C}}^{-1}[v])\}$. Thus, we can directly query $\mathcal{R}$ (see line 19). If, however, $\mathcal{R}$ is static, then we instead ensure that $v = 0$ if $\mathcal{C}[1] > m$, or $v \geqslant 1$ is maximal such that $\mathcal{C}[v] < m$ (see lines 2-5). This ensures that we consider all sources before i and thus eliminates the need for considering close sources separately in Paragraph 4.5.5.4.1. Finally, we query $\mathcal{R}$ with weight v (see line 21).

If $\mathcal{R}$ returns a point $p = (\tilde{x}, \tilde{y}) \neq \perp$, then the source $s' = \mathcal{C}[\mathsf{SA}_{\mathcal{C}}[\tilde{y}]] - k + 1$ yields an LZ factor of length $l' = k + t - 1$ for position i, we update $\langle s, l\rangle \leftarrow \langle s', l'\rangle$ if $l' > l$ (see lines 24-26), and return **true**. Else ($p = \perp$), then we return **false** in line 28.

Now, it remains to show that we can use Theorem 4.5.15 to implement $\mathcal{R}$ (see Observation 4.5.18).

4.5.18 Observation. Let $c = T[m]$. Since the head $T[i, i+k)$ ends with c, we have $[x_1, x_2] = \mathsf{piv}_{\mathcal{C}}(T[i, i+k)) \subseteq (C_{\mathcal{C}}[c], C_{\mathcal{C}}[c+1]] = \mathcal{V}_c$ (see Definition 4.5.13). Similarly, since the tail $T[m, m+t)$ starts with c, we have $[y_1, y_2] = \mathsf{siv}_{\mathcal{C}}(T[m, m+t)) \subseteq (C_{\mathcal{C}}[c], C_{\mathcal{C}}[c+1]] = \mathcal{V}_c$. Hence, the query rectangle $[x_1, x_2] \times [y_1, y_2]$ is fully contained in $\mathcal{V}_c$, and we can use Theorem 4.5.15 to implement $\mathcal{R}$.

4.5.5.6.1 Optimization for small Query Ranges If the width $|[x_1, x_2]|$ or the height $|[y_1, y_2]|$ of the query rectangle does not exceed a threshold value θ (in practice, we set $\theta = 4096$), then we answer the query to $\mathcal{R}$ in $\mathcal{O}(\theta)$ time using another method (see lines 7-17).

4.5.19 Definition. Let $\Pi[1..|\mathcal{C}|]$ be the unique permutation of $[1, |\mathcal{C}|]$ such that $\Psi[i] = \mathsf{SA}_{\mathcal{C}}^{-1}[\mathsf{PA}_{\mathcal{C}}[i]]$. Similarly, let $\Psi[1..|\mathcal{C}|]$ be the unique permutation of $[1, |\mathcal{C}|]$ such that $\Psi[i] = \mathsf{PA}_{\mathcal{C}}^{-1}[\mathsf{SA}_{\mathcal{C}}[i]]$.

The query to $\mathcal{R}$ is successful iff

$$\begin{aligned} & \mathsf{PA}_{\mathcal{C}}[x_1, x_2] \cap \mathsf{SA}_{\mathcal{C}}[y_1, y_2] \cap [1, v] \neq \varnothing \\ \Leftrightarrow\ & \exists x \in [x_1, x_2], y \in [y_1, y_2] : \mathsf{PA}_{\mathcal{C}}[x] = \mathsf{SA}_{\mathcal{C}}[y] \leqslant v \\ \Leftrightarrow\ & \exists x \in [x_1, x_2] : \Pi[x] \in [y_1, y_2] \wedge \mathsf{PA}_{\mathcal{C}}[x] \leqslant v \\ \Leftrightarrow\ & \exists y \in [y_1, y_2] : \Psi[y] \in [x_1, x_2] \wedge \mathsf{SA}_{\mathcal{C}}[y] \leqslant v. \end{aligned} \tag{4.37}$$

Algorithm 31: intersect$(k, m, t, [x_1, x_2], [y_1, y_2])$

```
1  if R.is-static() then
2      while v < N ∧ C[v + 1] < m do
3          v ← v + 1;
4      while v > 0 ∧ C[v] ⩾ m do
5          v ← v − 1;
6  p ← ⊥;
7  if min(|[x_1, x_2]|, |[y_1, y_2]|) < θ then
8      if |[x_1, x_2]| < |[y_1, y_2]| then
9          for x from x_1 to x_2 do
10             if Π[x] ∈ [y_1, y_2] ∧ PA_C[x] ⩽ v then
11                 p = (x̃, ỹ) ← (x, Π[x]);
12                 break;
13     else
14         for y from y_1 to y_2 do
15             if Ψ[y] ∈ [x_1, x_2] ∧ SA_C[y] ⩽ v then
16                 p = (x̃, ỹ) ← (Ψ[y], y);
17                 break;
18 else
19     if R.is-dynamic() then
20         p = (x̃, ỹ) ← R.query(T[m], [x_1, x_2], [y_1, y_2]);
21     else
22         p = (x̃, ỹ) ← R.query(T[m], v, [x_1, x_2], [y_1, y_2]);
23 if p ≠ ⊥ then
24     if k + t − 1 > l then
25         l ← k + t − 1;
26         s ← C[SA_C[ỹ]] − k + 1;
27     return true;
28 return false;
```

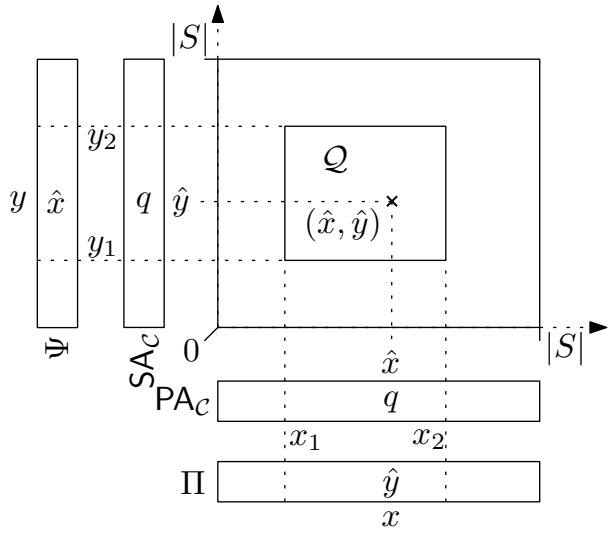


Figure 4.15: Illustration of how we can use Π and Ψ to answer an orthogonal range query. Here, we have $\Pi[x] \in [y_1, y_2]$ and $\Psi[y] \in [x_1, x_2]$. Thus, we can report the point $(x, \Pi[x]) = (\Psi[y], y)$.

Equation (4.37) allows us to answer the query to $\mathcal{R}$ by scanning over $\Pi[x_1, x_2]$ or $\Psi[y_1, y_2]$, respectively. If we encounter an $x \in [x_1, x_2]$ with $\Pi[x] \in [y_1, y_2]$ and $\mathsf{PA}_{\mathcal{C}}[x] \leqslant v$ (see line 10), then we answer the query to $\mathcal{R}$ by reporting the point $p = (x, \Pi[x])$ (see Figure 4.15). If there is no such x, then $\mathsf{PA}_{\mathcal{C}}[x_1, x_2] \cap \mathsf{SA}_{\mathcal{C}}[y_1, y_2] \cap [1, v] = \varnothing$ holds due to Equation (4.37), and we have $p = \perp$ as initialized in line 6.

Similarly, if we encounter a $y \in [y_1, y_2]$ with $\Psi[y] \in [x_1, x_2]$ and $\mathsf{SA}_{\mathcal{C}}[y] \leqslant v$ (see line 15), then we report $p = (\tilde{x}, \tilde{y}) \leftarrow (\Psi[y], y)$. If $|[x_1, x_2]| < |[y_1, y_2]|$, then we scan over $\Pi[x_1, x_2]$ (see lines 9-12), because it faster. Else, we scan over $\Psi[y_1, y_2]$ (see lines 14-17).

4.6 Parallelization

Overall, parallelizing the algorithms is straight-forward, except for the factorization of the gaps (see Section 4.6.2.1). The construction of the $\mathcal{O}(1)$ time LCE data structure (see Theorem 2.8.4) has already been parallelized[1]. In the following, p denotes the number of threads.

4.6.1 LPF Phrases

Here, We decompose T into p non-overlapping, roughly equally-sized regions $T[s_1, s_2), ...,$ $T[s_p, s_{p+1})$, where $s_i = (i-1)\lfloor n/p \rfloor$ for $i \in [1, p]$, and $s_{p+1} = n + 1$. Thread p performs the algorithm described in Section 4.3 only for the samples $S \cap [s_p, s_{p+1})$ that are contained in its region. It finds the first and the last sample within $[s_p, s_{p+1})$ using a binary search

[1] https://github.com/herlez/alx

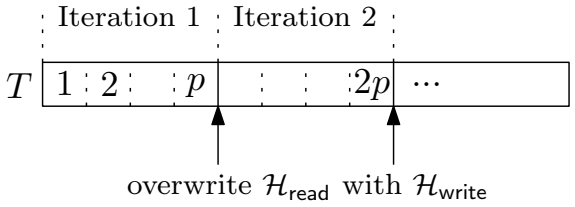


Figure 4.16: Illustration of the parallel gap factorization algorithm. The numbers in T denote the block IDs.

over S, respectively. To simplify the merging of the resulting arrays, we make sure that in section $[s_i, s_{i+1})$, no phrase starts before s_i or ends after s_{i+1}.

4.6.2 Factorizing Gaps

We divide T into $N = \lceil n/B \rceil$ blocks of size $B = \max(4096, I \cdot n/p)$, where $I > 1$ is a parameter. Thus, we have $N \leqslant I \cdot p$ blocks $T[b_1, b_2), ..., T[b_N, b_{N+1})$, where $b_i = i \cdot B$ for $i \in [1, N]$, and $b_{N+1} = n + 1$. We iteratively factorize p consecutive blocks in parallel. In iteration i, we factorize the blocks $(i-1)p + 1, ..., ip$ in parallel, i.e, thread j factorizes $T[b_{(i-1)p+j}, b_{(i-1)p+j+1})$. By the choice of B, we run through at most I iterations.

Suppose we would implement this algorithm using the rolling hash index described in Section 4.4.1. Let i be the current position in T of thread x, and let j be the current position in T of thread $y > x$. Since $y > x$ implies $j > i$, thread y writes values $> i$ into $\mathcal{H}$, thus they cannot be used by thread x to find a factor starting at i. This reduces the compression ratio especially in the leftmost blocks per iteration.

