Utility-Oriented String Mining

Giulia Bernardini* Huiping Chen† Alessio Conte‡ Roberto Grossi‡
Veronica Guerrini‡ Grigorios Loukides§ Nadia Pisanti‡ Solon P. Pissis¶

Abstract

A string is often provided with numerical scores (*utilities*) which quantify the importance, interest, profit, or risk of the letters occurring at every position of the string. For example, every DNA fragment produced by modern sequencing machines comes with a confidence score per position. Motivated by the abundance of strings with utilities, we introduce Utility-oriented String Mining (USM), a natural generalization of the classic frequent substring mining problem. Given a string S of length n and a threshold $\mathcal{V}$, USM asks for every string R whose utility $U(R)$ is at least $\mathcal{V}$, where U is a function that maps R to a utility score based on the utilities of all letters of every occurrence of R in S. In addition, our work makes the following contributions: (1) We identify a class $\mathbb{U}$ of utility functions for which USM admits an $\mathcal{O}(n^2)$-time algorithm. (2) We prove that no listing algorithm solves the USM problem in subquadratic time for every utility function, or even for every function in $\mathbb{U}$. (3) We propose an $\mathcal{O}(n \log n)$-time algorithm that solves USM for a class of monotone functions from $\mathbb{U}$. (4) We design another $\mathcal{O}(n \log n)$-time algorithm for the same problem that is comparable in runtime but offers drastic space savings in practice when, in addition, a lower bound on the length of the output strings is provided as input. (5) We demonstrate experimentally using publicly available, billion-letter datasets that our algorithms are many times more efficient, in terms of runtime and/or space, compared to an Apriori-like baseline which employs advanced string processing tools.

Keywords: Strings, Patterns, Utility-Oriented Mining

1 Introduction

A string (sequence of letters over an alphabet) is a fundamental data type that plays a key role in many application domains. This is because a string can, for example, model: (1) genomic information of an organism, with each letter representing a nucleotide in DNA [1]; (2) monitoring information in a sensor network, with each letter representing a sensor [2]; (3) advertising (or purchase) information in e-commerce, with each letter representing an advertisement shown [3] (or a product sold [4]) to a user. Mining patterns from a string is thus greatly useful in various application domains. In bioinformatics, it can improve clinical diagnostics [1]; in sensor networks, it can detect whether the sensors are performing as expected [2]; and in e-commerce it can improve business decision making [4].

In all these application domains, a string comes together with numerical scores (*utilities*) that quantify the importance, interest, profit, or risk of the letters occurring at every position of the string [5, 4, 6, 7]. Specifically, in DNA sequencing data, each nucleotide is automatically assigned a confidence score, which represents the probability of this nucleotide being correctly read by the sequencer and helps identifying sequencing errors [8]. In networks, each sensor is often automatically assigned a Received Signal Strength Index (RSSI), i.e., a signal strength value which helps assessing network link quality [9]. In e-commerce, each advertisement is often associated with a Click-Through Rate (CTR), i.e., an estimate of the probability a user clicks on the advertisement, which helps advertisement pricing and ranking [10]. Also, in e-commerce, each product is often associated with a profit made by a sale of the product [5].

Yet, despite significant research in pattern mining from strings with utilities (see [5] for a survey), the following general problem, coined *Utility-oriented String Mining* (USM), has not been studied: Given an input string $S = S[0\,..\,n-1]$, a function $w : [0, n) \to \mathbb{R}$ that assigns to each position $i \in [0, n)$ of S a real number (*utility*), and a utility threshold $\mathcal{V}$, find every string R whose *global utility* $U(R)$ is at least $\mathcal{V}$. The global utility $U(R)$ aggregates the *local utility* of all occurrences of R in S, and the local utility of an occurrence of R in S aggregates the utilities of the letters in the occurrence. Every string R satisfying $U(R) \geq \mathcal{V}$ is termed *useful*. Since every such R is necessarily a substring of S, we will refer to it as a *useful substring* of S.

