Reverse-Safe Data Structures for Text Indexing*

Giulia Bernardini † Huiping Chen ‡ Gabriele Fici § Grigorios Loukides ‡
Solon P. Pissis ¶

Abstract

We introduce the notion of reverse-safe data structures. These are data structures that prevent the reconstruction of the data they encode (i.e., they cannot be easily reversed). A data structure D is called *z-reverse-safe* when there exist at least z datasets with the same set of answers as the ones stored by D. The main challenge is to ensure that D stores as many answers to useful queries as possible, is constructed efficiently, and has size close to the size of the original dataset it encodes. Given a text of length n and an integer z, we propose an algorithm which constructs a z-reverse-safe data structure that has size $O(n)$ and answers pattern matching queries of length at most d optimally, where d is maximal for any such z-reverse-safe data structure. The construction algorithm takes $O(n^\omega \log d)$ time, where ω is the matrix multiplication exponent. We show that, despite the n^ω factor, our engineered implementation takes only a few minutes to finish for million-letter texts. We further show that plugging our method in data analysis applications gives insignificant or no data utility loss. Finally, we show how our technique can be extended to support applications under a realistic adversary model.

1 Introduction

Data structures organize data allowing for their efficient access and modification. They are thus the workhorse of many data analysis applications, such as clustering and outlier detection (e.g., through indexes for k-nearest neighbors join queries [9]), frequent pattern mining (e.g., through FP-trees [31]), document retrieval (e.g., through inverted indexes [48]), graph pattern matching (e.g., through graph indexes [68]), and range search in databases (e.g., through R-trees [29]).

These applications are often fueled by data collected from individuals, such as location, genomic, or customer data, and have led to justified privacy concerns [59]. To alleviate these concerns and comply with legislation such as HIPAA [19] in the US and GDPR [52] in the EU, it is necessary to guarantee that using data structures does not lead to the reconstruction of the stored individuals' data. This is a fundamentally different privacy goal than that of existing privacy-preserving techniques, such as anonymization [16, 15, 72, 32], sanitization [67, 25, 30, 10, 46, 6], query auditing [51], or access control [7]. Anonymization aims at preventing the disclosure of individuals' identities and/or sensitive information. Sanitization aims at preventing the mining of confidential knowledge. Query auditing aims at preventing answering aggregate queries that leak private information. Access control is the selective restriction of access to some parts of a database. Our privacy goal is also different from that of encryption techniques, such as searchable encryption [8, 43, 54], which aim at preventing unauthorized parties from accessing the data.

We consider a setting where a large group of users want to query a dataset directly via a data structure which prevents the reconstruction of the data. To this end, we introduce a novel encoding model that enables the construction of *reverse-safe data structures* (RSDSs). The ultimate aim of an RSDS is to make the reconstruction of a dataset sufficiently unlikely, so that an adversary cannot infer the dataset based on the query answers, but at the same time the RSDS stores as many answers to useful queries as possible in order to support applications. In addition, the RSDS should be constructed efficiently and have size close to the size of the original dataset it encodes. Our idea is inspired by *encoding data structures* (EDSs) [55]. The ultimate aim of an EDS is to break the information-theoretical lower bound, which is required to store a dataset, by storing only the answers to useful queries (e.g., range queries [22, 27] or nearest largest value queries [33]).

Given a data structure D, we denote by $\mathcal{A}_D$ its set of *consistent* datasets: all datasets with the same set of answers as the answers stored by D. Let us

---

[*] GB was partially supported by a research internship at CWI. HC was supported by a CSC scholarship. GF was partially supported by the Italian MIUR project PRIN 2017K7XPAN.

[†] Università di Milano - Bicocca, Italy and CWI, The Netherlands. giulia.bernardini@unimib.it

[‡] King's College London, United Kingdom. {huiping.chen,grigorios,loukides}@kcl.ac.uk

[§] Università degli Studi di Palermo, Italy. gabriele.fici@unipa.it

[¶] CWI, The Netherlands. solon.pissis@cwi.nl

denote $\alpha_D = |\mathcal{A}_D|$. Given an integer threshold $z > 1$, which we call the *privacy threshold*, we say that D is z-RSDS if and only if $\alpha_D \geq z$. A large z implies strong data privacy because an adversary cannot distinguish between the $\alpha_D \geq z$ consistent datasets, which implies that it is less likely that the adversary infers the dataset used to construct D in the first place. Still, it could be the case that D stores answers to many useful queries.

In this work, we consider string data (sometimes called text, word, or document depending on the context). A string is a sequence of letters from an alphabet. A string may represent various types of confidential information about individuals, including their movement history [65], diagnosed diseases [60], purchased products [63], or DNA sequence [47]. Our goal is to construct a z-RSDS for string data which allows for decision and counting pattern matching queries to be accurately and efficiently answered. Decision queries are fundamental for intrusion detection [44], activity monitoring [67], as well as for cataloguing human genetic variation [5], while counting queries are fundamental for pattern mining that is central in application domains ranging from bioinformatics [58] to marketing [46] and to public health [4].

Pattern matching queries in strings are answered efficiently by means of indexing data structures. These structures enable fast access to the substrings of a string, which is important in many data analysis applications [28]. The main idea behind indexing a string S for efficient substring querying is that every substring of S is a prefix of some suffix of S. Indexing data structures thus arrange the suffixes of S lexicographically in an ordered tree data structure. One popular such data structure is the *suffix tree* [69]. The suffix tree of S is the compacted trie of all the suffixes of S. The term compacted refers to the fact that it reduces the number of nodes by replacing each maximal branchless path segment with a single edge, and it uses intervals over S to store the labels of these edges. This ensures that the suffix tree has size linear in $|S|$: it has no more than $2|S|$ nodes. Importantly, the suffix tree answers several types of pattern matching queries over S in optimal time; see [28] for a nice exposition.

However, the suffix tree of S, which provides (random) access to *all* substrings of S, is *not* a z-RSDS, because it uniquely represents S. The *privacy-utility trade-off* we consider here is thus to provide access only to the substrings of S whose length is at most d, for some $d \in [1, |S|)$. In particular, we want our z-RSDS to support the following types of on-line queries.

Decision Query: check if a string P of length $m \leq d$ is a substring of S.

Counting Query: count the occurrences of a string P of length $m \leq d$ in S.

Given a string S and a privacy threshold z, the computational challenge is to compute the *maximal* d for which a z-RSDS for indexing S can be constructed. The maximality of d offers *data utility*, since any query for a substring of S of length d or less has the same answer, irrespectively of whether it is posed on S or on the z-RSDS. The fact that the data structure is z-reverse-safe offers *data privacy*, since the probability that an adversary infers S, based solely on knowledge of the z-RSDS, is no more than $1/z$.

We are now in a position to formally define the main computational problem considered in this paper[1].

Problem 1. *Given a string S of length n and a privacy threshold $1 < z \leq n^c$, for some constant $c \geq 1$, construct a z-RSDS that answers decision and counting pattern matching queries for any pattern of length $m \leq d$, such that d is maximal, or output* FAIL *if no such d exists.*

Our Contributions

The main theoretical result of this paper is the following (ω denotes the matrix multiplication exponent[2]).

Theorem 1.1. *Given a string S of length n, there exists an $O(n^\omega \log d)$-time algorithm to construct an $O(n)$-sized z-RSDS over S for a maximal d that answers decision and counting pattern matching queries, for any pattern of length $m \leq d$, in the optimal $O(m)$ time per query. The algorithm outputs* FAIL *if no such d exists.*

The main ingredients of our construction algorithm include (truncated) suffix trees [50, 69], a combinatorial theorem on de Bruijn graphs [34, 36], and fast matrix multiplication [70, 41]. To the best of our knowledge we are the first to combine these ingredients. We show that, despite the n^ω factor, our engineered implementation can construct z-RSDSs over million-letter texts in only a few minutes. To achieve this practical performance, we rely on further theoretical insight. We also show that plugging our method in data analysis applications gives insignificant or no data utility loss. Finally, we show how our technique can be extended at no extra cost to construct a z-RSDS that supports applications under a realistic adversary model, in which the adversary knows an arbitrary-length substring of S.

Organization of the Paper

---

[1] The problem of inferring a string from a text indexing data structure (see [38] and references therein) is conceptually related but fundamentally different to the problem investigated here.

[2] At the time of writing this paper, $\omega < 2.373$ [70, 41].

The basic definitions and notation are introduced in Section 2. In Section 3, we propose a z-RSDS for text indexing. In Section 4, we present our construction algorithm. We then describe a series of practical improvements in Section 5. In Section 6, we present our implementation and extensive experimental results. In Section 7, we discuss how to construct an adapted version of our z-RSDS under an adversary model. We conclude this paper in Section 8 with some open questions.

2 Definitions and Notation

An *alphabet* Σ is a finite non-empty set whose elements are called *letters*. A *string* is a sequence of letters from Σ. We fix a string $S = S[0] \cdots S[n-1]$ over $\Sigma = \{1, \ldots, n^{O(1)}\}$. The *length* of S is denoted by $|S| = n$. We also assume that S contains at least two different letters, otherwise the problem considered in this paper is trivial. By $S[i\,..\,j] = S[i] \cdots S[j]$, we denote the *substring* of S starting at position i and ending at position j of S. A substring $S[i\,..\,j]$ is called a *prefix* if $i = 0$; it is called a *suffix* if $j = n-1$. Given a positive integer k, we denote by $(S)_{k,i}$ the length-k substring of S starting at position i, i.e., $(S)_{k,i} = S[i\,..\,i+k-1]$, for all $0 \le i < n-k+1$. A string P has an *occurrence* in S or, more simply, it *occurs* in S if $P = (S)_{|P|,i}$, for some i. An occurrence of P is thus characterized by its starting position i in S.