4.6.2.1 Parallel Rolling Hash Index

Instead of storing $\mathcal{H}[1..2^h]$ as described in Section 4.4.1, we store $\mathcal{H}_{\mathsf{read}}[1..2^{h-1}]$ and $\mathcal{H}_{\mathsf{write}}[1..2^{h-1}]$, and read only from $\mathcal{H}_{\mathsf{read}}[1..2^{h-1}]$ and write only to $\mathcal{H}_{\mathsf{write}}[1..2^{h-1}]$. After each iteration, we overwrite $\mathcal{H}_{\mathsf{read}}$ with $\mathcal{H}_{\mathsf{write}}$ (see Figure 4.16).

This mitigates the issue described in the last section, but still reduces the compression ratio, especially in the rightmost blocks per iteration, because the positions stored in $\mathcal{H}_{\mathsf{read}}$ have a greater distance to those blocks.

Since we run through at most I iterations, overwriting $\mathcal{H}_{\mathsf{read}}$ with $\mathcal{H}_{\mathsf{write}}$ takes overall $\Theta(I \cdot 2^h/p) = \Theta(I \cdot n/p)$ time, because $2^h = \Theta(t) = \Theta(\max(n/12, g/3)) = \Theta(n)$ (see Section 4.4.1). In practice, we set $I = 512$.

Let g be the length of the gaps. Compared with the sequential algorithm, this parallel algorithm reduces the compression ratio by a factor up to 2, especially if $0.1 < g/n < 0.5$. If $g/n \leqslant 0.2$, we use the sequential algorithm described in Section 4.4, because in this case, the factorization of the gaps only accounts for a small fraction of the runtime, and using the parallel algorithm only reduces the compression ratio unnecessarily.

4.6.3 Exact LZ77 Algorithm

We parallelized the construction of all data structures used in the exact LZ77 Algorithm. The computation of the exact factorization has been parallelized by splitting the text into p non-overlapping regions $T[s_1, s_2), ..., T[s_p, s_{p+1})$ such that $\forall i \in [1, p) : |\mathcal{E} \cap [s_i, s_{i+1})| = \lfloor |\mathcal{E}|/p \rfloor$ (see Lemma 3.2.7), and $s_{p+1} = n + 1$, i.e, we choose $s_i = e_{\lfloor i|\mathcal{E}|/p \rfloor}$, where $\mathcal{E} = \{e_1, ..., e_{|\mathcal{E}|}\}$ and $e_1 < ... < e_{|\mathcal{E}|}$. Then, thread i computes a sequence $\mathcal{F}_i$ of perfect factors in the range $[s_i, s_{i+1})$. Finally, we output $\mathcal{F} = \mathcal{F}_1 ... \mathcal{F}_p$.

Since only the last factor in each $\mathcal{F}_i$ can be non-perfect, there are at most $p - 1$ non-perfect factors in $\mathcal{F}$, hence we have $z \leqslant |\mathcal{F}| < z + p$. We chose the regions $[s_i, s_{i+1})$ in this way, because the running time to compute $\mathcal{F}_i$ rises with the number of perfect factors in $T[s_i, s_{i+1})$, and the number $|\mathcal{E} \cap [s_i, s_{i+1})|$ of approximate LZ factors in $T[s_i, s_{i+1})$ is a good estimate for it. Note that we have to use a data structure for the SW-OROR problem in the parallel setting.

Chapter 5

Experimental Evaluation

Now, we discuss implementation details, the experimental setup and results.

5.1 Implementation Details

We implemented all algorithms in `C++20`. Our implementation is available on GitHub [1]. We use the implementation of Theorem 2.8.4 from [2], which builds upon [10]. For sorting, we use the (parallel) in-place sample sort implementation `ips4o` [3] [2]. For the sparse prefix- and suffix array sampling described in Section 4.5.3, we use the memory-efficient hash table implementation `sparse-map` [4] and adapted the Karp-Rabin fingerprinting implementation [5], which uses the mersenne prime $q = 2^{61} - 1$. For the rolling hash index described in Section 4.2.1.3, we use the rolling Karp-Rabin fingerprinting implementation that is included in [2]. It uses the mersenne prime $q = 2^{107} - 1$.

We chose a larger mersenne prime $q = 2^{107} - 1$ for the rolling hash index, because it improves the compression ratio. This is likely due to the reduced fingerprint collision rate. For the sparse prefix and suffix array sampling, however, we chose the smaller mersenne prime $q = 2^{61} - 1$, because using $q = 2^{107} - 1$ did not reduce the time to look up an interval. This is probably because the running time is dominated by the LCE queries inLemma 4.5.5.

5.2 Experimental Setup

All measurements have been performed on a system with two AMD EPYC 7452 CPUs (32/64x 2.35-3.35GHz, 2/16/128MB L1/2/3 cache) and 1TB of 3200 MT/s DDR4 RAM.

[1] `https://github.com/LukasNalbach/lz77-sss/`
[2] `https://github.com/LukasNalbach/lce/`
[3] `https://github.com/ips4o/ips4o/`
[4] `https://github.com/Tessil/sparse-map/`
[5] `https://github.com/pdinklag/fp/`

Text	Size [GB]	σ	n/z	g/n
einstein.en.txt	0.467	139	5226	0.00033
cere	0.461	6	271.2	0.15856
english	2.21	239	19.74	0.70016
boost	0.629	96	27658	0.00054
dewiki.20Gi	20.47	210	1543.1	0.01567
sars2.20Gi	20.47	80	2434.7	0.01206
chr19.20Gi	20.47	52	550.15	0.05899
commoncrawl.txt.16Gi	16.0	243	30.86	0.57768

Table 5.1: Statistics of the tested texts. g is the length of the gaps.

We used the GCC 12.3.0 compiler with the compile flags `"-march=native -DNDEBUG -Ofast"` and parallelized our algorithms with OpenMP [6].

Table 5.1 shows the tested texts. einstein.en.txt, cere, english and boost are part of the Pizza&Chili Corpus [7]. dewiki is a highly repetitive text that has been handcrafted from german Wikipedia entries. chr19 consists of concatenated human chromosome 19 haplotypes, and sars2 is a collection of Sars-Cov-2 genomes, both of which were crafted out of datasets from the *National Center for Biotechnology Information* [8] (NCBI) database. Finally, commoncrawl.txt has been handcrafted from the *Common Crawl Project* [9] database.

5.3 Results

We begin by determining the optimal value for τ (see Section 5.3.1). Then, we compare the data structures for OROR from Section 4.5.4 in Section 5.3.2. In Section 5.3.3, we compare our (approximate) LZ77 algorithms with each other and the LZ77 LPF algorithm from Section 2.4. In Section 5.3.4, we compare our approach to separately compress the gaps from Section 4.2.2 with standard compressors. Finally, we examine the parallel scaling of our algorithms in Section 5.3.5.

5.3.1 The optimal value for τ

To determine the optimal value for τ, we compared the running time, peak memory consumption and approximation ratio of the LZ77 3-Approximation with different values for τ, i.e, $\tau = 2^i$ for $i \in [4, 11]$ (see Figure 5.1). In general, the approximation ratio is independent of τ. Choosing $\tau \in \{128, 256\}$ can yield higher compression throughput compared

[6] `https://openmp.org/`
[7] `https://pizzachili.dcc.uchile.cl/`
[8] `https://www.ncbi.nlm.nih.gov/`
[9] `https://commoncrawl.org/`

with $\tau \geqslant 512$ if the text has long periodic regions (see cere). However, choosing $\tau \geqslant 512$ reduces the peak memory consumption and increases the throughput with non-periodic texts. Therefore, we determined $\tau = 512$ to be optimal and use it for all following measurements.

5.3.2 Comparison of Data Structures for IO-/SW-OROR

In this section, we measured the performance of the OROR data structures described in Section 4.5.4.

5.3.2.1 Methodology

To generate practical measurement data, we extracted the OROR instances, queries and inserts during the execution of the algorithm `exact` from Section 5.3.3, i.e, the exact LZ77 algorithm described in Section 4.5.5, but without the sparse prefix- and suffix array sampling. Queries that are answered by scanning over Ψ or Π (see Paragraph 4.5.5.6.1) are not included.

Let Q be the number of performed queries, let I be the number of inserts, and let N be the number of points in the OROR instance of a given text. Let $\mathcal{D}$ be a data structure for the OROR problem that solves this instance. Let $s_{\mathcal{D}}$ be the size of $\mathcal{D}$, let $c_{\mathcal{D}}$ be the time to construct $\mathcal{D}$, and let $q_{\mathcal{D}}$ and $i_{\mathcal{D}}$ be the times to perform all queries with and insert all points into $\mathcal{D}$, respectively. Then, the marker in Figure 5.2 for $\mathcal{D}$ shows $\mathcal{D}$'s combined construction-, insert- and query throughput $\frac{N+Q+I}{c_{\mathcal{D}}+q_{\mathcal{D}}+i_{\mathcal{D}}}$ versus its size $s_{\mathcal{D}}$. Figure A.2 shows construction time $c_{\mathcal{D}}$ versus peak memory consumption during the construction. Figure A.1 shows insert- and query throughput $\frac{Q+I}{q_{\mathcal{D}}+i_{\mathcal{D}}}$ versus data structure size $s_{\mathcal{D}}$.

The upper rows in the legends list the data structures described in Section 4.5.4. The lower rows list their "decomposed" counterparts (indicated by the prefix "`D-`", see Section 4.5.4.3). We measured the grid-based data structures with different grid sizes m, i.e, $m = 2^i$ for $i \in [11, 16]$. The markers for different values of m are connected, and the leftmost marker uses $m = 2^{11}$, respectively.

5.3.2.2 Discussion

As argued in Section 4.5.4, the decomposed grid-based data structures require less memory than their non-decomposed counterparts. However, decomposing the instance $\langle \mathcal{P}, \mathcal{I} \rangle$ into the instances $\langle \mathcal{P}_1, \mathcal{I}_1 \rangle, ..., \langle \mathcal{P}_\sigma, \mathcal{I}_\sigma \rangle$ introduces a construction time overhead (see e.g. boost). It also increases the peak memory consumption during construction (see Figure A.2).