Example 1. *Consider the string S below and the utilities of its positions assigned by w. Consider also the*

*University of Trieste, giulia.bernardini@units.it
†University of Birmingham, h.chen.13@bham.ac.uk
‡University of Pisa, {alessio.conte, roberto.grossi, veronica.guerrini, nadia.pisanti}@unipi.it
§King's College London, gloukides@acm.org
¶Centrum Wiskunde Informatica, solon.pissis@cwi.nl

following global utility function [11]: $U(R)$ sums up the local utilities of all occurrences of R in S, where the local utility of an occurrence of R is the product of the utilities of its letters. Let $\mathcal{V} = 1.4$. Substring $R =$ `TACCCC` occurs in S at positions 1 and 12, and it is useful as $U(R) = (1\cdot1\cdot1\cdot0.7\cdot1\cdot1)+(1\cdot1\cdot1\cdot0.9\cdot1\cdot1) = 1.6 \geq \mathcal{V}$. Thus, R is in the output of USM *with $\mathcal{V} = 1.4$.*

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
S	A	T	A	C	C	C	C	G	A	T	A	A	T	A	C	C	C	C	C	G
w	.9	1	1	1	.7	1	1	.6	.5	.5	.5	.8	1	1	1	.9	1	1	.9	1

USM is a natural generalization of the well-known frequent substring mining problem [12, 13, 14], which asks for all substrings of an input string S that occur frequently enough in S. The latter problem is obtained from USM by considering utility 1 for each position of S; a local utility of an occurrence of a substring in S given by the product of utilities of the letters in the occurrence; a global utility given by the sum of local utilities of all occurrences of the substring in S (i.e., the global utility sums 1 for each occurrence); and setting $\mathcal{V}$ to the minimum frequency threshold. Due to its generality, USM captures various application requirements. For example, it can identify frequent DNA patterns that have high confidence scores, sensors with connectivity issues, or advertisements that are effective, if displayed one after another.

USM poses a *fundamental computational challenge*: identify when and how it can be solved efficiently, i.e., in time quadratic or subquadratic in the length $n = |S|$. This is necessary, as n is usually large. It is impractical to use a naive $\mathcal{O}(n^3)$-time method which considers all $\mathcal{O}(n^2)$ substrings of S and computes the global utility of each substring in $\mathcal{O}(n)$ time, so as to identify the *complete set Γ of useful substrings* of S. Furthermore, existing mining methods for strings with utilities (e.g., [4, 6, 7]) are not applicable to USM, as they mine high-utility *subsequences* from a collection of strings and use utility functions that are not appropriate for USM.

Contributions. Our work introduces and studies the USM problem, and it makes the following contributions:

1. We identify a special class $\mathbb{U}$ of global utility functions that admit a simple $\mathcal{O}(n^2)$-time algorithm for USM. Every function U in this class aggregates the values of a local utility function with a *sliding window* property: Let $S[i\,..\,j]$ be a fragment of S, $S[i\,..\,i']$ be its prefix and $S[i'+1\,..\,j]$ be its suffix, for $i' \in [i, j)$. The local utility of any of these three fragments can be computed from the local utilities of the other two in $\mathcal{O}(1)$ time. For instance, the local utility of $S[0\,..\,2] =$ `ATA` in Example 1 can be computed in $\mathcal{O}(1)$ time from the local utilities of $S[0\,..\,1] =$ `AT` and $S[2\,..\,2] =$ `A`.

2. We prove that no listing algorithm solves the USM problem for every U in $\mathcal{O}(n^{2-\epsilon})$ time, for any $\epsilon > 0$, even if U belongs to class $\mathbb{U}$. In particular, we show that $|\Gamma| = \Theta(n^2)$ in the worst case. To circumvent this difficulty, we explore several different avenues as described by the following two contributions.

3. We identify a subclass $\mathbb{U}_M$ of $\mathbb{U}$ for which there exists a compact representation of Γ that takes $\mathcal{O}(n)$ space, even though $|\Gamma|$ can still be $\Theta(n^2)$. This representation consists of all *maximal* useful substrings (i.e., we discard any $X \in \Gamma$ that is a substring of another $Y \in \Gamma$). Armed with this representation, we design an $\mathcal{O}(n \log n)$-time and $\Theta(n)$-space algorithm for USM, for any function $U \in \mathbb{U}_M$. Our algorithm, called MIA, constructs such a representation by employing string indexes and several non-trivial combinatorial insights.