The *weighted de Bruijn graph* of order k over a string S of length n is a directed multigraph $G_{S,k} = (V_{S,k}, E_{S,k})$, where the set of vertices $V_{S,k}$ is the set of length-$(k-1)$ substrings of S and $E_{S,k}$ is the multiset of edges from vertex u to vertex v for every occurrence of u and v as consecutive length-$(k-1)$ substrings of S. More formally, there is a multi-edge $(u, v) \in E_{S,k}$ with multiplicity m if and only if $u[0] \cdot v = u \cdot v[k-2]$ and this string occurs in S exactly m times. Thus $G_{S,k}$ has exactly $n-k+1$ edges; in general, $G_{S,k}$ contains self-loops and multi-edges (inspect Fig. 3 for an example).

3 A z-RSDS for Text Indexing

Let S be a string of length n. For a positive integer d, we define a *d-substring* of S as a substring of length d of S, or a suffix of S whose length is less than d.

The *d-truncated suffix tree* of a string S, denoted by $\mathcal{T}_d(S)$, is a path-compacted trie representing every d-substring of S [50]. We make use of a terminating letter $\# \notin \Sigma$ for technical purposes. Formally, $\mathcal{T}_d(S)$ is a rooted tree satisfying the following conditions (see Fig. 1 for an example):

1. Each edge is labeled with a non-empty substring of string $S\#$ encoded as an $[i, j]$ interval over $[0, n]$.
2. Each internal node v, except possibly the root, has at least two children. The labels of edges from v to its children start with distinct letters.
3. Let $\mathcal{L}(v)$ denote the string obtained by concatenating labels on the path from the root to node v. For every d-substring U, there is exactly one leaf w such that $U = \mathcal{L}(w)$ (if $|U| = d$) or $U\# = \mathcal{L}(w)$ (if $|U| < d$). For each leaf w, there is at least one d-substring U such that $\mathcal{L}(w) = U$ or $\mathcal{L}(w) = U\#$.
4. Each node v other than the root has a counter that stores the number of substrings of string $S\#$ that are equal to $\mathcal{L}(v)$.

Therefore, the number of leaves is at most n and the total number of nodes is less than $2n$. Recall that the label of the edge between node u and its child v, denoted by *label*(u, v), is represented implicitly by an interval over $[0, n]$. Thus the space occupied by $\mathcal{T}_d(S)$ is $O(n)$. The children of internal nodes are indexed by the alphabet letters using perfect hashing to ensure $O(1)$-time access [23]. Importantly, $\mathcal{T}_d(S)$ supports the following on-line pattern matching operations:

Decision Query: Check if a string P of length $m \le d$ is a substring of S in $O(m)$ time.

Counting Query: Count the occurrences of a string P of length $m \le d$ in S in $O(m)$ time.

Theorem 3.1. ([50, 14]) *Given a string S of length n and $0 < d \le n$, $\mathcal{T}_d(S)$ has size $O(n)$ and it can be constructed in $O(n)$ time. $\mathcal{T}_d(S)$ answers decision and counting pattern matching queries, for any pattern of length $m \le d$, in the optimal $O(m)$ time per query.*

The following off-line operations are also supported:

Frequent Substrings: Find all most frequent substrings, for all lengths $1, 2, \ldots, d$, in $O(n)$ time.

Repeated Substrings: Find all longest repeated substrings of length at most d in $O(n)$ time.

Unique Substrings: Find all shortest unique substrings of length at most d in $O(n)$ time.

We next consider a different representation of $\mathcal{T}_d(S)$ towards defining the notion of z-reverse-safe data structure. If *label*(u, v) is represented explicitly by a string we denote the resulting data structure by $\mathsf{TRIE}_d(S)$. In this case, string S is not part of the data structure, and thus $\mathsf{TRIE}_d(S)$ *does not*, generally, define S uniquely.

Definition 3.1. (d-Equivalent Strings) *Given the set of all possible strings of length n over an alphabet Σ*

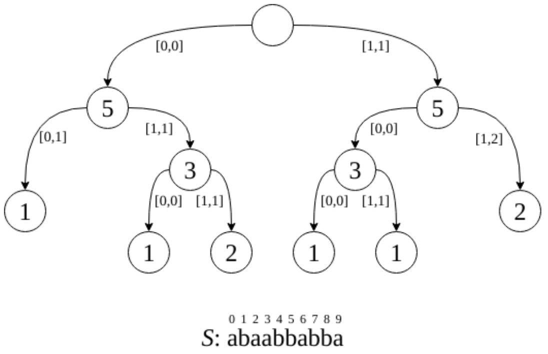


Figure 1: $\mathcal{T}_d(S)$ for $S =$ `abaabbabba` and $d = 3$. We omit edges whose labels start with letter # for clarity.

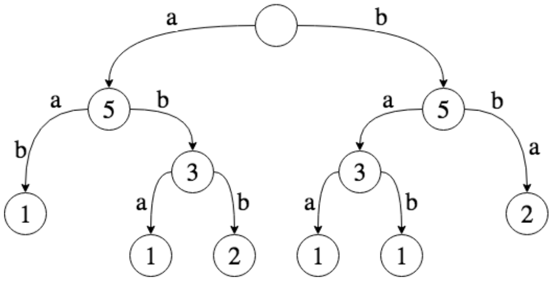


Figure 2: Let $S =$ `abaabbabba` and $S' =$ `abbaabbaba`. $S \sim_3 S' \iff \mathsf{TRIE}_3(S) = \mathsf{TRIE}_3(S')$.

*and an integer d, string S is d-*equivalent *to string S' if and only if $\mathsf{TRIE}_d(S) = \mathsf{TRIE}_d(S')$. In this case, we write $S \sim_d S'$ and say that S' is* consistent *with $\mathcal{T}_d(S)$.*

See Fig. 2 for an example. We can now formally define a z-reverse-safe data structure for text indexing.

DEFINITION 3.2. (z-RSDS FOR TEXT INDEXING) *Given an integer $z > 1$, $\mathcal{T}_d(\cdot)$ is called* z-reverse-safe *if and only if there exist at least z distinct strings that are consistent with $\mathcal{T}_d(\cdot)$.*

In what follows, we denote the set of strings that are consistent with $\mathcal{T}_d(S)$ by $\mathcal{A}_d(S)$, and $|\mathcal{A}_d(S)|$ by $\alpha_d(S)$, for $d \in [1, n]$. We omit (S) when this is clear from the context, and we also set $\alpha_0(S) = \infty$ for completeness.

4 Constructing z-RSDS

Clearly, $\mathcal{T}_n(S)$, the (non-truncated) suffix tree of S, has $\alpha_n = 1$ (i.e., it uniquely represents S), so it can never be a solution to Problem 1 since $z > 1$, by definition.

The following lemma is important for efficiency.

LEMMA 4.1. *The sequence $\alpha_0, \alpha_1, \ldots, \alpha_n$ is monotonically non-increasing.*

Proof. Let $\mathcal{A}_d$ be the set of strings consistent with $\mathcal{T}_d$, $d \in [1, n]$, and $\alpha_d = |\mathcal{A}_d|$. Further let S be any element of $\mathcal{A}_d$. By construction, if U is a d-substring of S, then $U = \mathcal{L}(w)$ or $U\# = \mathcal{L}(w)$, for some leaf w of $\mathcal{T}_d$. Every $(d-1)$-substring $S[i \mathinner{.\,.} i+d-2]$ of S is a prefix of the d-substring $S[i \mathinner{.\,.} i+d-1]$ of S. Thus string S is consistent with $\mathcal{T}_{d-1}$, the path-compacted trie that represents every such $(d-1)$-substring, and thus $S \in \mathcal{A}_{d-1}$. This implies the following relation: $\mathcal{A}_n \subseteq \mathcal{A}_{n-1} \subseteq \cdots \subseteq \mathcal{A}_1$. The statement follows directly from this relation and the fact that $\alpha_0 = \infty$. □

By Lemma 4.1, for increasing d, $\mathcal{T}_d(S)$ generally decreases α_d and increases utility. We thus need an algorithm to compute the maximum possible d that results in a z-RSDS. We next provide an algorithm, called z-RC (for z-RSDS Construction), to find this d.

```
Algorithm: $z$-RC
Input: string $S$ of length $n$ and integer $z > 1$
Output: $d$ and $\mathcal{T}_d(S')$, for some $S' \in \mathcal{A}_d$, or FAIL
1  $\ell \leftarrow 0$; $r \leftarrow n$;
2  if $\ell \geq r$ then
3      go to Line 10;
4  $d \leftarrow \lfloor \frac{\ell+r}{2} \rfloor$;
5  if $\alpha_d(S) \geq z$ then
6      $\ell \leftarrow d + 1$;
7  else
8      $r \leftarrow d$;
9  go to Line 2;
10 if $\ell > 0$ then
11     output $d \leftarrow \ell - 1$ and $\mathcal{T}_d(S')$, for some $S' \in \mathcal{A}_d$
12 else
13     output FAIL
```

As can be seen in the pseudocode, z-RC performs binary search on n (the length of S), computing α_d until d results in a z-RSDS and d is maximal. At this point, the z-RSDS $\mathcal{T}_d(S')$ is output, where S' is an element of $\mathcal{A}_d$ chosen at random, and the algorithm terminates. If $\ell > 0$ and $\alpha_{\ell-1} = z$, then $\alpha_{\ell-1}$ is the rightmost element that equals z. Even if such an element is not found, $n - \ell$ is the number of elements that are smaller than z.