The dynamic data structures (`D-SG`, `SD-SG`, `D-D-SG` and `D-SD-SG`) provide better throughput and are smaller if the grid size m is set optimally, because they do not store the weight of each point (see Table 4.1). Out of the static weighted data structures, `SW-SS` and `D-SW-SS` are consistently the smallest, but also the slowest. `SW-SG` and `D-SW-SG` are sometimes smaller and provide better throughput than `SW-KDT` and `D-SW-KDT` for specific (but

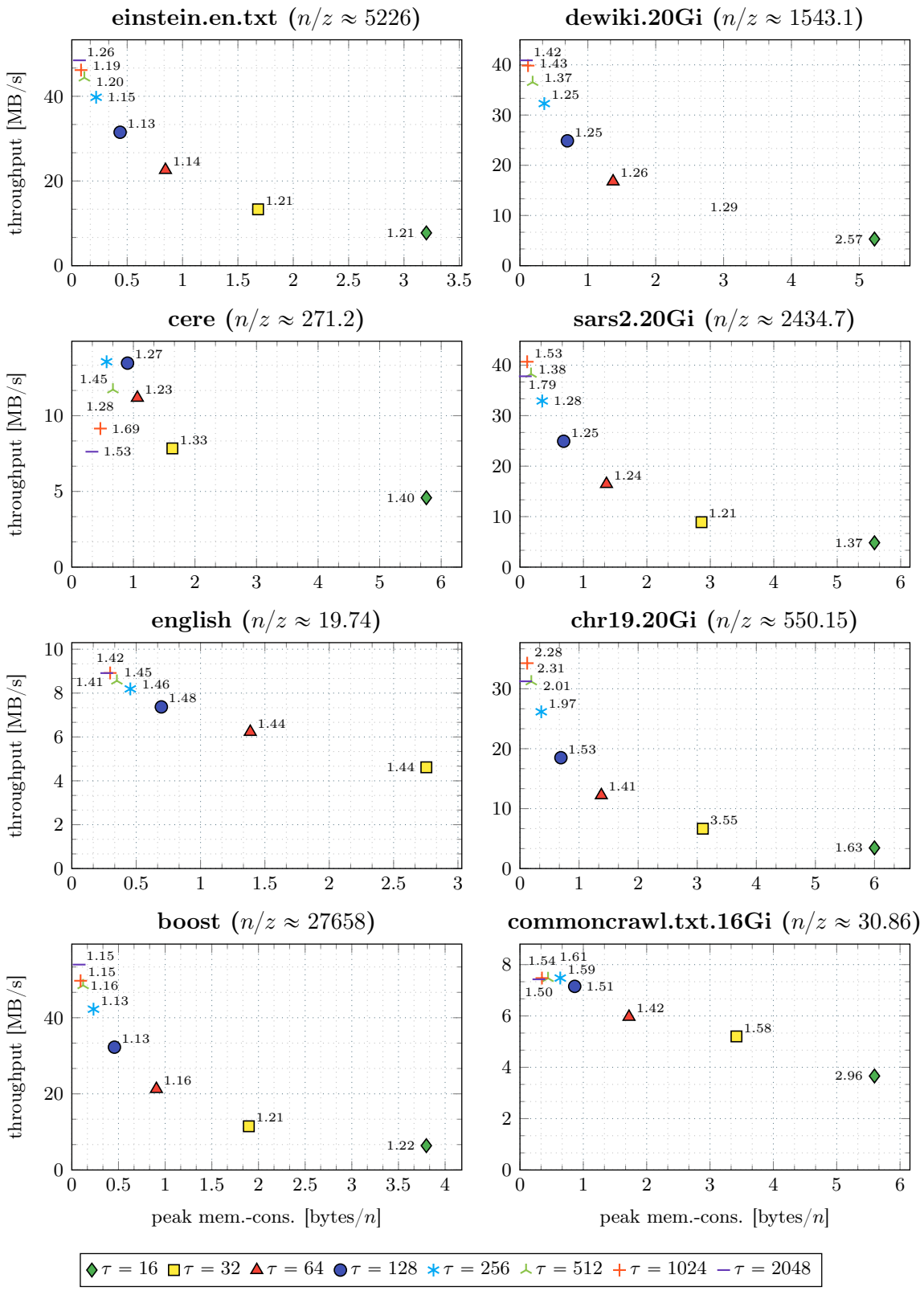


Figure 5.1: Throughput versus peak memory consumption. Each marker is annotated with the approximation ratio.

varying) values for m (see einstein.en.txt and boost). Overall, setting $m = 2^{14}$ provides the best trade-off between performance and space.

However, although `SW-KDT` and `D-SW-KDT` are slightly larger than `SW-SG` and `D-SW-SG`, they generally offer more consistent performance, and do not rely on the grid size m being set optimally.

Since we have to use a static weighted data structure in the parallel exact LZ77 algorithm (see Section 4.5.5), we perform all following measurements with `D-SW-SG` and $m = 2^{14}$, unless stated otherwise.

5.3.3 Comparison of LZ77 Construction Algorithms

In this section, we compare our LZ77 algorithms with each other and the LZ77 LPF algorithm from Section 2.4. Figure 5.3 shows running time versus additional peak memory usage (T is not included in the memory usage).

- `lpf` is a version of the LZ77 LPF algorithm [10] (see Section 2.4) that does not use the arrays NSV and PSV, and instead scans for the previous and next smaller values naively in SA. This reduces the peak memory consumption. It builds SA with `libsais` [11] and then naively constructs SA^{-1}. Hence, it needs $8n$ additional bytes if $n < 2^{31}$ (bscause it uses signed integers), and $16n$ bytes, else.

- `3-aprx-naive` is an implementation of the LZ77 3-approximation that adheres as closely as possible to the theoretical description in Section 3.1.2. It computes LPF phrases as described in Lemma 3.1.5, uses the rolling hash index described in Section 4.4.1, and factorizes the gaps using the algorithm from Section 4.4.3.

- `3-aprx` is a practically optimized version of `3-aprx-naive`. It computes the LPF phrases as described in Section 4.3.1 and factorizes the gaps using the algorithms described in Sections 4.4.4 and 4.6.2.1.

- `lpf-lnf-aprx` computes LPF- and LNF phrases as described in Section 4.3.2. Then, it factorizes the gaps like `3-aprx`.

- `exact-naive` is an implementation of the LZ77 exact algorithm that adheres as closely as possible to the theoretical description in Section 3.2, however, it also uses the optimizations from Sections 4.5.1 and 4.5.5.1. In this algorithm, the ORORQ queries with small ranges (width and height $\leqslant \theta$) are not answered using the arrays Π and Ψ from Paragraph 4.5.5.6.1. It uses the `SW-SG` to implement $\mathcal{R}$.

[10] `https://github.com/pdinklag/lz77/`
[11] `https://github.com/IlyaGrebnov/libsais/`

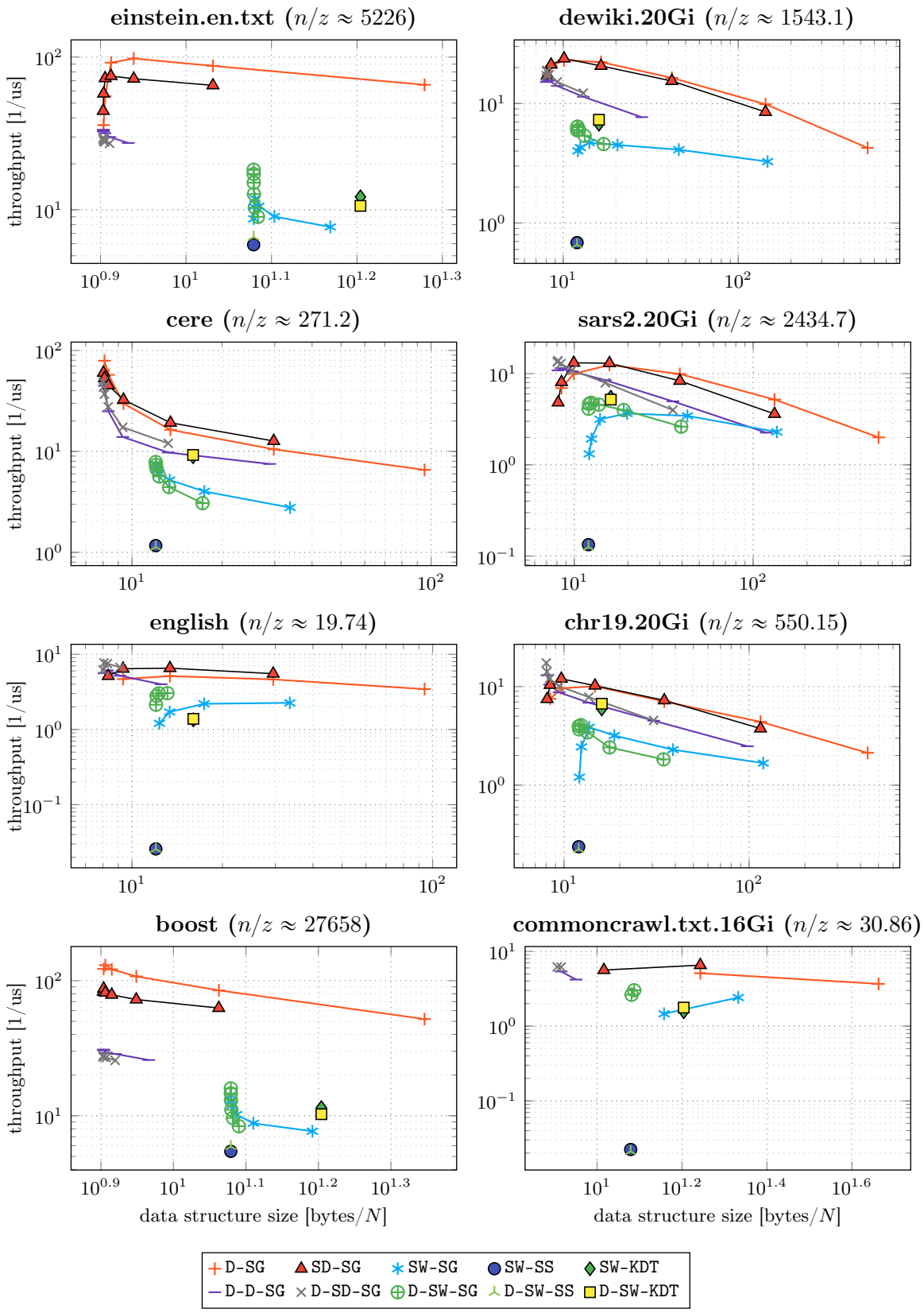


Figure 5.2: Construction-, insert- and query throughput versus data structure size.

- `exact` is an optimized version of `exact-naive`. It additionally employs the optimizations described in Sections 4.5.1.1, 4.5.4.3 and 4.5.5.6 and implements $\mathcal{R}$ with `D-SW-SG`.

- `exact-samples` is the exact LZ77 algorithm described in Section 4.5.5. It is an optimized version of `exact`, as it additionally employs the optimizations described in Sections 4.5.2, 4.5.3 and 4.5.5.2 to 4.5.5.5.

`lpf` requires the most memory out of all algorithms, because it constructs SA and SA^{-1}. The SSS-based LZ77 approximation algorithms (`3-aprx-naive` `3-aprx`, `lpf-lnf-aprx`) use 2-4 times less memory than `exact`.