Example 2. *Consider the string S, the functions w and U, as well as the threshold $\mathcal{V}$ of Example 1.* MIA *outputs* `ATAC`*,* `CCG`*, and* `TACCCC`*, as these are the only maximal useful substrings. Indeed, we can easily and uniquely reconstruct $\Gamma = \{$`A`, `T`, `C`, `G`, `AT`, `TA`, `AC`, `CC`, `CG`, `ATA`, $\ldots$, `ATAC`, `CCG`, `TACCCC`$\}$ by listing* `ATAC`*,* `CCG`*,* `TACCCC`*, and all their substrings.*

4. The MIA algorithm uses an index of size $\Theta(n)$ over the input string S, which can be impractical when S is very long. To circumvent this, we consider a parameterized version of USM which, for any $U \in \mathbb{U}_M$ and any integer $L > 0$, asks for the maximal useful substrings in Γ *of length at least L*. We design MSA, an $\mathcal{O}(n \log n)$-time algorithm for this version, which is based on hashing and sketching. MSA is comparable to MIA in terms of runtime, but it achieves drastic space improvements in practice, via utilizing external memory, even though its space is still $\Theta(n)$ in the worst case.

5. We present extensive experiments using 4 publicly available, large-scale datasets from different domains demonstrating that our algorithms are much more efficient in terms of runtime and/or space compared to an Apriori-like [15] baseline which utilizes advanced string processing tools [16, 17]. For example, MSA needed only 4 GBs of RAM to process a DNA dataset of size 22.5 GBs, while the baseline needed more than 232 GBs of RAM! At the same time, MSA was 2.5 times faster than this baseline and comparable to MIA.

2 Showcasing Our Methods

Existing sequencers do not sequence a whole genome at once but produce a collection of sequenced genome fragments (*reads*), accompanied by confidence scores. Thus, many methods perform DNA analyses, including frequent substring mining, after processing the reads using costly operations, e.g., genome assembly. Current research aims to bypass such operations by designing methods that operate directly on the reads [18].

This is challenging due to the presence of sequencing errors, namely, of nucleotides with low confidence scores [8]. The probability of a substring occurrence can be modeled as the product of the confidence scores of its letters, and the sum of these products over all the occurrences of the substring in the reads gives the *expected frequency* [11] of the substring. This function is the same as U in Example 1.

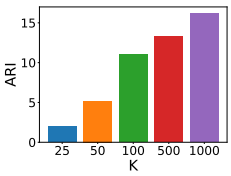


Figure 1: ARI over $\mathcal{V} \in \{\frac{c}{4}, \frac{c}{2}, \frac{3}{4}c, c, \frac{3}{2}c, 2c\}$, where $c = 490$ is the coverage, and varying K.

By employing the expected frequency as their global utility function, our algorithms mine higher-quality substrings compared to frequent substring mining algorithms, which, by design, do not consider the confidence scores. We show this using a real reads dataset, called ECOLI (see Table 1). In particular, we show that MIA and MSA mine maximal substrings which are more frequent in the standard ground truth (reference) dataset for ECOLI [19] than those mined by frequent substring mining algorithms.

For this: (1) We mine the set $O_{K,\mathcal{V}}$ of the K maximal substrings with the largest *expected frequency*, for a given K and utility threshold $\mathcal{V}$ that is a multiple of *coverage* (i.e., the average number of reads that overlap a position in the ground truth dataset). (2) We mine the set $O_{K,F}$ of the K maximal substrings with the largest *frequency*, for the same K as in step 1 and frequency threshold $F = \mathcal{V}$. (3) We compare these two sets with respect to their *cumulative frequency* $\mathcal{F}(O_{K,\mathcal{V}})$ and $\mathcal{F}(O_{K,F})$ in the ground truth dataset, by computing $\frac{\mathcal{F}(O_{K,\mathcal{V}})-\mathcal{F}(O_{K,F})+1}{\mathcal{F}(O_{K,F})+1}$. This measure quantifies the relative improvement of $O_{K,\mathcal{V}}$ compared to $O_{K,F}$.

Fig. 1 shows the average value of this measure, denoted by ARI (for Average Relative Improvement), over different values of $\mathcal{V}$ and $F = \mathcal{V}$, for varying K. Our algorithms were at least 2 and up to 16 times more effective than frequent substring mining algorithms in this experiment, which highlights their benefit.

3 Preliminaries, Problems, and Results

Strings. An *alphabet* Σ is a finite nonempty set of elements called *letters*. A *string* $S = S[0\,..\,n-1]$ of *length* $|S| = n$ is a sequence of n letters from Σ, where $S[i]$ denotes the i-th letter of the sequence. We refer to each $i \in [0, n)$ as a *position* of S. We consider throughout that Σ is an integer alphabet.