The computational challenge is thus to implement the check of Line 5 efficiently and to find a consistent S' when this is possible (Line 11). To this end, we start with the following simple yet crucial observation.

OBSERVATION 1. *Given two strings X and Y, X is d-equivalent to Y if and only if X and Y have the same multisets of substrings of length i, for every $i \in [1, d]$.*

In the terminology of combinatorics on words, d-equivalence is known as *d-abelian equivalence* [37]. We report a lemma from [37], which gives several equivalent conditions that characterize d-equivalence.

LEMMA 4.2. ([37]) *Let X and Y be two strings of length at least d that have the same multiset of substrings of length d. The following are equivalent:*

1. *X and Y have the same multiset of substrings of length i for every $1 \leq i \leq d$;*

2. *X and Y have the same prefix of length $d-1$ and the same suffix of length $d-1$;*

3. *X and Y have the same prefix of length $d-1$;*

4. *X and Y have the same suffix of length $d-1$.*

Lemma 4.2 tells us that we should rely on the construction of weighted de Bruijn graphs over string S in order to compute $\alpha_d(S)$. The weighted de Bruijn graph of order d over string S is denoted by $G_{S,d} = (V_{S,d}, E_{S,d})$. Recall that its set of vertices $V_{S,d}$ is the set of distinct substrings of S of length $d-1$ (we implicitly identify a vertex by the string it represents) and there is an edge $(u, v) \in E_{S,d}$ with multiplicity m if and only if $u[0] \cdot v = u \cdot v[d-2]$ and this string occurs in S exactly m times. We borrow the terminology used in [39]. Let $d^-(u)$ and $d^+(u)$ be, respectively, the in- and out-degree of vertex u of $G_{S,d}$. Let s and t be the vertices of $G_{S,d}$ corresponding, respectively, to the prefix and to the suffix of length $d-1$ of S. Since any weighted de Bruijn graph is either Eulerian (if $s = t$) or semi-Eulerian (if $s \neq t$), we have that $d^+(u) = d^-(u)$ for all u with the possible exception of the two nodes s and t for which $d^-(s) = d^+(s) - 1$ and $d^+(t) = d^-(t) - 1$, if $s \neq t$. Clearly, S corresponds to an Eulerian path in $G_{S,d}$ that starts at s and ends at $t \neq s$ (if $s = t$, then it corresponds to an Eulerian cycle starting from s). The graph $G_{S,d}$ may contain other Eulerian paths (resp. cycles). Notice, however, that if two distinct Eulerian paths (resp. cycles) traverse the vertices of $G_{S,d}$ in the same order, but the edges in different order, then they give rise to the same string. We call these Eulerian paths (resp. cycles) *equivalent*. We summarize these observations into the following statement, which is crucial for the correctness of the z-RC algorithm.

Observation 2. *(a) If $S \sim_d S'$, then S' corresponds to an Eulerian path in $G_{S,d}$ that starts from vertex s and ends at vertex $t \neq s$ (if $s = t$, then it corresponds to an Eulerian cycle starting from s). (b) The number of distinct strings that are d-equivalent to S is the number of* non-equivalent *Eulerian paths (resp. cycles) in $G_{S,d}$.*

The number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d}$ can be computed via the following theorem, which is attributed to Hutchinson [34].

Theorem 4.1. ([34], cf. [39, 36]) *Let $A = (a_{uv})$ be the adjacency matrix of the weighted de Bruijn graph $G_{S,d} = (V_{S,d}, E_{S,d})$, with both $a_{uv} > 1$ (multi-edges) and $a_{uu} > 0$ (self-loops) allowed. Let $r_u = d^+(u) + 1$ if $u = t$ or $r_u = d^+(u)$ otherwise. The number of non-equivalent Eulerian paths starting at s and ending at t (resp. the number of non-equivalent Eulerian cycles starting at s, when $t = s$) is given by*

$$(4.1)\quad (\det L_{S,d}) \cdot \left(\prod_{u \in V_{S,d}} (r_u - 1)! \right) \cdot \left(\prod_{(u,v) \in E_{S,d}} a_{uv}! \right)^{-1},$$

where $L_{S,d} = (l_{uv})$ is the $|V_{S,d}| \times |V_{S,d}|$ matrix with $l_{uu} = r_u - a_{uu}$ and $l_{uv} = -a_{uv}$.

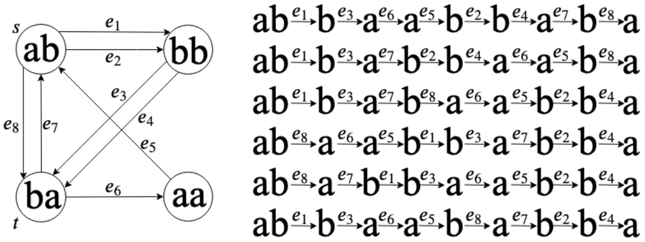

s
ab
e1
e2
bb
e3
e4
e5
e8
e7
ba
e6
aa
t
$ab \xrightarrow{e_1} b \xrightarrow{e_3} a \xrightarrow{e_6} a \xrightarrow{e_5} b \xrightarrow{e_2} b \xrightarrow{e_4} a \xrightarrow{e_7} b \xrightarrow{e_8} a$
$ab \xrightarrow{e_1} b \xrightarrow{e_3} a \xrightarrow{e_7} b \xrightarrow{e_2} b \xrightarrow{e_4} a \xrightarrow{e_6} a \xrightarrow{e_5} b \xrightarrow{e_8} a$
$ab \xrightarrow{e_1} b \xrightarrow{e_3} a \xrightarrow{e_7} b \xrightarrow{e_8} a \xrightarrow{e_6} a \xrightarrow{e_5} b \xrightarrow{e_2} b \xrightarrow{e_4} a$
$ab \xrightarrow{e_8} a \xrightarrow{e_6} a \xrightarrow{e_5} b \xrightarrow{e_1} b \xrightarrow{e_3} a \xrightarrow{e_7} b \xrightarrow{e_2} b \xrightarrow{e_4} a$
$ab \xrightarrow{e_8} a \xrightarrow{e_7} b \xrightarrow{e_1} b \xrightarrow{e_3} a \xrightarrow{e_6} a \xrightarrow{e_5} b \xrightarrow{e_2} b \xrightarrow{e_4} a$
$ab \xrightarrow{e_1} b \xrightarrow{e_3} a \xrightarrow{e_6} a \xrightarrow{e_5} b \xrightarrow{e_8} a \xrightarrow{e_7} b \xrightarrow{e_2} b \xrightarrow{e_4} a$

Figure 3: $G_{S,d}$ with $S =$ `abaabbabba` and $d = 3$ (on the left); and the set of d-equivalent strings (on the right).

Let us denote by $|S|_x$ the number of occurrences of a string x in S. Since, by definition, $r_u = |S|_u$ and $a_{uv} = |S|_{u \cdot v[k-2]} = |S|_{u[0] \cdot v}$, Eq. 4.1 is equivalent to

$$(4.2)\quad (\det L_{S,d}) \cdot \left(\prod_{u \in V_{S,d}} (|S|_u - 1)! \right) \cdot \left(\prod_{a \in \Sigma} |S|_{ua}! \right)^{-1}.$$

Eq. 4.2, together with a combinatorial study of the strings that belong to the same d-equivalence class, can be found in [36]. An example is provided with Fig. 3.

It is, however, not immediate that Eq. 4.1 (or the equivalent Eq. 4.2), involved in the check of Line 5 in algorithm z-RC, can be computed efficiently. We show this next starting with a known fact on de Bruijn graphs.

Fact 4.1. ([13]) *Given a string S of length n and $d < n$, its weighted de Bruijn graph $G_{S,d}$ can be constructed in $O(n)$ time.*

Lemma 4.3. $\det A$ *of an $n \times n$ non-singular matrix A can be computed in $O(n^\omega)$ time.*

Proof. The decomposition of a non-singular matrix $A = LU$, where L and U is a lower and upper triangular matrix, respectively, is known as LU decomposition and can be computed in the same time as matrix multiplication [11]. Given this decomposition, the determinant can be computed as $\det A = \det L \cdot \det U = \prod_{i=1}^{n} l_{ii} \cdot \prod_{i=1}^{n} u_{ii}$. This is because the determinant of any triangular matrix (such as L and U) is the product of its diagonal entries. ∎

Lemma 4.4. *Given* $\det L_{S,d}$*, the check of Line 5 in algorithm* z-RC *can be performed in* $O(n \log n)$ *time.*

Proof. We unfold all factorials involved in the two products of Eq. 4.1. Let us first consider the leftmost product. Observe that the total number of multiplications involved is no more than n because the sum of out-degrees over all nodes of $G_{S,d}$ is no more than n. Moreover, observe that each factor of the product is represented by $\log n$ bits because its value is no more than n. We assume a word-RAM algorithm that takes $O(n_1 + n_2)$ arithmetic operations to multiply an n_1-bit integer by an n_2-bit integer [40] resulting in an $(n_1 + n_2)$-bit integer. Using a $\log n$-depth divide and conquer, we can multiply these n integers in time $O(\frac{n}{2^1}2^0 \log n + \frac{n}{2^2}2^1 \log n + \ldots + \frac{n}{2^{\log n}}2^{\log n-1} \log n) = O(n \log n)$. Using an analogous argument, the rightmost product can be computed in $O(n \log n)$ time, because $G_{S,d}$ has no more than n edges, which implies that the product has at most n factors.