`3-aprx` is 1.3-4 times faster than `3-aprx-naive`, because here, the rolling hash index skips the gaps. `3-aprx` also yields better compression rates due to the optimizations described in Section 4.3.1 and Section 4.4.4. With repetitive texts, `lpf-lnf-aprx` is $\approx$ 2 times slower than `3-aprx`, because factorizing the gaps only accounts for a negligible part of the total runtime (see boost). With unrepetitive texts, their difference is smaller, because here, the running time is more dependent on the factorization of the gaps (see commoncrawl.txt.16Gi). `lpf-lnf-aprx` requires more memory than `3-aprx`, because it stores more LPF phrases. Overall, `lpf-lnf-aprx` achieves better approximation ratios.

With repetitive texts, the compression rate of `3-aprx` does not decrease with the number of threads, because in this case, the gaps only account for a small fraction of the text (see Table 5.1), and we use the sequential algorithm to factorize the gaps if $g/n < 0.2$ (see Section 4.6.2.1). For $g/n \geqslant 0.2$, however, the compression ratio decreases when using multiple threads (see english and commoncrawl.txt.16Gi).

Out of the SSS-based exact LZ77 algorithms, `exact-naive` is the slowest and requires the least memory. `exact` is 5-10 times faster than `exact-naive`, but requires 50% more additional memory (because it additionally stores Π and Ψ, and the construction of the `D-SW-SG` requires more memory than the construction of `SW-SG`, see Figure A.2). `exact-samples` needs 1.2-5 times as much additional memory (10-50% more memory overall) than `exact`, but is 1-2.2 times as fast.

5.3.4 Comparison with Standard Compressors

Out of the two variants for separately compressing the gaps (see Section 4.2.2), Variant 2 preforms better in practice. We use `zstd` with encoding quality `"-4"` to compress the intermediary representation $\mathcal{F}'$. We call our implementation `ssszip` and compare it with standard compressors. Some of them use dictionary compression.

A dictionary contains frequent strings and is used during decompression to copy substrings from. Static dictionaries are stored separately with the decompressor and are not part of the compressed representation of the text. Dynamic dictionaries are created during

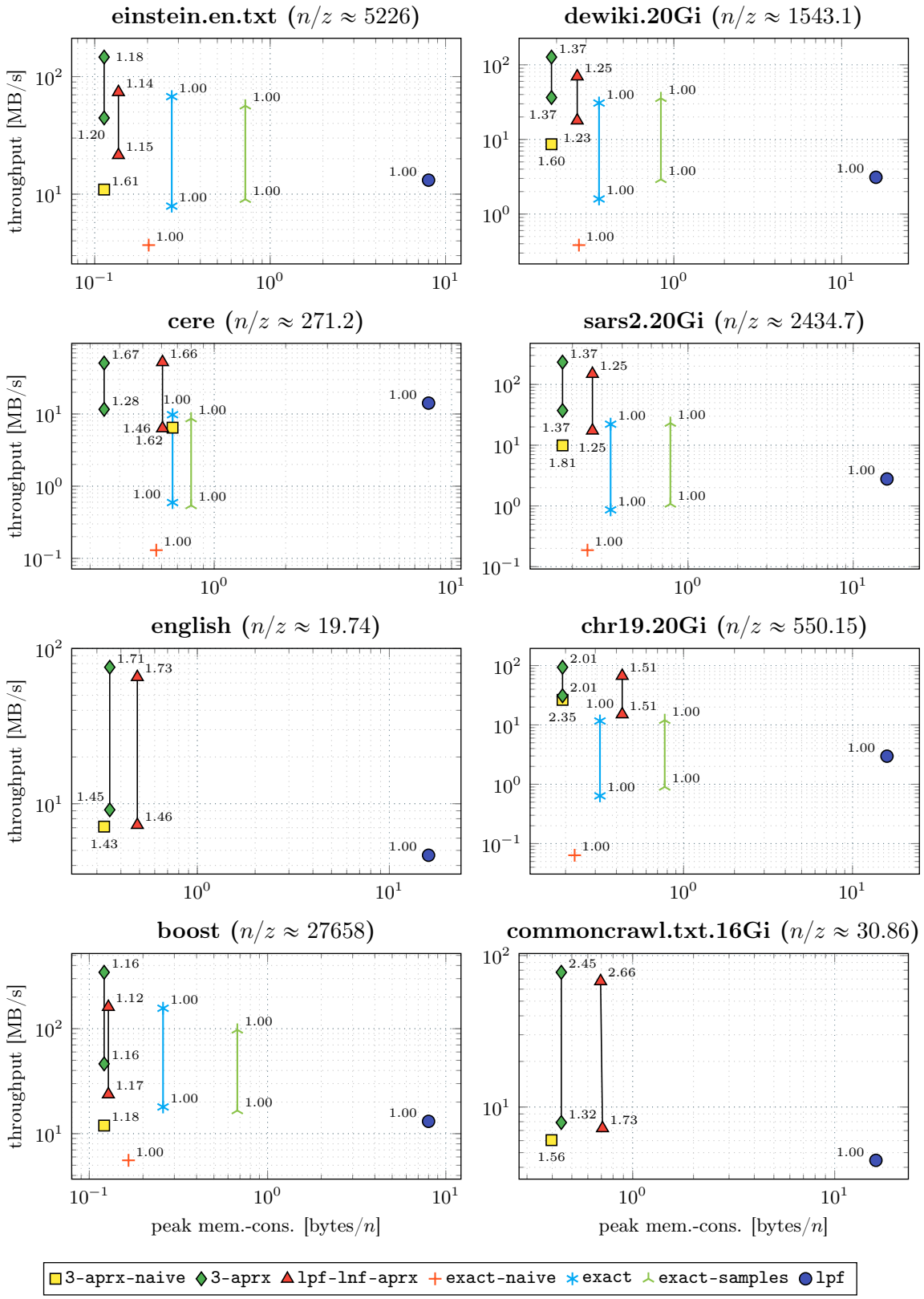


Figure 5.3: Throughput versus additional peak memory consumption (without T). Each marker is annotated with the approximation ratio. For each pair of connected markers, the lower/upper mark denotes the execution with 1/32 threads.

compression and may therefore be more suitable for the given text. However, constructing them requires work, and they must be included in the compressed representation.

- `gzip` [12] uses the *deflate algorithm* [9]. It combines sliding-window LZ77 [4] and *Huffman coding* [24], which replaces frequently occurring symbols with shorter codes. `gzip` is widely used for HTTP compression.
- `bzip2` [13] divides the text into 900 KB blocks (by default) and then compresses each block using the *Burrows-Wheeler Transform* (BWT) [6], followed by *Move-to-Front* (MTF) encoding [3] and Huffman coding. The BWT reversibly rearranges the text to group similar characters together, while MTF converts these groups into sequences that are easier to compress. Huffman coding then further reduces the space to store those sequences. `bzip2` offers a higher compression ratio than `gzip`, but is generally slower. It is rarely used today, because faster and more optimal compression algorithms are available.
- Like `bzip2`, `bsc` [14] also uses the BWT and Move-to-Front encoding. However, it uses the more optimized suffix array construction algorithm `libsais` [11] to construct the BWT, chooses a larger 25 MB block size by default, uses *quantized local frequency coding* [20] instead of Huffman coding, and employs other preprocessing techniques like *Lempel-Ziv preprocessing* (LZP), which removes redundancies with LZ phrases beforehand. `bsc` supports multi-threading.
- `lz4` [15] is designed for speed, and implements a fast variant of sliding-window LZ77 [4]. It is widely used in real-time applications, where short and predictable execution times are essential.
- `xz` [16] employs the *Lempel–Ziv–Markov chain algorithm* (LZMA) to achieve high compression rates. It uses *range coding* [36] instead of Huffman coding and uses an 8 MB static dictionary by default. `xz` is common for distributing Linux packages and supports multi-threading.
- `7z` [17] uses LZMA2, which is an improved version of LZMA. It supports multi-threading and is commonly used for compressing very large data sets, where the highest compression is neccessary.
- `zstd` [18] was designed by Facebook and aims to offer fast compression with high compression rates by combining sliding-window LZ77, Huffman coding and other

[12] `https://gnu.org/software/gzip/`
[13] `https://sourceware.org/bzip2/`
[14] `http://libbsc.com/`
[15] `https://lz4.org/`
[16] `https://tukaani.org/xz/`
[17] `https://7-zip.org/`
[18] `https://facebook.github.io/zstd/`

entropy coding techniques like *Finite State Entropy* [19]. Additionally, it uses static and dynamic dictionaries. It supports multi-threading and is used in many modern applications, such as containerized environments (e.g. Docker), backup systems, and big data processing.

5.3.4.1 Compression Performance

Among the very memory efficient compressors `lz4`, `gzip` and `bzip2`, `lz4` is the fastest and `bzip2` uses the least memory. `xz` and `7z` achieve very good compression rates, but are slower and require more memory. When using multiple threads, their throughput increases, but memory consumption also increases, and compression ratio decreases (see boost). This is because the data is split into chunks that are processed independently of each other. `zstd` requires a medium amount of memory, yields medium compression rates, but is the fastest compressor overall (just as fast as `lz4` on average).

`ssszip` always uses slightly more than n bytes of memory, because it stores the whole text in memory. `bsc` requires a similar amount of memory on average and is sometimes faster and sometimes slower than `ssszip`. Although `bsc` achieves slightly better compression than `ssszip` on unrepetitive texts (see english and commoncrawl.txt.16Gi), `ssszip` yields much higher compression rates on repetitive texts (see all other texts, except for dewiki.20Gi). Especially on large texts with long repetitions, `ssszip` provides the best compression rate (see chr19.20Gi). Other than `xz` and `7z`, `ssszip` and `bsc` do not achieve parallelism by applying the same algorithm to multiple chunks of the text at once, but by parallelizing the underlying algorithms. Therefore, their memory consumption does not increase with the number of used threads. When using one thread, `ssszip` yields slightly lower compression rates than `xz` and `7z`. When using multiple threads, however, their compression rates decrease below that of `ssszip`.

When decreasing the block size with `bsc`, the throughput increases slightly. However, the compression rate also decreases (see `bsc-12`, `bsc-6` and `bsc-3` in Figure 5.5). The memory consumption of `bsc` is roughly equal when compressing and decompressing, and decreases when decreasing the block size. To preserve the readability of Figure 5.4, we included only `bsc-25`.