A substring R of S may occur multiple times in S. The set of its occurrences in S is denoted by $\mathsf{occ}_S(R)$; we may omit the subscript S when it is clear from the context. An occurrence of R in S starting at position i is referred to as a *fragment* of S and is denoted by $\mathsf{frag}_S(i, |R|) = S[i\,..\,i + |R| - 1]$. Thus, different fragments may correspond to different occurrences of the same substring.

A *prefix* of S is a substring of the form $S[0\,..\,j]$, and a *suffix* of S is a substring of the form $S[i\,..\,n-1]$ (thus, any fragment of S is a prefix of some suffix of S).

Utility Definitions. Let S be a string of length n and let $w : [0, n) \to \mathbb{R}$ be a function that assigns to each position $i \in [0, n)$ of S a real number $w[i]$, referred to as the *utility* of $S[i]$. We may refer to the pair (S, w) as a *weighted string*. For any fragment $\mathsf{frag}_S(i, |R|)$, a *local utility function* $u(i, |R|)$ aggregates the utilities of all letters of the fragment (i.e., $w[k]$, for each $k \in [i, i + |R| - 1]$). For any substring R of S, a *global utility function* $U(R)$ aggregates the value of the local utility of all the occurrences of the substring in S.

We define a class $\mathbb{U}$ of global utility functions, such that for every $U \in \mathbb{U}$: (1) U is *linear-time* computable (e.g., sum, min, max, or avg); *and* (2) the local utility function of U has the *sliding window* property (e.g., sum): for any three fragments of S, $S[i\,..\,j]$, its prefix $S[i\,..\,i']$, $i \le i'$, and its suffix $S[i'+1\,..\,j]$, $i'+1 \le j$, the local utility of any of these three fragments can be obtained from the local utilities of the other two in $\mathcal{O}(1)$ time. Specifically, given the local utilities of $S[i\,..\,j]$, $S[i]$, and $S[j+1]$, the local utility of $S[i+1\,..\,j+1]$ can be computed in $\mathcal{O}(1)$ time yielding *directly* the following:

LEMMA 3.1. *Given a local utility function u with the sliding window property, and any fixed length k, the value of u for all fragments of length k of S can be computed in $\mathcal{O}(n)$ time.*

We also consider monotone functions in $\mathbb{U}$ that *decrease* (or stay the same) as a fragment gets longer and *increase* (or stay the same) as local utilities of different fragments $u(i_1, |R|), u(i_2, |R|), \ldots$ of R are aggregated. We refer to these functions as $\mathbb{U}_M$ (for monotone). Our example in this paper is the "sum of products" function: $U(R) = \sum_{i \in \mathsf{occ}_S(R)} u(i, |R|)$. Its local utility function is $u(i, |R|) = \prod_{k=i}^{i+|R|-1} w[k]$, for $i \in [0, n-|R|]$, and $u(i, |R|) = 0$ otherwise, and $w : [0, n) \to [0, 1]$. U is used in [11] and is an adaptation of the widely-used *expected support* [5] in our setting.

EXAMPLE 3. *Consider S, w, U, and the local utility u as in Example 1. The fragment $\mathsf{frag}(9, 2) = \mathtt{TA}$ has local utility $u(9, 2) = w[9] \cdot w[10] = 0.25$. The substring $\mathtt{TA}$ has global utility $U(\mathtt{TA}) = u(1, 2) + u(9, 2) + u(12, 2) = 2.25$, as $\mathsf{occ}_S(\mathtt{TA}) = \{1, 9, 12\}$.*

A substring R of S is *useful* w.r.t. a global utility function U and a given utility threshold $\mathcal{V}$, if and only if $U(R) \geq \mathcal{V}$. A useful substring R of S is *maximal*, if it is *not* a substring of a useful substring $R' \neq R$ of S.

Problems and Results. We prove that, for a weighted string (S, w) of length n, solving USM by listing all its useful substrings requires in general $\Omega(n^2)$ time, even if U belongs to $\mathbb{U}$, and that it can be performed in $\Theta(n^2)$ time, for any $U \in \mathbb{U}$.