The leftmost product results in an $(n \log n)$-bit integer (we multiply n $\log n$-bit integers). By Hadamard's inequality [26] an upper bound on the value of $\det L_{S,d}$ is $B^n \cdot n^{n/2}$, where B is an upper bound on the values in $L_{S,d}$. Since here $B \leq n$, an upper bound on $\det L_{S,d}$ is $n^n \cdot n^{n/2} = n^{3n/2}$, which can be expressed using $\log(n^{3n/2}) = 1.5n \log n$ bits. Multiplying $\det L_{S,d}$ by the leftmost product is thus done in $O(n \log n)$ time. The rightmost product also results in an $(n \log n)$-bit integer, which we multiply by z. Since $z \leq n^c$ is a $c \log n$-bit integer, this is done in $O(n \log n)$ time. Thus, Line 5 is checked in $O(n \log n)$ time if $\det L_{S,d}$ is known. ∎

We arrive at the main theoretical result.

Theorem 4.2. *Given a string* S *of length* n*, there exists an* $O(n^\omega \log d)$*-time algorithm to construct an* $O(n)$*-sized* z*-RSDS over* S *for a maximal* d *that answers decision and counting pattern matching queries, for any pattern of length* $m \leq d$*, in the optimal* $O(m)$ *time per query. The algorithm outputs* FAIL *if no such* d *exists.*

Proof. The correctness of z-RC algorithm follows by Lemma 4.1 and Observation 2. The correctness of querying follows by the definition of d-equivalent strings.

The construction time follows by Fact 4.1, Lemmas 4.3-4.4, Theorem 3.1, and the binary search cost over $[0, n]$. Specifically, the check of Line 5 is implemented in $O(n^\omega)$ time by Fact 4.1 and Lemmas 4.3-4.4. If we find a valid d, we choose an Eulerian path (resp. cycle) of $G_{S,d}$ to construct a string S' and then construct $\mathcal{T}_d(S')$ using Theorem 3.1 in $O(n)$ time (Line 11). The z-RSDS size and the time per query follow by Theorem 3.1. If no such d exists the algorithm outputs FAIL.

If we apply exponential search (instead of binary search), we get an $O(n^\omega \log d)$-time construction. ∎

Colbourn et al. [17] gave an algorithm allowing for sampling of a random arborescence rooted at a given node to be carried out in the same time as counting all such arborescences, which forms the basis of counting Eulerian paths and cycles in directed multigraphs. Hence, by plugging the algorithm of Colbourn et al. in our construction algorithm (Line 11), we can also choose a string $S' \sim_d S$ *randomly* in the same time complexity.

5 Engineering the z-RC Algorithm

In what follows we describe a series of practical improvements, which are based on theoretical insight.

5.1 Improvement I: Reducing the BS Interval.

Lemma 5.1. *Let* S *be a string and* $r(S)$ *be the length of a longest substring of* S *occurring at least twice in* S*.* $\mathcal{T}_d(S)$ *cannot be a* z*-RSDS over* S *if* $d \geq r(S) + 2$*.*

Proof. Let I be the set of substrings of length $r(S) + 2$ of string S. Having set I is a sufficient condition for the unique reconstruction of S from I [21, 12]. This implies that, if $d \geq r(S) + 2$, $\mathcal{T}_d(S)$ defines S in a unique way (i.e., $\alpha_d = 1$), and thus $\mathcal{T}_d(S)$ cannot be a z-RSDS (since by definition $z > 1$). ∎

Note that the upper bound of $r(S) + 1$ can be computed in $O(n)$ time using the suffix tree of S [20], which is much faster than computing the bounds found by an exponential search. This is because exponential search takes $O(n^\omega)$ time for each of its iterations. As a consequence of Lemma 5.1, we can reduce the binary search interval from $[0, n]$ to $[0, r(S) + 1]$ in $O(n)$ time. Furthermore, it is known that $r(S)$ tends to $\log_{|\Sigma|} n$ as n tends to infinity under a Bernoulli i.i.d. model (cf. [21]).

5.2 Improvement II: Checking Prefixes of S.

Lemma 5.2. *Let* S *be a string and* P *be a prefix of* S*. Further, let* $\mathcal{A}_d(P)$ *(respectively,* $\mathcal{A}_d(S)$*) be the set of strings that are consistent with* $\mathcal{T}_d(P)$ *(respectively, with* $\mathcal{T}_d(S)$*). It holds that* $\alpha_d(P) \leq \alpha_d(S)$*.*

Proof. We show the lemma by showing that for any string X and any letter a, $\alpha_d(X) \leq \alpha_d(Xa)$. This implies that $\alpha_d(P) \leq \alpha_d(S)$. Indeed, by Lemma 4.2, it follows that if X' is d-equivalent to X, then $X'a$ is d-equivalent to Xa. ∎

Lemma 5.2 lets us implement the check in Line 5 of the z-RC algorithm by operating on the prefixes of S. The length of a longest substring of every prefix P

of S occurring at least twice in P can be computed by means of the longest previous factor (LPF) array [18]. The LPF array gives, for each position i in S, the length of a longest substring occurring both at i and to the left of i in S. We can thus construct an array R, where $R[i]$ stores the length of a longest substring occurring at least twice in the prefix $P = S[0\,..\,i]$ of S, by traversing the LPF array. Then, we only need to perform the check $\alpha_d(P) \geq z$ when $d < R[i] + 2$. This is because of applying Improvement I on P. The LPF array, and thus array R, can be computed both in $O(n)$ time [18]. Note that $R[i] \leq R[i+1]$. Thus, having R, we can find (whether there exists) a prefix $P = S[0\,..\,i]$ satisfying $d < R[i] + 2$, for all d, in $O(n)$ time in total.

5.3 Improvement III: Sparse LU Decomposition. Let $G_{S,d} = (V_{S,d}, E_{S,d})$ be the weighted de Bruijn graph for which we must compute the determinant $\det L_{S,d}$. $L_{S,d}$ is a $|V_{S,d}| \times |V_{S,d}|$ non-singular matrix, where $|V_{S,d}|$ is the number of distinct substrings of length $d-1$ occurring in S. Hence we have that $|V_{S,d}| \leq \min(|\Sigma|^{d-1}, n-d+1)$. If $|V_{S,d}| = O(n^{1/\omega})$, then $\det L_{S,d}$ is computed in $O(n)$ time by Lemma 4.3. If $|V_{S,d}| = \Theta(n)$, then $L_{S,d}$ is sparse: it has no more than $|V_{S,d}|+n-d+1$ non-zero elements, because in the worst case there is a non-zero element for each edge and there are $n-d+1$ edges with multiplicity 1. Thus, in any case, $L_{S,d}$ cannot contain more than $2n-d+1$ non-zero elements. We can therefore employ highly-optimized algorithms for sparse LU decomposition (e.g., [24, 35]) to compute $\det L_{S,d}$ efficiently. Let flops(XY) be the number of multiplications of non-zero elements performed while computing the product XY by conventional matrix multiplication. The algorithm of [24], for instance, takes $O(\text{flops}(LU)+m)$ time to compute the LU decomposition of a matrix with m non-zero elements. Thus, in our case, computing $\det L_{S,d}$ takes $O(\text{flops}(LU)+n)$ time.

6 Implementations and Experiments

6.1 Implementations. We have implemented the following algorithms in `C++`: (I) z-RC with Improvement III; (II) z-RCB (for Binary search interval reduction), which implements Improvements I and III; and (III) z-RCBP (for Binary search interval reduction and Prefix checking), which implements Improvements I, II, and III. We have omitted the results of the versions of the algorithms without Improvement III, because they were too slow to be practical.

For Improvement II, we have combined the idea described in Section 5.2 with exponential search: we start from an initial prefix P_0 of S that has length $|P_0| = \kappa$ and use it to perform the check in Line 5 of our algorithm in Section 4. Due to Lemma 5.2, we know that $\alpha_d(P_0) \geq z$ implies $\alpha_d(S) \geq z$; we thus check if $\alpha_d(P_0) \geq z$, because this is clearly more efficient than checking $\alpha_d(S) \geq z$. If $\alpha_d(P_0) < z$, $\alpha_d(S) \geq z$ may or may not hold. In this case, we consider a longer prefix of S that has length $|P_1| = 2^1 \cdot \kappa$ and proceed similarly. Clearly, significant computational savings can be brought when the last considered prefix P_i has small length $|P_i| = 2^i \cdot \kappa$, while in the worst case $P_i = S$, and the total cost of our algorithm with Improvement II is twice the cost of the algorithm without it due to doubling. We also apply Improvement I on these prefixes: if $d \geq R[i] + 2$ for prefix P_i, we do not check $\alpha_d(P_i) \geq z$, because Lemma 5.1 already ensures that $\alpha_d(P_i) = 1 < z$.