5.3.4.2 Decompression Performance

Regarding decompression performance, `zstd` is generally the fastest. `bzip2` is always the slowest. `gzip` requires almost no additional memory. `bsc` decompression speed is very variable. It requires the most memory of all compressors, because it has to revert the BWT of each block, which requires the BWT and a data structure to compute `LF` to

[19] `https://github.com/Cyan4973/FiniteStateEntropy/`

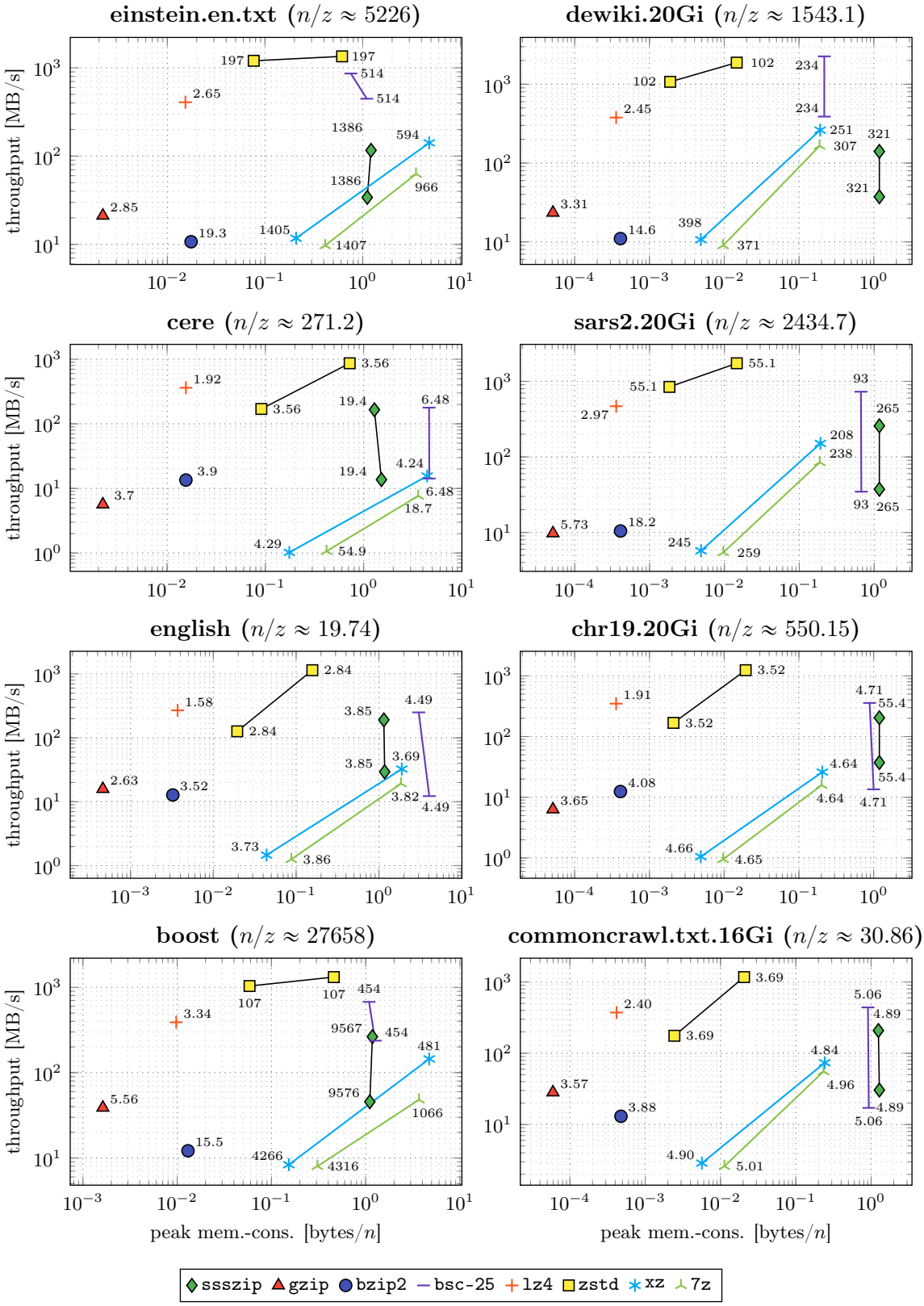


Figure 5.4: Compression throughput versus peak memory consumption. Each marker is annotated with the compression ratio.

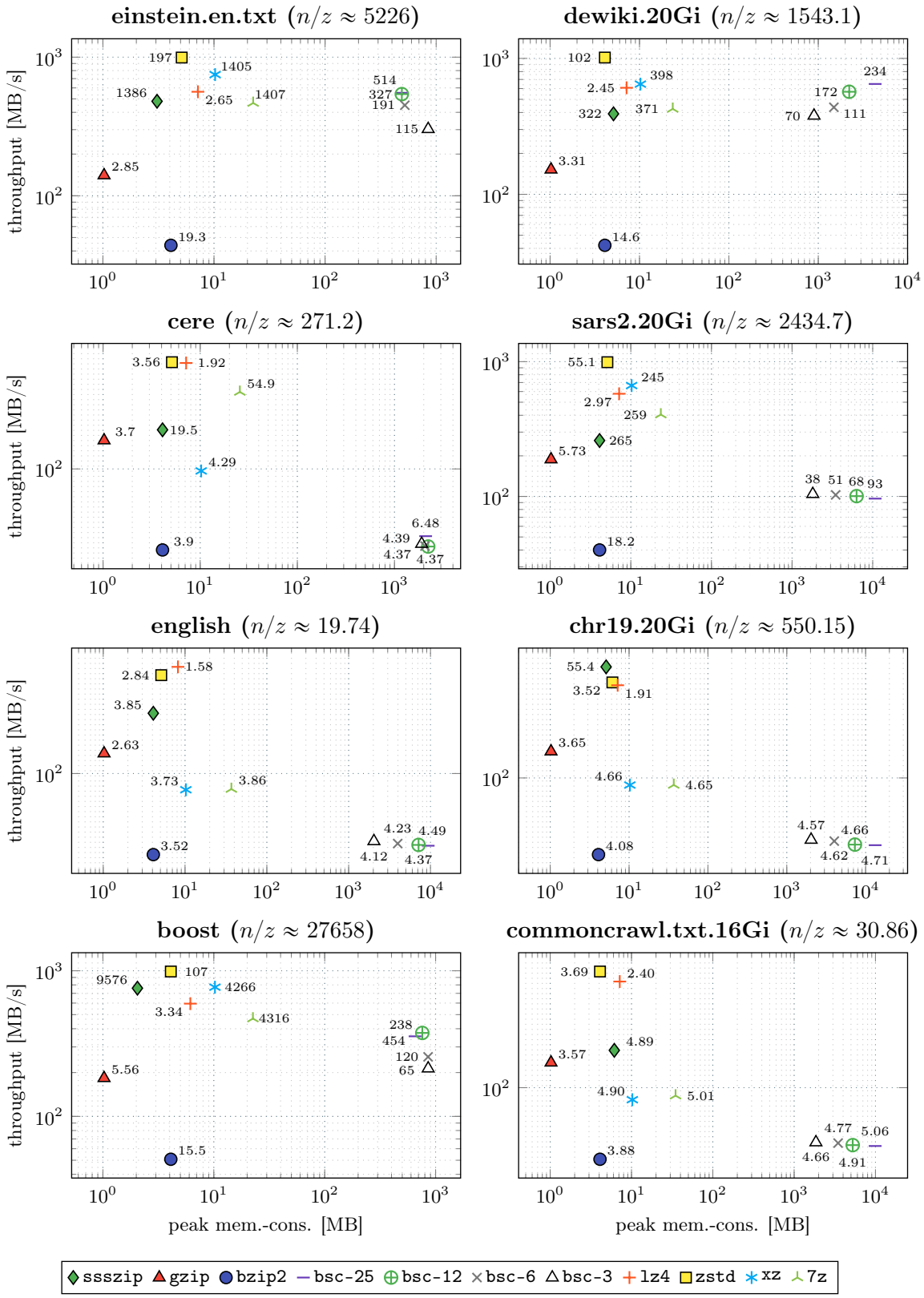


Figure 5.5: Decompression throughput versus peak memory consumption. Each marker is annotated with the compression ratio.

be stored in the RAM. The other compressors (`ssszip`, `xz`, `7z`) have a consistently low memory requirement and offer medium but variable decompression speed.

5.3.5 Construction Phases and Parallel Scaling

In this section, we examine the running times of individual construction phases and the parallel scaling of our algorithms. Figures 5.6 to 5.10 show the construction phases of `3-aprx`, `ssszip`, `lpf-lnf-aprx`, `exact` and `exact-samples` from the previous sections.

$\mathcal{F}'$ and $\mathcal{F}$ denote the computation of the approximate and the exact factorization, respectively. $\mathcal{H}$ denotes the initialization of the rolling hash index. SEL stands for the selection of LPF and LNF phrases, i.e, the final step of the algorithm described in Section 4.3.2. Finally, KRS denotes the construction of the data structure for computing Karp-Rabin fingerprint of substrings from T (see Section 4.5.2).

For repetitive texts, the running times of `3-aprx` and `lpf-lnf-aprx` are dominated by the construction of the LCE data structure. The running time to compute NSV_S, PSV_S and the LPF phrases are always negligible. The relative running time to factorize the gaps decreases with the relative length of the gaps, which decreases with the repetitiveness of the text (compare boost and commoncrawl.txt.16Gi).

The running time to construct the LCE data structure scales better in parallel than factorizing the gaps. Recall that if the relative length g/n of the gaps is less than 0.2, then we use the sequential algorithm to factorize the gaps, even if multiple threads are available (compare cere and commoncrawl.txt.16Gi). Generally, `ssszip` compresses the gaps much faster than the rolling hash index, and scales better in parallel (compare commoncrawl.txt.16Gi in Figure 5.6 and Figure 5.7).

Now, we discuss the exact LZ77 algorithms. The time to compute the Karp-Rabin, sparse prefix- and suffix array interval samplings (phases KRS, $\mathsf{H}(\mathsf{PIV}_{\mathcal{C}})$ and $\mathsf{H}(\mathsf{SIV}_{\mathcal{C}})$ in Figure 5.10) is negligible, especially for large texts. The use of these samples reduces the running time to compute the exact factorization by factors between 1 and 2.2 (see cere and dewiki.20Gi). However, the speedup is reduced when increasing the number of threads, i.e, the overall running times of `exact` and `exact-samples` are almost identical when using 32 threads (compare Figure 5.9 and Figure 5.10).

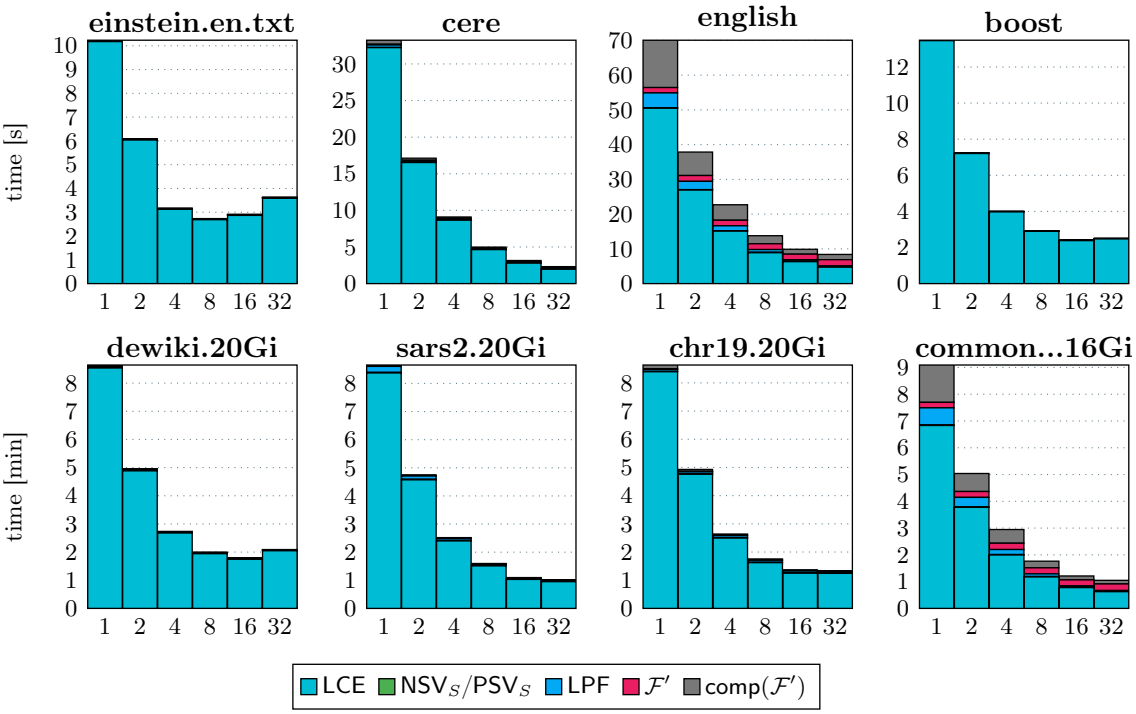


Figure 5.6: Construction time versus number of threads of `ssszip` from Section 5.3.4.

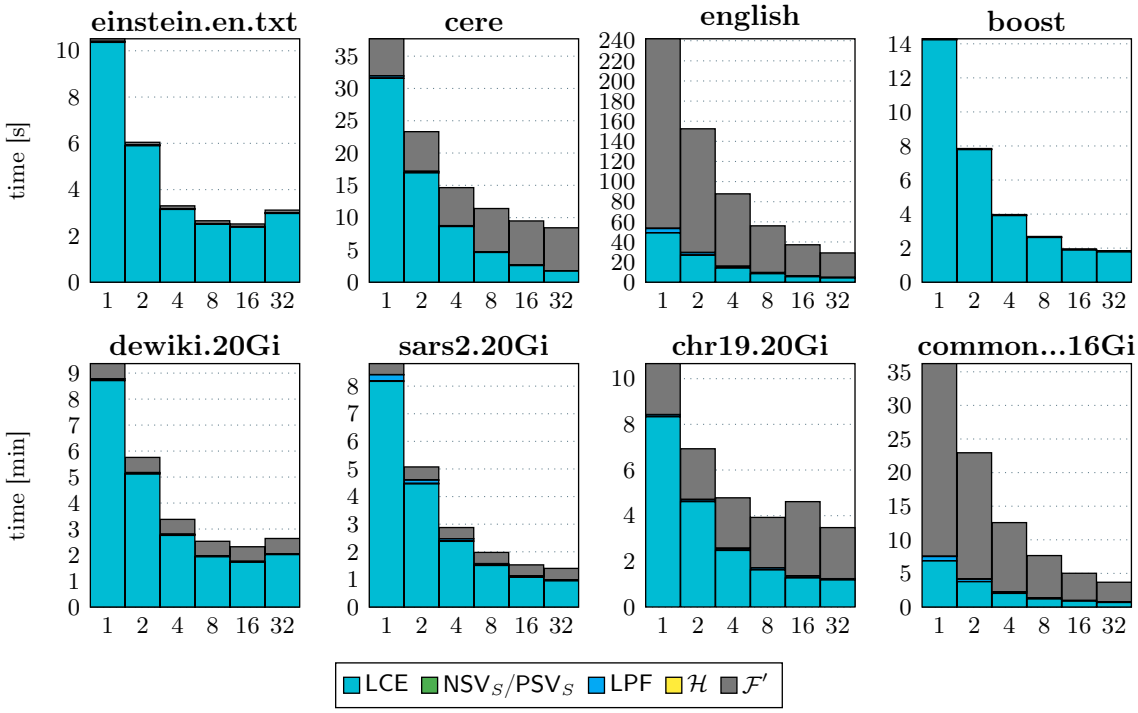


Figure 5.7: Construction time versus number of threads of the LZ77 3-Aproximation, i.e, `3-aprx` from Section 5.3.3.

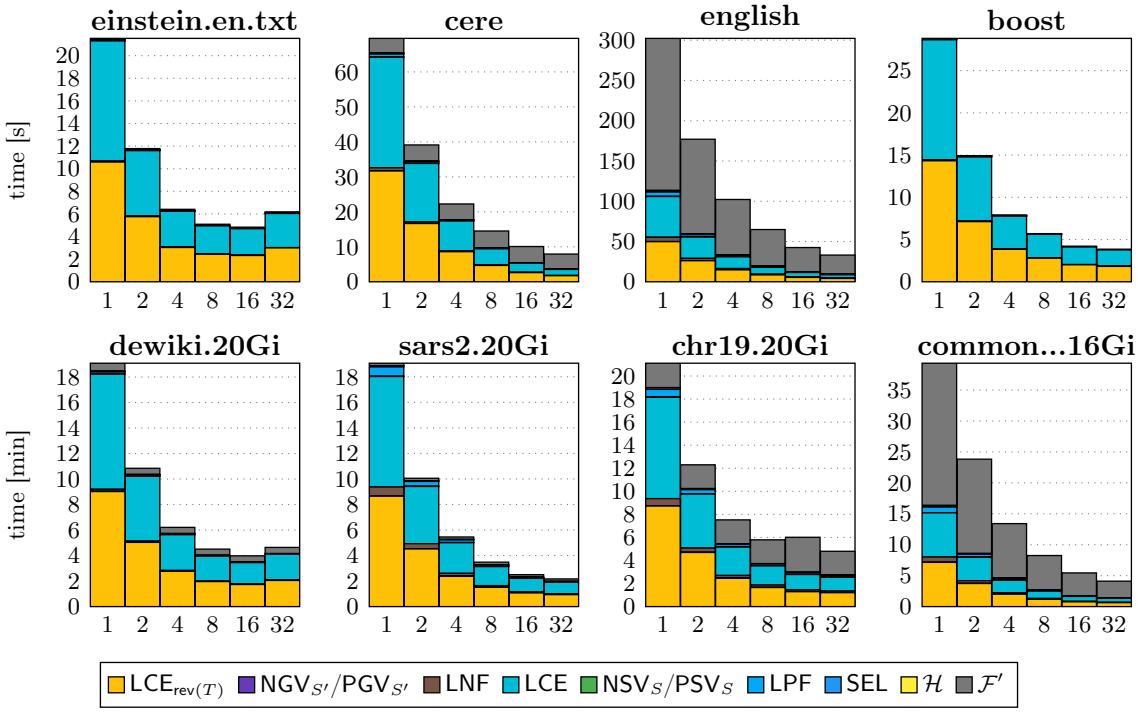


Figure 5.8: Construction time versus number of threads of the LZ77 LPF/LNF Aproximation, i.e, `lpf-lnf-aprx` from Section 5.3.3.

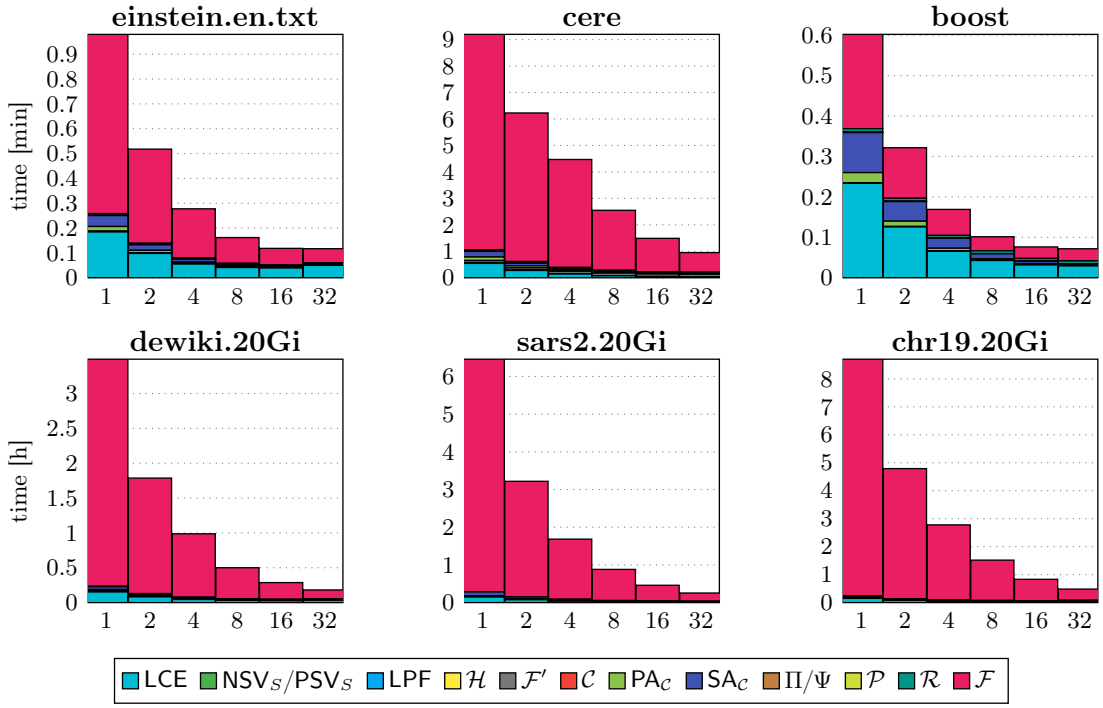


Figure 5.9: Construction time versus number of threads of the exact LZ77 algorithm without sparse prefix- and suffix array interval sampling, i.e, `exact` from Section 5.3.3.