Dataset	**Data domain**	**Total length n**	**Alphabet size $\lvert\Sigma\rvert$**
MSN	Web	4,698,764	17
EC	Genomic	4,641,652	4
PR	Genomic	446,246 (27 strings)	4
SYN$_{50M}$	Synthetic	50,000,000	10

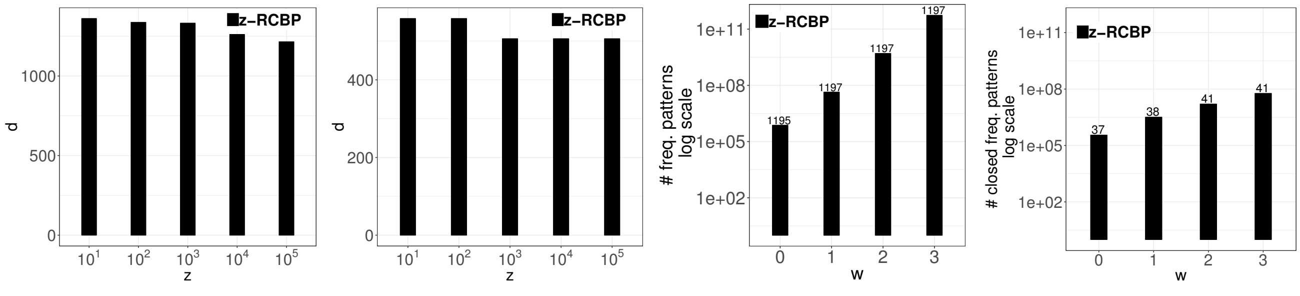
Table 1: Characteristics of datasets used.

For Improvement III, we used the Sparse LU decomposition function of the open-source Eigen library (v. 3.3.7) [1], which is based on the algorithm of [35], to compute $\det L_{S,d}$.

6.2 Experimental Setup and Datasets. We have evaluated z-RC, z-RCB, and z-RCBP in terms of data utility and efficiency. We do not compare our methods to existing approaches, because they are not alternatives to our work as mentioned in Section 1.

We used the following publicly available datasets: MSNBC (MSN), which contains page categories visited by users on `msnbc.com` over a 24-hour period; the complete genome of *Escherichia coli* (EC); and a dataset containing 27 Primate mitochondrial genomes (PR). MSN was used in [25, 30, 46], EC was used in [6], and PR was used in [64]. We also generated a uniformly random string of length 50M over an alphabet of size 10, and used its prefixes of length $1\text{M}, \ldots, 50\text{M}$ as synthetic datasets, referred to as $\text{SYN}_{1\text{M}}, \ldots, \text{SYN}_{50\text{M}}$, respectively. Each dataset contains a single string, except for PR which contains 27 strings (one for each mitochondrial genome). In PR, we applied our methods to each string independently. Table 1 summarizes the characteristics of the datasets.

To evaluate data utility, we report the length d found by our methods for different values of z, and also investigate the accuracy of performing two classes of data analysis applications: *pattern mining* [73] and *phylogenetic tree reconstruction* [64]. Unlike decision and counting pattern matching queries of length at



(a) EC (b) MSN (c) EC (d) EC

Figure 4: Length d for different z values for (a) EC and (b) MSN. Number of (c) frequent patterns and (d) closed frequent patterns with up to $w \in [0,3]$ wildcards mined from EC using $\mathsf{minSup} = 1.8 \cdot 10^{-6}$. The length of the longest mined pattern is on the top of each bar.

most d, which are answered exactly using the z-RSDS constructed by our methods, these applications are not guaranteed to be performed accurately on the output encoding. Yet, we show that plugging in our approach gives insignificant or no data utility loss in these applications.

We now discuss each of these applications.

(Closed) Frequent Pattern Mining. Frequent patterns and closed frequent patterns in string datasets model knowledge that aids decision making [2, 58] and can be used for data classification and clustering [73]. Given a string S and a user-specified threshold minSup, a pattern is *frequent* if its relative frequency in S, also referred to as *support*, is at least minSup. A frequent pattern of S is *closed* if none of its superstrings has the same relative frequency in S. Closed frequent patterns are typically fewer than the frequent ones and they are mined much more efficiently. Their benefit is that they uniquely determine the set of frequent patterns and their exact frequency. Our methods allow mining the frequent and closed frequent patterns of length at most d and only those. Thus, our methods preserve data utility well when the d computed is sufficiently large for low minSup values. In our experiments, we used the algorithm of [2] to mine a more general class of frequent and closed frequent patterns having up to $w \in [0,3]$ occurrences of a wildcard letter $\diamond$. A pattern with wildcards occurs in a string S if it is a subtring of S after replacing the wildcard letters with alphabet letters (e.g., pattern $\texttt{a}\diamond\diamond\texttt{e}$ occurs in $S = \texttt{b}\underline{\texttt{abde}}\texttt{b}$). Mining patterns with wildcards poses a further challenge to our approach, since (closed) frequent patterns with wildcards are a superset of the (closed) frequent patterns and are typically longer.

Phylogenetic Tree Reconstruction. A phylogenetic tree illustrates the evolutionary relationships among a set of species. To reconstruct phylogenetic trees, we applied the methodology in [64] on the PR dataset. That is, we compute the pairwise Average Common Substring with k mismatches (k-ACS) distance [66, 42] between the 27 strings in PR, using the ALFRED-G [64] algorithm, and then apply the neighbor-joining (NJ) algorithm [57] to reconstruct the phylogenetic tree. We apply the methodology to S and to $S' \sim_d S$, $S' \neq S$: intuitively, data utility is preserved well when the phylogenetic tree for S is similar to the one for S'. Following [64], we measured similarity using the normalized Robinson-Foulds (nRF) distance [56].

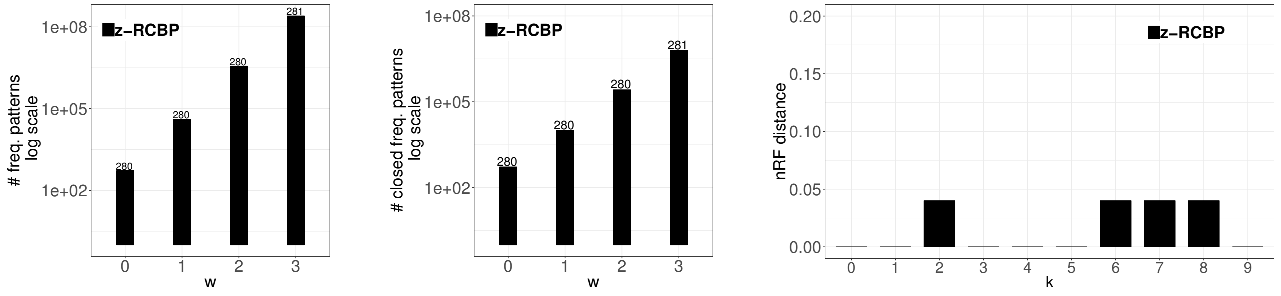
Unless otherwise stated, we used $z = 100$ and $\kappa = 1000$. All experiments ran on a machine with an Intel Xeon E5-2640 at 2.66GHz and 160GB RAM.

6.3 Data Utility.

Recall that our approach allows for answering pattern matching queries of length at most d in optimal time, and at the same time it prevents the reconstruction of the original dataset. In this section, we demonstrate that z-RCBP (and z-RC, z-RCB, which by design create the same output as z-RCBP), allow for other meaningful data analysis tasks to be applied with insignificant or no utility loss.

6.3.1 Length d.

We first show that z-RCBP provides access to very long substrings of the original dataset (i.e., the output length d is large). Figs. 4a and 4b show d for different values of the privacy threshold z in EC and MSN, respectively. As expected, d decreases when z increases. However, d is in the order of several hundreds, even when z is set to $100,000$. This implies (I) no accuracy loss for applying the pattern matching queries described in Section 3 on very long substrings and (II) strong privacy against dataset reconstruction.



(a) MSN (b) MSN (c) PR

Figure 5: Number of (a) frequent patterns and (b) closed frequent patterns with up to $w \in [0, 3]$ wildcards mined from MSN using $\mathsf{minSup} = 3.1 \cdot 10^{-3}$. The length of the longest mined pattern is on the top of each bar. (c) nRF distance vs. k between phylogenetic trees constructed for S and for $S' \sim_d S$, $S' \neq S$, using the k-ACS distance.