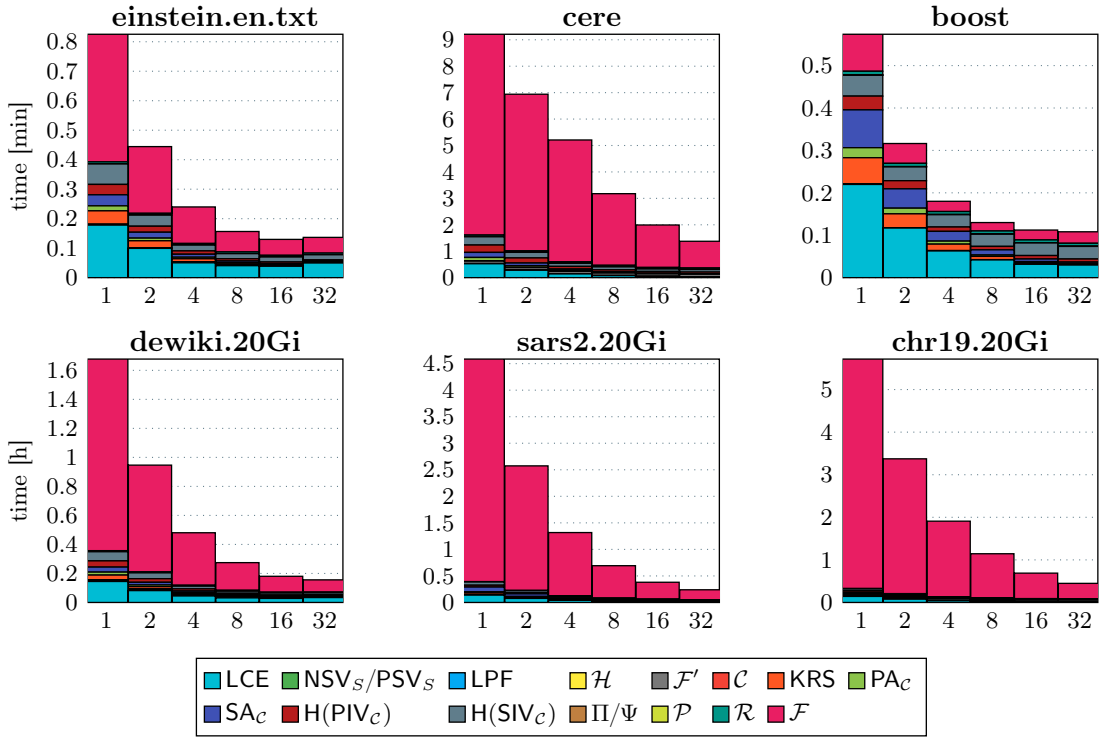


Figure 5.10: Construction time versus number of threads of the exact LZ77 algorithm with sparse prefix- and suffix array interval sampling, i.e, `exact-samples` from Section 5.3.3.

Chapter 6

Conclusion

Overall, we have shown that the LZ77 algorithms described in [12] can be adapted to perform well in practice.

The LZ77 3-Approximation needs $8 - 16$ times less memory than a practical implementation of the LZ77 LPF algorithm, and is 0.9-10 (typically 3-10) times faster while achieving approximation ratios between 1 and 2 (typically 1.1-1.5).

The exact LZ77 algorithm still needs $2 - 4$ times less memory than the LPF algorithm, but performs well only for repetitive texts. We implemented a slightly more space-consuming version of the algorithm, which needs 10-50% more memory, but is up to 2.2 times faster.

Finally, our practical compression tool `ssszip` uses a predictable amount of memory ($1.1n$ to $1.5n$ bytes) and is an order of magnitude faster than state-of-the-art compressors that yield similar compression rates. Although faster and/or more space-efficient compressors exist, they yield much lower compression rates, especially with repetitive texts.

Appendix A

Appendix

A.1 Pseudocode for Sparse Suffix Array Interval Search

Algorithm 32: interpolate-ssa$([i, i+l), \langle [b_1, e_1], l_1 \rangle, \langle [b_2, e_2], l_2 \rangle)$

```
1  [b, e] ← [b_1, b_2];
2  lce_b ← l_1 + LCE(S[SA_S[b]] + l_1, i + l_1);
3  lce_e ← l_2;
4  while |[b, e]| > 2 do
5      m ← ⌊(b + e)/2⌋;
6      p ← S[SA_S[m]];
7      lce_min ← min(lce_b, lce_e);
8      lce_m ← lce_min +
         LCE(p + lce_min, i + lce_min);
9      if lce_m < l then
10         b ← m;
11         lce_b ← lce_m;
12     else
13         e ← m;
14         lce_e ← lce_m;
15 if lce_b ⩾ l then
16     x ← b;
17 else
18     x ← e;
19 [b, e] ← [e_2, e_1];
20 lce_b ← l_2;
21 lce_e ← l_1 + LCE(S[SA_S[e]] + l_1, i + l_1);
22 while |[b, e]| > 2 do
23     m ← ⌊(b + e)/2⌋;
24     p ← S[SA_S[m]];
25     lce_min ← min(lce_b, lce_e);
26     lce_m ← lce_min +
         LCE(p + lce_min, i + lce_min);
27     if lce_m < l then
28         e ← m;
29         lce_e ← lce_m;
30     else
31         b ← m;
32         lce_b ← lce_m;
33 if lce_e ⩾ l then
34     x' ← e;
35 else
36     x' ← b;
37 return [x, x'];
```

Algorithm 33: ssa-interval$([i, i+l), [b, e])$

```
1  lce_b ← LCE(S[SA_S[b]], i);
2  lce_e ← LCE(S[SA_S[e]], i);
3  [b̂, ê] ← [b, e];
4  lce_b̂ ← lce_b;
5  lce_ê ← lce_e;
6  while |[b, e]| > 2 do
7      m ← ⌊(b + e)/2⌋;
8      let p = S[SA_S[m]];
9      lce_min ← min(lce_b, lce_e);
10     lce_m ← lce_min +
         LCE(p + lce_min, i + lce_min);
11     if lce_m ⩾ l then
12         e ← m;
13         lce_e ← lce_m;
14         if m > b̂ then
15             b̂ ← m;
16             lce_b̂ ← lce_m;
17     else if p + lce_m = n + 1 ∨
         T[p + lce_m] ≺ T[i + lce_m] then
18         b ← m;
19         lce_b ← lce_m;
20         if m > b̂ then
21             b̂ ← m;
22             lce_b̂ ← lce_m;
23     else
24         e ← m;
25         lce_e ← lce_m;
26         if m < ê then
27             ê ← m;
28             lce_ê ← lce_m;
29 if max(lce_b, lce_e) < l then
30     return ∅
31 if lce_b ⩾ l then
32     x ← b;
33 else
34     x ← e;
35 [b, e] ← [b̂, ê];
36 lce_b ← lce_b̂;
37 lce_e ← lce_ê;
38 while |[b, e]| > 2 do
39     m ← ⌊(b + e)/2⌋;
40     let p = S[SA_S[m]];
41     lce_min ← min(lce_b, lce_e);
42     lce_m ← lce_min +
         LCE(p + lce_min, i + lce_min);
43     if lce_m ⩾ l then
44         b ← m;
45         lce_b ← lce_m;
46     else
47         e ← m;
48         lce_e ← lce_m;
49 if lce_e ⩾ l then
50     x′ ← e;
51 else
52     x′ ← b;
53 return [x, x′];
```

A.2 OROR Construction and Query Results

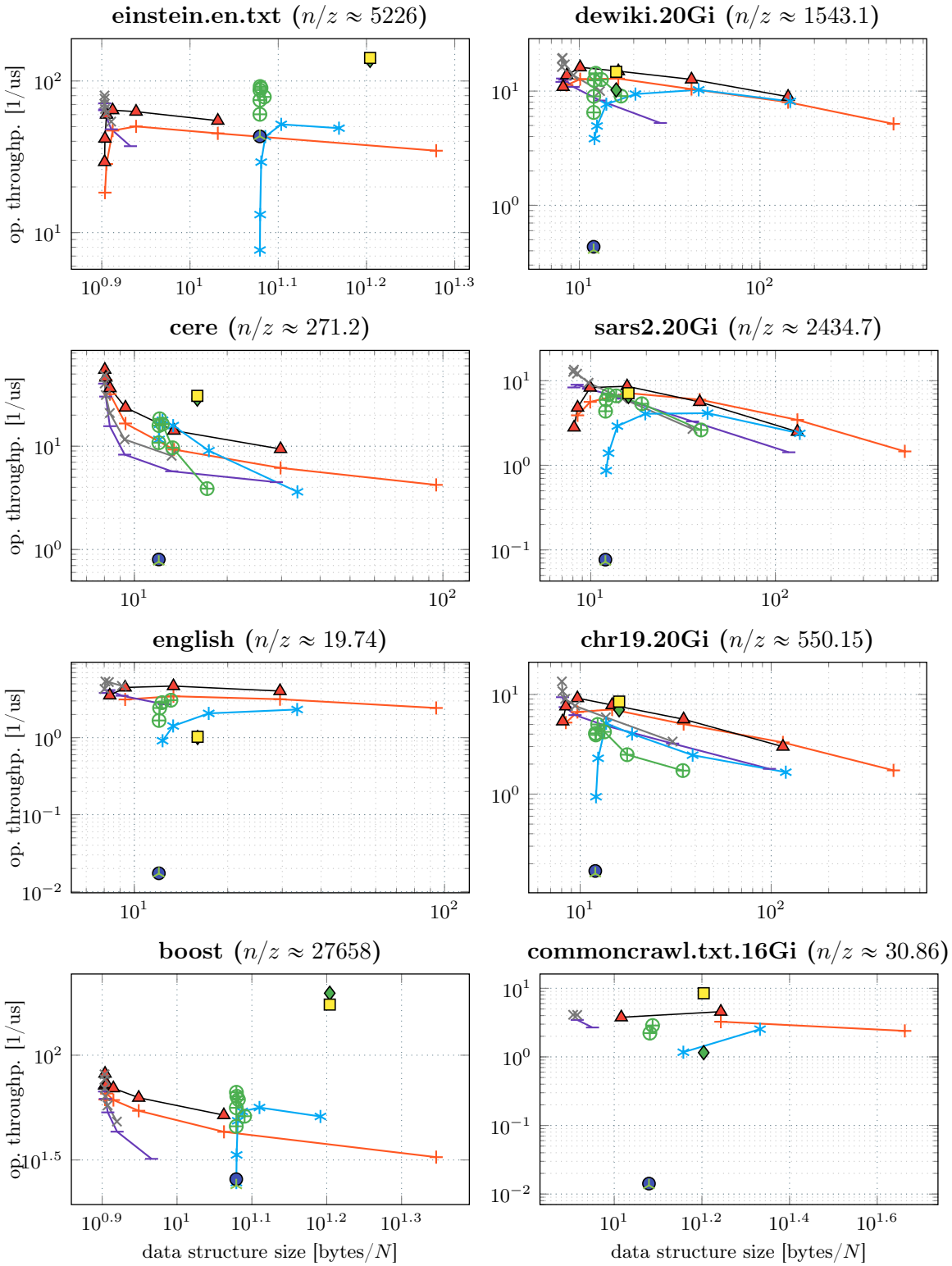


Figure A.1: Insert- and query throughput versus data structure size.