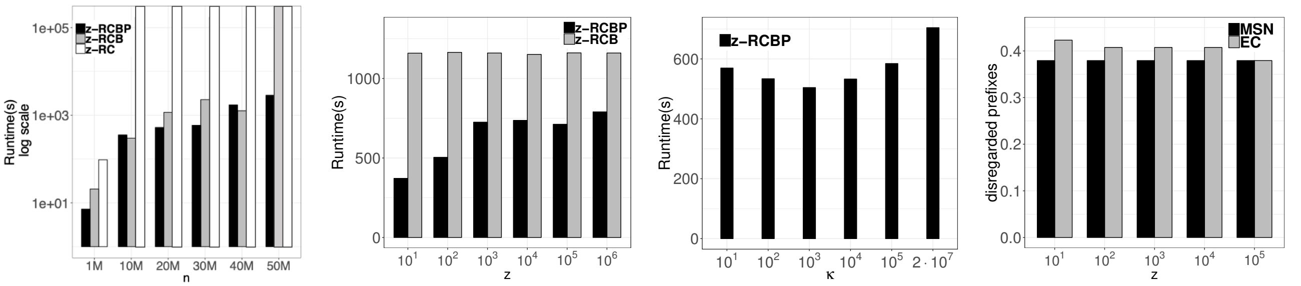
6.3.2 Frequent Pattern Mining. We demonstrate that z-RCBP allows for accurately mining frequent and closed frequent patterns with up to $w \in [0, 3]$ wildcard letters at very low minSup values. To this aim, we have computed the *smallest possible* value of minSup such that the mined frequent and closed frequent patterns have length no more than d. We denote this value by τ. Clearly, our method has no data utility loss for any $\mathsf{minSup} \geq \tau$. For EC, the smallest such minSup value (up to 8 decimal digits) was $\tau = 1.8 \cdot 10^{-6}$. Figs. 4c and 4d show the number and the maximal length of the mined patterns with $\mathsf{minSup} = \tau = 1.8 \cdot 10^{-6}$ for EC. For MSN, the smallest such minSup value (up to 4 decimal digits) was $\tau = 3.1 \cdot 10^{-3}$. The results for mining MSN with $\mathsf{minSup} = \tau = 3.1 \cdot 10^{-3}$ in Figs. 5a and 5b are qualitatively similar to those in Figs. 4c and 4d, respectively. The plots show that a large number of (potentially interesting) patterns can still be mined from the randomly selected S', even if some of them occur a small number of times in S (since τ was very low). Thus, our method permits the fundamental task of frequent pattern mining to be performed accurately.

6.3.3 Phylogenetic Tree Reconstruction. We next demonstrate that z-RCBP leads to phylogenetic trees constructed from $S' \sim_d S$, $S' \neq S$, which are either the same or very similar with respect to the nRF distance to the phylogenetic trees constructed from S. Fig. 5c shows the nRF distance between these trees. The trees were obtained using the k-ACS distance for different k values in $[0, 9]$ and the NJ algorithm as in [64]. Note that the tree constructed from S was the same to the one constructed from S' in six out of ten cases, implying no data utility loss, and in the remaining four cases the nRF had a very small value of 0.04, implying insignificant data utility loss for this fundamental bioinformatics task.

6.4 Runtime. In this section, we show that, despite the n^{ω} factor, z-RCBP takes only a few minutes to finish for million-letter texts. Fig. 6a shows the runtime of z-RC, z-RCB, and z-RCBP using the synthetic datasets as input. Recall that the largest synthetic dataset is $\mathrm{SYN}_{50\mathrm{M}}$ and the other datasets are prefixes of $\mathrm{SYN}_{50\mathrm{M}}$. z-RCBP was substantially more efficient than both z-RC and z-RCB and scaled better with the dataset size, confirming the necessity of Improvements I and II for being able to apply our methodology to large texts. On the other hand, z-RC did not finish within 48 hours for $\mathrm{SYN}_{10\mathrm{M}}, \ldots, \mathrm{SYN}_{50\mathrm{M}}$. In addition, z-RCB did not finish within 48 hours for $\mathrm{SYN}_{50\mathrm{M}}$, although it was slightly faster than z-RCBP for $\mathrm{SYN}_{10\mathrm{M}}$ and $\mathrm{SYN}_{40\mathrm{M}}$ because z-RCBP had to apply exponential search several times (see Section 6.1).

We also measured the runtime of z-RCB and z-RCBP for different z values (see Fig. 6b). We do not report the runtime of z-RC because it did not finish within 48 hours. The runtime of z-RCBP is much less when z is small, because z-RCBP considered fairly short prefixes. Specifically, z-RCBP was two times faster than z-RCB on average, and three times faster when $z = 10$. The runtime of z-RCB was not affected substantially by z. This is because z-RCB outputs the same d as z-RCBP does for all z values (i.e., constructs the same output) but it operates on the entire string S.

Next, we studied the impact of the initial prefix length κ on the runtime of z-RCBP, the only method that uses Improvement II (see Fig. 6c). The runtime of z-RCBP decreased when κ increased, but up to $\kappa = 1000$. Until then, prefixes were too short (i.e., the condition $\alpha_d(S') \geq z$ did not hold), so longer prefixes were considered. For $\kappa > 1000$, z-RCBP took more time because it is more expensive to check the condition on longer prefixes (e.g., z-RCBP took 40% more time when $\kappa = |S|$ compared to when $\kappa = 1000$).



(a) $\mathrm{SYN}_{1\mathrm{M}}, \ldots, \mathrm{SYN}_{50\mathrm{M}}$ (b) $\mathrm{SYN}_{20\mathrm{M}}$ (c) $\mathrm{SYN}_{20\mathrm{M}}$ (d) MSN and EC

Figure 6: Runtime vs. (a) n (z-RC and z-RCB did not finish within 48 hours for $\mathrm{SYN}_{10\mathrm{M}}, \ldots, \mathrm{SYN}_{50\mathrm{M}}$ and $\mathrm{SYN}_{50\mathrm{M}}$, respectively). Runtime vs. (b) z and (c) κ. (d) Ratio of disregarded prefixes vs. z.

Similar results were observed for MSN, PR, and EC (e.g., z-RCBP finished in less than 436 seconds).

6.5 Disregarded Prefixes. Last, we demonstrate that, applying Improvement I on the prefixes of S, which are used in Improvement II, allows for disregarding a large ratio of them from the computation. That is, we often avoid computing $\alpha_d(P)$ for a prefix P of S, because when $d \geq r(P) + 2$, we have that $\alpha_d(P) = 1 < z$ by Lemma 5.1. Specifically, Fig. 6d shows that the ratio of disregarded prefixes over all prefixes considered for MSN and EC is at least 0.38. The benefit of the improvement when this ratio is large is time efficiency, since computing $\alpha_d(P)$ to check whether $\alpha_d(P) \geq z$ can be expensive particularly for a long prefix P of S.

7 Application to an Adversary Model

In this section, we discuss an adapted version of our z-RSDS that can be applied to an adversary model alike those considered in [62, 45, 53, 49, 61].

7.1 Adversary Model. Our privacy goal is to limit the probability of inferring string S when the adversary possesses the following knowledge.

Definition 7.1. (Adversarial Knowledge) *A pair $\mathcal{K} = (\mathcal{T}_d(S'), \tilde{S})$, where $S' \sim_d S$ and $\tilde{S}$ is a (possibly empty) substring of S.*

The adversarial knowledge $\mathcal{K}$ is comprised of $\mathcal{T}_d(S')$, which is accessible by the adversary, and of $\tilde{S}$ which is the adversary's *background knowledge.* Background knowledge is obtained by an adversary, typically from external data sources and/or communication with individuals represented in the input dataset [62, 45, 53, 49, 61]. As it will become clear later, such knowledge may make a z-RSDS more likely to be reversed. Thus, when certain background knowledge is known or can be assumed, it should be modeled and taken into account in the construction of a z-RSDS to ensure that the z-RSDS remains sufficiently unlikely to reverse.

We model the background knowledge as a substring to capture manifested attacks [45, 61] in which the adversary observes an individual's actions within a timeframe. The actions are represented by $\tilde{S}$. For example, when S models the diagnoses in an individual's electronic health record, $\tilde{S}$ models the diagnoses assigned to the individual during a hospital visit, which may be known by a hospital employee [45]. Similarly, when S models an individual's credit card purchases, $\tilde{S}$ models the products purchased by the individual during a visit to a shop, which may be known by a shop employee [61]. $\tilde{S}$ may be specified by the data provider [6] or the data custodian [63], according to policies. Note that from $\mathcal{T}_d(S')$, the adversary can also learn (see Fig. 1): the length $n = |S|$, the maximal string depth d, and the suffixes of S of length at most $d-1$. Thus, we did not include such information in $\mathcal{K}$ explicitly.

An adversary may not be able to uniquely infer S, based on their knowledge $\mathcal{K}$. This is because they have to distinguish S among the set of strings that are d-equivalent to S and have $\tilde{S}$ as substring. In fact, the probability that the adversary infers S, based solely on their knowledge $\mathcal{K}$, is defined as follows.

Definition 7.2. (Inference Probability of S) *The* inference probability *of S, based on the knowledge $\mathcal{K}$, is defined as $\mathcal{P}(I_S \mid \mathcal{K}) = {}^{1}/_{|\mathcal{A}_{\mathcal{K}}|}$, where I_S is the event "the adversary infers S" and $\mathcal{A}_{\mathcal{K}}$ is the set of strings consistent with $\mathcal{T}_d(S')$ having $\tilde{S}$ as substring.*

$\mathcal{P}(I_S \mid \mathcal{K})$ is defined based on: (I) The fact that the adversary can construct all strings that are consistent with $\mathcal{T}_d(S')$ and contain $\tilde{S}$ as substring (see Section 7.2). (II) The *random worlds assumption* [3] (i.e., each such instance is equally likely). This assumption is followed by most related works (see [71] and references therein).

We aim at constructing a $\mathcal{T}_d(S')$, for some $S' \sim_d S$

chosen at random, that an adversary cannot use to infer S with sufficiently large $\mathcal{P}(I_S \mid \mathcal{K})$. We also require $\mathcal{T}_d(S')$ to have maximal d in order to support the operations discussed in Section 3 on larger substrings, thereby providing higher utility. This leads to the following computational problem.

Problem 2. *Given a string S of length n, a substring $\tilde{S}$ of S, and a privacy threshold $1 < z \leq n^c$, for some constant $c \geq 1$, construct a $\mathcal{T}_d(S')$ such that: (I) $S' \sim_d S$, (II) d is maximal, and (III) $\mathcal{P}(I_S \mid \mathcal{K}) \leq \frac{1}{z}$, for $\mathcal{K} = (\mathcal{T}_d(S'), \tilde{S})$; or output FAIL if no such d exists.*

7.2 Construction Algorithm. We next show how the z-RC algorithm for constructing a z-RSDS can be applied in an extended way to solve Problem 2. In this case, we need to account for $\mathcal{A}_\mathcal{K}$, a modified version of $\mathcal{A}_d$: a string S' is in $\mathcal{A}_\mathcal{K}$ if and only if it is d-equivalent to S and contains $\tilde{S}$ as a substring. In the graph formulation of the problem, we need to ensure that a path representing $\tilde{S}$ must always be visited. Thus, we modify the z-RC algorithm as follows.

Let $\alpha_\mathcal{K} = |\mathcal{A}_\mathcal{K}|$. Consider a binary search iteration for some value of d, in which we must check whether $\alpha_\mathcal{K} \geq z$. We first construct the weighted de Bruijn graph $G_{S,d}$. If $|\tilde{S}| \leq d$ all strings in $\mathcal{A}_d$ contain $\tilde{S}$ as a substring by construction and so we do not need to modify the algorithm for such an $\tilde{S}$. Intuitively, in this case, knowledge of $\tilde{S}$ is of no use to the adversary. We thus consider the case when $|\tilde{S}| > d$. A path $v_1 v_2 \dots v_h$ in $G_{S,d}$ represents a substring of length $h-1$. We remove the edges of a path $v_1 v_2 \dots v_h$ in $G_{S,d}$ representing *some occurrence* of $\tilde{S}$ in S, $|\tilde{S}| > d$. (There may be multiple such paths). We add a *shortcut edge* $e_{\tilde{S}} = (v_1, v_h)$ directed from node v_1 to node v_h to represent an occurrence of string $\tilde{S}$. Let us denote the resulting graph by $G_{S,d,\tilde{S}}$ (see Fig. 7 vs. Fig. 3).

Lemma 7.1. *(a) If $S \sim_d S'$ and $\tilde{S}$ is a substring of S', then S' corresponds to an Eulerian path in $G_{S,d,\tilde{S}}$ that starts from vertex s and ends at vertex $t \neq s$ (if $s = t$, then it corresponds to an Eulerian cycle starting from s). (b) $\alpha_\mathcal{K}$ is equal to the number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d,\tilde{S}}$.*

Proof. (a) Trivial. (b) We first observe that $G_{S,d,\tilde{S}}$ is Eulerian (resp. semi-Eulerian) by construction, because $G_{S,d}$ is Eulerian (resp. semi-Eulerian) and the procedure above does not change the parity of any vertex. Indeed, consider path $v_1 v_2 \dots v_h$ in $G_{S,d}$ representing the string $\tilde{S}$, which we replace with a shortcut edge $e_{\tilde{S}}$. Exactly one outgoing and one incoming edge is removed from each $v_2, \dots, v_{h-1}$; one outgoing edge is removed from v_1 and replaced with outgoing edge $e_{\tilde{S}}$, one incoming edge is removed from v_h and replaced with $e_{\tilde{S}}$ incoming. Moreover, since the multiplicity of any substring U of length d is given by the multiplicity of the edge from node $U[0\,.\,.\,d-2]$ to node $U[1\,.\,.\,d-1]$, the multiplicities of d-substrings are not affected by this transformation.

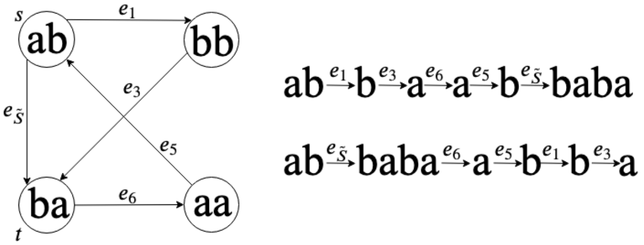

s
ab
e1
bb
e3
$e_{\tilde{S}}$
e5
ba
e6
aa
t
$ab \xrightarrow{e_1} b \xrightarrow{e_3} a \xrightarrow{e_6} a \xrightarrow{e_5} b \xrightarrow{e_{\tilde{S}}} baba$
$ab \xrightarrow{e_{\tilde{S}}} baba \xrightarrow{e_6} a \xrightarrow{e_5} b \xrightarrow{e_1} b \xrightarrow{e_3} a$

Figure 7: $G_{S,d,\tilde{S}}$ with $S =$ `abaabbabba`, $d = 3$, $\tilde{S} =$ `baba`, and $\alpha_\mathcal{K} = 2$.

To show that the number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d,\tilde{S}}$ is at most $\alpha_\mathcal{K}$, consider any Eulerian path (resp. cycle) in $G_{S,d,\tilde{S}}$. By definition, there is at least one occurrence of $\tilde{S}$ given by edge $e_{\tilde{S}}$, and it thus represents a string that belongs to $\mathcal{A}_\mathcal{K}$. Symmetrically, to show that $\alpha_\mathcal{K}$ is at most the number of non-equivalent Eulerian paths (resp. cycles) in $G_{S,d,\tilde{S}}$, consider a string $U \in \mathcal{A}_\mathcal{K}$. Among the (possibly multiple) Eulerian paths (resp. cycles) in $G_{S,d}$ that represent U, consider one that has $v_1 v_2 \dots v_h$ as subpath as representative of its equivalence class: such path exists because of Observation 2 and the properties of weighted de Bruijn graphs. This path corresponds to the path in $G_{S,d,\tilde{S}}$, where $v_1 v_2 \dots v_h$ is replaced with $v_1 v_h$. ◻

Theorem 7.1. *Problem 2 can be solved in $O(n^\omega \log d)$ time.*

Proof. The correctness of the construction algorithm follows by Lemma 7.1 and the fact that it is correct to apply binary or exponential search due to the monotonicity of $\alpha_\mathcal{K}$ (the monotonicity proof is almost identical to that of Lemma 4.1 and is thus omitted).

For a given d, finding a path $v_1 v_2 \dots v_h$ in $G_{S,d}$ and replacing it with $v_1 v_h$ can be done while constructing $G_{S,d}$ at no extra cost. Recall that $v_1 v_2 \dots v_h$ represents some occurrence of string $\tilde{S}$ in S, and that all occurrences of $\tilde{S}$ in S can be found in $O(n)$ time using the suffix tree of S [69]. The time complexity thus follows from the proof of Theorem 1.1. ◻

8 Final Remarks

We have introduced the notion of z-reverse-safe data structures and presented such a data structure for text indexing. Let us remark that our encoding model can be used in conjunction with other privacy-preserving techniques to ensure that certain privacy-utility trade-offs are maintained.

There are at least three open questions:

1. Can we improve the time complexity of the construction?

2. Can we have a faster construction algorithm for certain values of z?

3. Can we efficiently generalize the construction algorithm for the case when the adversary knows of a set of substrings of S (instead of a single substring)?

References

[1] Eigen library. `http://eigen.tuxfamily.org`.

[2] Hiroki Arimura and Takeaki Uno. An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. *Journal of Combinatorial Optimization*, 13(3):243–262, 2007.

[3] Fahiem Bacchus, Adam J. Grove, Joseph Y. Halpern, and Daphne Koller. From statistical knowledge bases to degrees of belief. *Artificial Intelligence*, 87(1-2):75–143, 1996.

[4] Imon Banerjee, Kevin Li, Martin Seneviratne, Michelle Ferrari, Tina Seto, James D Brooks, Daniel L Rubin, and Tina Hernandez-Boussard. Weakly supervised natural language processing for assessing patient-centered outcome following prostate cancer treatment. *Journal of the American Medical Informatics Association Open*, 2(1):150–159, 2019.

[5] David R. Bentley. Whole-genome re-sequencing. *Current Opinion in Genetics & Development*, 16(6):545–552, 2006.

[6] Giulia Bernardini, Huiping Chen, Alessio Conte, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. String sanitization: A combinatorial approach. In *ECML/PKDD*, 2019.

[7] Elisa Bertino, Gabriel Ghinita, and Ashish Kamra. *Access Control for Databases: Concepts and Systems*. Now Foundations and Trends, 2011.

[8] Bruhadeshwar Bezawada, Alex X. Liu, Bargav Jayaraman, Ann L. Wang, and Rui Li. Privacy preserving string matching for cloud computing. In *ICDCS*, pages 609–618, 2015.

[9] Christian Böhm and Florian Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowledge and Information Systems*, 6(6):728–749, 2004.

[10] Luca Bonomi, Liyue Fan, and Hongxia Jin. An information-theoretic approach to individual sequential data sanitization. In *WSDM*, pages 337–346, 2016.

[11] James R. Bunch and John E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974.

[12] Arturo Carpi and Aldo de Luca. Words and special factors. *Theoretical Computer Science*, 259(1-2):145–182, 2001.

[13] Bastien Cazaux, Thierry Lecroq, and Eric Rivals. Linking indexing data structures to de bruijn graphs: Construction and update. *Journal of Computer and System Sciences*, 104:165–183, 2019.