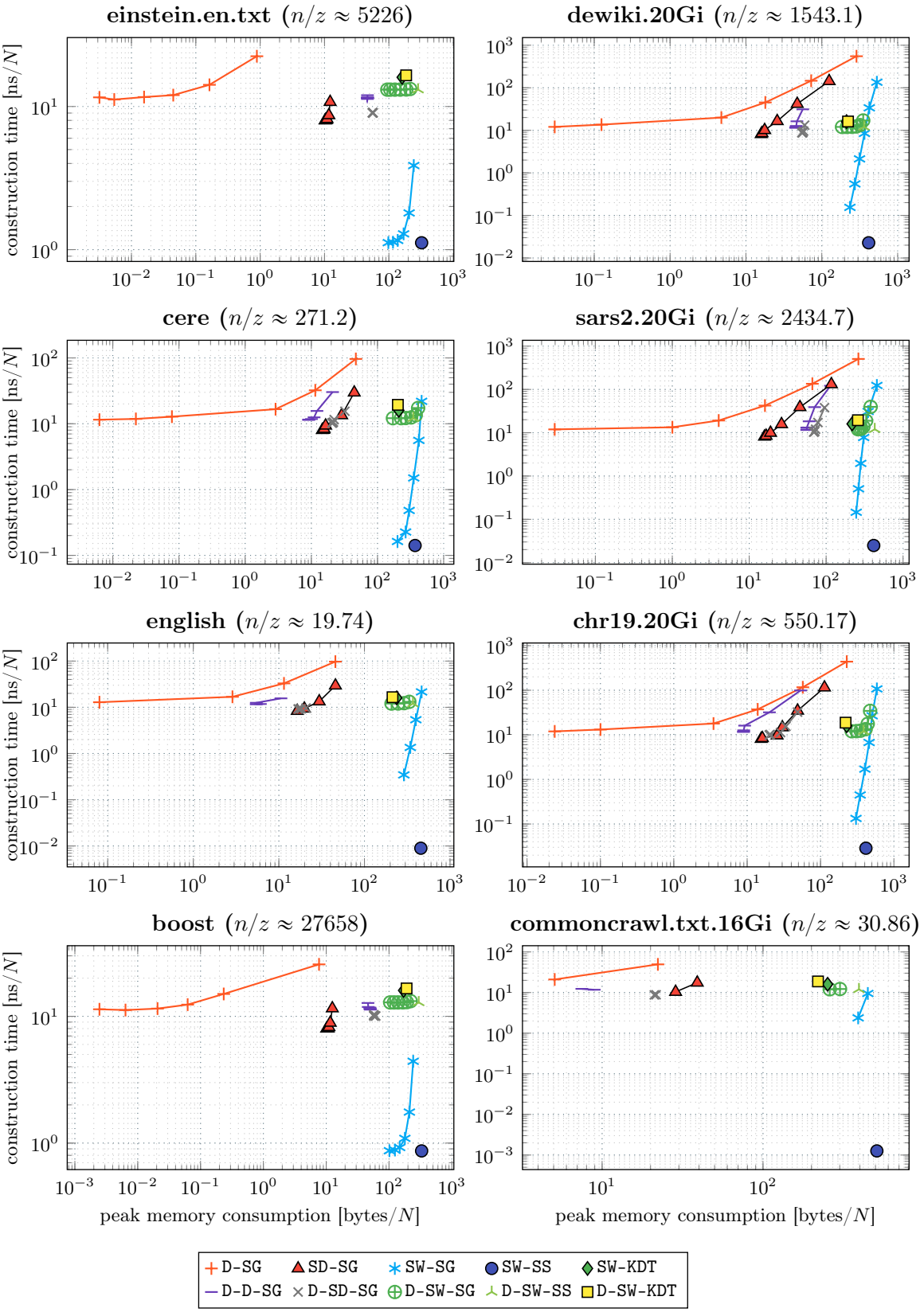


Figure A.2: Construction time versus peak memory consumption during the construction.

Bibliography

[1] Pankaj Agarwal and Jeff Erickson. Geometric Range Searching and Its Relatives. In *Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference on Discrete and Computational Geometry: Ten Years Later, July 14–18, 1996, South Hadley, MA, USA*, volume 223, pages 342–351, 1997.

[2] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. Engineering In-place (Shared-memory) Sorting Algorithms. *ACM Transactions on Parallel Computing*, 9(1):2:1–2:62, 2022.

[3] Jon Louis Bentley, Daniel Dominic Sleator, Robert Endre Tarjan, and Victor K. Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[4] Philip Bille, Patrick Hagge Cording, Johannes Fischer, and Inge Li Gørtz. Lempel-Ziv Compression in a Sliding Window. In *Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, Warsaw, Poland*, volume 78 of *Leibniz International Proceedings in Informatics*, pages 15:1–15:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[5] Anselm Blumer, J. Blumer, David Haussler, Ross M. McConnell, and Andrzej Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.

[6] Michael Burrows and David J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. In *Technical Report DEC*, volume 124, 1994.

[7] Francisco Claude and Gonzalo Navarro. Self-Indexed Grammar-Based Compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.

[8] Maxime Crochemore and Wojciech Rytter. Efficient Parallel Algorithms to Test Square-Freeness and Factorize Strings. *Information Processing Letters*, 38(2):57–60, 1991.

[9] Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. *Request for Comments*, 1951:1–17, 1996.

[10] Patrick Dinklage, Johannes Fischer, Alexander Herlez, Tomasz Kociumaka, and Florian Kurpicz. Practical Performance of Space Efficient Data Structures for Longest Common Extensions. In *28th Annual European Symposium on Algorithms, ESA 2020, September 7-9, Pisa, Italy*, volume 173 of *Leibniz International Proceedings in Informatics*, pages 39:1–39:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[11] Patrick Dinklage, Johannes Fischer, and Nicola Prezza. Top-k Frequent Patterns in Streams and Parameterized-Space LZ Compression. In *22nd International Symposium on Experimental Algorithms, SEA 2024, July 23-26, Vienna, Austria*, volume 301 of *Leibniz International Proceedings in Informatics*, pages 9:1–9:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024.

[12] Jonas Ellert. Sublinear Time Lempel-Ziv (LZ77) Factorization. In *Proceedings of the 30th International Symposium on String Processing and Information Retrieval, SPIRE 2023, September 26-28, Pisa, Italy*, volume 14240 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2023.

[13] Jonas Ellert, Johannes Fischer, and Max Rishøj Pedersen. New Advances in Rightmost Lempel-Ziv. In *Proceedings of the 30th International Symposium on String Processing and Information Retrieval, SPIRE 2023, September 26-28, Pisa, Italy*, volume 14240 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2023.

[14] Martin Farach and S. Muthukrishnan. Optimal Parallel Dictionary Matching and Compression (Extended Abstract). In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 95, July 17-19, Santa Barbara, California, USA*, pages 244–253. Association for Computing Machinery, 1995.

[15] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[16] Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via Small-Space Multiple-Pattern Matching. In *Proceedings of the 23rd Annual European Symposium on Algorithms, ESA 2015, September 14-16, Patras, Greece*, volume 9294 of *Lecture Notes in Computer Science*, pages 533–544. Springer, 2015.

[17] Johannes Fischer and Volker Heun. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In *Proceedings of the Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, July 5-7, Barcelona, Spain*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.

[18] Travis Gagie. Space-efficient RLZ-to-LZ77 conversion, 2022. `https://arxiv.org/abs/2211.13254`.

[19] Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. LZ77-Based Self-indexing with Faster Pattern Matching. In *Proceedings of the 11th Latin American Symposium on Theoretical Informatics, LATIN 2014, March 31-April 4, Montevideo, Uruguay*, volume 8392 of *Lecture Notes in Computer Science*, pages 731–742. Springer, 2014.

[20] Florin Ghido. QLFC - A Compression Algorithm Using the Burrows-Wheeler Transform. In *2005 Data Compression Conference, DCC 2005, 29-31 March, Snowbird, UT, USA*, page 459. IEEE Computer Society, 2005.

[21] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[22] Torben Hagerup. Sorting and Searching on the Word RAM. In *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science, STACS 98, February 25-27, Paris, France*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer, 1998.

[23] Charles A. R. Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321—322, 1961.

[24] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[25] Richard M. Karp and Michael O. Rabin. Efficient Randomized Pattern-Matching Algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[26] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001, July 1-4, Jerusalem, Israel*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001.

[27] Dominik Kempa. Optimal Construction of Compressed Indexes for Highly Repetitive Texts. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, January 6-9, San Diego, California, USA*, pages 1344–1357. Society for Industrial and Applied Mathematics, 2019.

[28] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, June 23-26, Phoenix, AZ, USA*, pages 756–767. ACM, 2019.

[29] Dominik Kempa and Tomasz Kociumaka. Lempel-Ziv (LZ77) Factorization in Sublinear Time, 2024. https://arxiv.org/abs/2409.12146.

[30] Dominik Kempa and Dmitry Kosolobov. LZ-End Parsing in Linear Time. In *Proceedings of the 25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, Vienna, Austria*, volume 87 of *Leibniz International Proceedings in Informatics*, pages 53:1–53:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[31] Dominik Köppl. Non-Overlapping LZ77 Factorization and LZ78 Substring Compression Queries with Suffix Trees. *Algorithms*, 14(2):44, 2021.

[32] Dmitry Kosolobov, Daniel Valenzuela, Gonzalo Navarro, and Simon J. Puglisi. Lempel-Ziv-Like Parsing in Small Space. *Algorithmica*, 82(11):3195–3215, 2020.

[33] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[34] Abraham Lempel and Jacob Ziv. On the Complexity of Finite Sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

[35] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, January 22-24, San Francisco, California, USA*, pages 319–327. Society for Industrial and Applied Mathematics, 1990.

[36] G. Nigel Martin. * Range encoding: an algorithm for removing redundancy from a digitised message, 1979. https://api.semanticscholar.org/CorpusID:17358601.

[37] Gonzalo Navarro. Indexing Highly Repetitive String Collections, Part II: Compressed Indexes. *ACM Computing Surveys*, 54(2):26:1–26:32, 2022.

[38] Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.

[39] Ge Nong, Sen Zhang, and Wai Hong Chan. Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

[40] Luís M. S. Russo, Ana Sofia D. Correia, Gonzalo Navarro, and Alexandre P. Francisco. Approximating Optimal Bidirectional Macro Schemes. In *Data Compression Conference, DCC 2020, March 24-27, Snowbird, UT, USA*, pages 153–162. Institute of Electrical and Electronics Engineers, 2020.