[14] Panagiotis Charalampopoulos, Costas S. Iliopoulos, Chang Liu, and Solon P. Pissis. Property suffix array with applications. In *LATIN*, pages 290–302, 2018.

[15] Rui Chen, Gergely Acs, and Claude Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *CCS*, pages 638–649, 2012.

[16] Rui Chen, Benjamin C.M. Fung, Bipin C. Desai, and Nériah M. Sossou. Differentially private transit data publication: A case study on the montreal transportation system. In *KDD*, pages 213–221, 2012.

[17] Charles J. Colbourn, Wendy J. Myrvold, and Eugene Neufeld. Two algorithms for unranking arborescences. *Journal of Algorithms*, 20(2):268–281, 1996.

[18] Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Walen. Computing the longest previous factor. *Eur. J. Comb.*, 34(1):15–26, 2013.

[19] U.S. Department of Health & Human Services. Health Insurance Portablility and Accountability Act. `https://aspe.hhs.gov/report/health-insurance-portability-and-accountability-act-1996`, 1996.

[20] Martin Farach-Colton. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143, 1997.

[21] Gabriele Fici, Filippo Mignosi, Antonio Restivo, and Marinella Sciortino. Word assembly through minimal forbidden words. *Theoretical Computer Science*, 359(1-3):214–230, 2006.

[22] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

[23] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $\mathcal{O}(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[24] John R. Gilbert and Tim Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific Computing*, 9(5):862–874, 1988.

[25] Aris Gkoulalas-Divanis and Grigorios Loukides. Revisiting sequential pattern hiding to enhance utility. In *KDD*, pages 1316–1324, 2011.

[26] Izrail S. Gradshteyn and Iosif M. Ryzhik. *Table of integrals, series, and products*. Elsevier/Academic Press, Amsterdam, seventh edition, 2007.

[27] Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and S. Rao Satti. Asymptotically optimal encodings of range data structures for selection and top-k queries. *ACM Transactions on Algorithms*, 13(2):28:1–28:31, 2017.

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

[29] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[30] Robert Gwadera, Aris Gkoulalas-Divanis, and Grigorios Loukides. Permutation-based sequential pattern hiding. In *ICDM*, pages 241–250, 2013.

[31] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Record*, 29(2):1–12, May 2000.

[32] Raymond D. Heatherly, Grigorios Loukides, Joshua C. Denny, Jonathan L. Haines, Dan M. Roden, and Bradley A. Malin. Enabling genomic-phenomic association discovery without sacrificing anonymity. *PLOS ONE*, 8:1–13, 02 2013.

[33] Michael Hoffmann, John Iacono, Patrick K. Nicholson, and Rajeev Raman. Encoding nearest larger values. *Theoretical Computer Science*, 710:97–115, 2018.

[34] Joan P. Hutchinson. On words with prescribed overlapping subsequences. *Utilitas Mathematica*, 7:241–250, 1975.

[35] John R. Gilbert Xiaoye S. Li James Weldon Demmel, Stanley C. Eisenstat and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20(3):720–755, 1999.

[36] Juhani Karhumäki, Svetlana Puzynina, Michaël Rao, and Markus A. Whiteland. On cardinalities of k-abelian equivalence classes. *Theoretical Computer Sciene*, 658:190–204, 2017.

[37] Juhani Karhumäki, Aleksi Saarela, and Luca Q. Zamboni. On a generalization of abelian equivalence and complexity of infinite words. *Journal of Combinatorial Theory, Series A*, 120(8):2189–2206, 2013.

[38] Juha Kärkkäinen, Marcin Piatkowski, and Simon J. Puglisi. String Inference from Longest-Common-Prefix Array. In *ICALP*, pages 62:1–62:14, 2017.

[39] Carl Kingsford, Michael C. Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11(1):21, 2010.

[40] Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998.

[41] François Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303, 2014.

[42] Chris-Andre Leimeister and Burkhard Morgenstern. Kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison. *Bioinformatics*, 30(14):2000–2008, 05 2014.

[43] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy keyword search over encrypted data in cloud computing. In *INFOCOM*, pages 1–5, 2010.

[44] Po-Ching Lin, Ying-Dar Lin, Yuan-Cheng Lai, and Tsern-Huei Lee. Using string matching for deep packet inspection. *IEEE Computer*, 41(4):23–28, 2008.

[45] Grigorios Loukides, Aris Gkoulalas-Divanis, and Bradley Malin. Anonymization of electronic medical records for validating genome-wide association studies. *Proceedings of the National Academy of Sciences USA*, 107(17):7898–7903, 2010.

[46] Grigorios Loukides and Robert Gwadera. Optimal event sequence sanitization. In *SDM*, pages 775–783, 2015.

[47] Bradley Malin and Latanya Sweeney. Determining the identifiability of DNA database entries. In *AMIA*, pages 537–541, 2000.

[48] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[49] Noman Mohammed, Benjamin C. M. Fung, Patrick C. K. Hung, and Cheuk-Kwong Lee. Centralized and distributed anonymization for high-dimensional healthcare data. *ACM Transactions on Knowledge Discovery from Data*, pages 18:1–18:33, 2010.

[50] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(1):87–101, 2003.

[51] Shubha U. Nabar, Krishnaram Kenthapadi, Nina Mishra, and Rajeev Motwani. A survey of query auditing techniques for data privacy. In Charu C. Aggarwal and Philip S. Yu, editors, *Privacy-Preserving Data Mining: Models and Algorithms*, pages 415–431. Springer US, 2008.

[52] European Parliament. General Data Protection Regulation. `http://data.consilium.europa.eu/doc/document/ST-9565-2015-INIT/en/pdf`.

[53] Giorgos Poulis, Spiros Skiadopoulos, Grigorios Loukides, and Aris Gkoulalas-Divanis. Apriori-based algorithms for k^{m}-anonymizing trajectory data. *Transactions on Data Privacy*, 7(2):165–194, 2014.

[54] Hong Qin, Hao Wang, Xiaochao Wei, Likun Xue, and Lei Wu. Privacy-preserving wildcards pattern matching protocol for iot applications. *IEEE Access*, 7:36094–36102, 2019.

[55] Rajeev Raman. Encoding data structures. In *WALCOM*, pages 1–7, 2015.

[56] David F. Robinson and Les R. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131–147, 1981.

[57] Naruya Saitou and Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(4):406–425, 07 1987.

[58] Jingbo Shang, Jian Peng, and Jiawei Han. Macfp: Maximal approximate consecutive frequent pattern mining under edit distance. In *SDM*, pages 558–566, 2016.

[59] Henry J. Smith, Tamara Dinev, and Heng Xu. Information privacy research: An interdisciplinary review. *MIS Quarterly*, 35(4):989–1015, 2011.

[60] Acar Tamersoy, Grigorios Loukides, Mehmet Ercan Nergiz, Yücel Saygin, and Bradley Malin. Anonymization of longitudinal electronic medical records. *IEEE Transactions on Information Technology in Biomedicine*, 16(3):413–423, 2012.

[61] Manolis Terrovitis and Nikos Mamoulis. Privacy preservation in the publication of trajectories. In *MDM*, pages 65–72, 2008.

[62] Manolis Terrovitis, Nikos Mamoulis, and Panos Kalnis. Local and global recoding methods for anonymizing set-valued data. *The VLDB Journal*, 20(1):83–106, 2011.

[63] Manolis Terrovitis, Giorgos Poulis, Nikos Mamoulis, and Spiros Skiadopoulos. Local suppression and splitting techniques for privacy preserving publication of trajectories. *IEEE Transactions on Knowledge and Data Engineering*, 29(7):1466–1479, 2017.

[64] Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Ambujam Krishnan, and Srinivas Aluru. A greedy alignment-free distance estimator for phylogenetic inference. *BMC Bioinformatics*, 18(8):238:1–238:8, 2017.

[65] George Theodorakopoulos, Reza Shokri, Carmela Troncoso, Jean-Pierre Hubaux, and Jean-Yves Le Boudec. Prolonging the hide-and-seek game: Optimal trajectory privacy for location-based services. In *WPES*, pages 73–82, 2014.

[66] Igor Ulitsky, David Burstein, Tamir Tuller, and Benny Chor. The average common substring approach to phylogenomic reconstruction. *Journal of Computational Biology*, 13(2):336–350, 2006.

[67] Di Wang, Yeye He, Elke Rundensteiner, and Jeffrey F. Naughton. Utility-maximizing event stream suppression. In *SIGMOD*, pages 589–600, 2013.

[68] Jing Wang, Nikos Ntarmos, and Peter Triantafillou. Indexing query graphs to speedup graph query processing. In *EDBT*, pages 41–52, 2016.

[69] Peter Weiner. Linear pattern matching algorithms. In *SWAT*, pages 1–11, 1973.

[70] Virginia Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *STOC*, pages 887–898, 2012.

[71] Xiaokui Xiao, Yufei Tao, and Nick Koudas. Transparent anonymization: Thwarting adversaries who know the algorithm. *ACM Transactions on Database Systems*, 35(2):8:1–8:48, 2010.

[72] Shengzhi Xu, Xiang Cheng, Sen Su, Ke Xiao, and Li Xiong. Differentially private frequent sequence mining. *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2910–2926, 2016.

[73] Mohammed J. Zaki, Wagner Meira Jr, and Wagner Meira. *Data mining and analysis: fundamental concepts and algorithms.* Cambridge University Press, 2014.