THEOREM 3.1. *A weighted string (S, w) of length n can have $\Theta(n^2)$ useful substrings for a global utility function $U \in \mathbb{U}$ and a threshold $\mathcal{V}$. Thus, no algorithm lists all useful substrings of S in $\mathcal{O}(n^{2-\epsilon})$-time, for any $\epsilon > 0$.*

Proof. Let $S = \mathtt{A}(\mathtt{ATA})^\ell\mathtt{C}(\mathtt{ATA})^\ell\mathtt{A}$, where $(\mathtt{ATA})^\ell$ concatenates $\mathtt{ATA}$ ℓ times, $n = |S| = 6\ell + 3 \to \infty$, and w be 1 (resp., -2 and 0) for each position containing $\mathtt{A}$ (resp., $\mathtt{T}$ and $\mathtt{C}$). Let also $U \in \mathbb{U}$ be a function that sums up the local utilities which in turn sum the utilities w, and let $\mathcal{V} = 1.5$. For any $i, j \in [1, \ell]$, the useful substrings of type $R = \mathtt{A}(\mathtt{ATA})^i\mathtt{C}(\mathtt{ATA})^j\mathtt{A}$ have $U(R) \geq 2 > \mathcal{V}$ and there are $\ell^2 = \Theta(n^2)$ of them, as $\ell = \Theta(n)$. These substrings contain among others the non-useful substrings of type $(\mathtt{ATA})^i\mathtt{C}(\mathtt{ATA})^j$ and those starting with $\mathtt{T}$ that are $\Theta(n^2)$. Thus, there are $\Theta(n^2)$ non-useful substrings contained in the $\ell^2 = \Theta(n^2)$ useful ones and the latter cannot be listed in $\mathcal{O}(n^{2-\epsilon})$ time for any $\epsilon > 0$. ∎

THEOREM 3.2. *There is a $\Theta(n^2)$-time algorithm that, for any weighted string (S, w) of length n, global utility function $U \in \mathbb{U}$, and threshold $\mathcal{V}$, lists all useful substrings of S.*[1]

Theorem 3.2 applies Lemma 3.1 for any feasible k. To improve over Theorem 3.2, we consider a global utility function $U \in \mathbb{U}_M \subset \mathbb{U}$, so that there is a compact representation of the useful strings, comprised of *inclusion-maximal* useful substrings. Based on this representation, we define our main *string mining* problem, which is a special case of USM.

PROBLEM 1. MAXIMAL USEFUL STRING MINING (MUSM) *Given a weighted string (S, w) of length n, a global utility function $U \in \mathbb{U}_M$, and a utility threshold $\mathcal{V}$, find all* maximal useful *substrings of S.*

For any such U, MIA underlies the following result.

THEOREM 3.3. *There exists an $\mathcal{O}(n \log k_{\max})$-time algorithm that solves* MUSM *using $\Theta(n)$ extra space, where $k_{\max} \leq n$ is the length of the longest useful substring of S.*

MSA takes $\mathcal{O}(n \log n)$ time and $\Theta(n)$ extra space in the worst case but with huge space savings in practice.

[1] *All missing proofs are deferred to the full version.*

4 MIA: An Index-Based Algorithm for MUSM

MIA (for MUSM *Index-based Algorithm*) is an efficient algorithm for MUSM, comprised of three phases:

- In Phase I, MIA finds, for each position $i \in [0, n)$ of the input string S, the largest power of 2 that is equal to the length of a useful substring of S occurring at i. Let this power be 2^{p_i}. The fragment at position i of length 2^{p_i} will be further considered in Phase II for potential extension.
- In Phase II, MIA considers, for each position $i \in [0, n)$ of S, a potential extension of length $2^{p_i - 1}$ to the current fragment occurring at i (i.e., appending $2^{p_i - 1}$ letters to it). If this extension leads to a useful substring, it is applied; otherwise, it is discarded. Next, MIA repeats the process for further potential extensions of length equal to the next smaller power of 2, stopping after 2^0.
- In Phase III, MIA finds and outputs all maximal useful substrings of S. This is performed by discarding every useful substring that is output by Phase II but is *not* maximal, as it cannot be part of the output of MUSM.

Phases I and II of MIA each consist of at most $\log n$ iterations; *in each iteration d,* MIA *processes substrings of length 2^d*. Two remarks are important for efficiency. First, without Phases I and II, Phase III would be applied to all substrings of S. This would solve MUSM but in $\Omega(n^2)$ time. Second, independently processing each position i of S would need $\Omega(n^2)$ time. To achieve $\mathcal{O}(n \log n)$ time, we introduce two key ideas: (1) We factorize the length of the candidate useful substrings into powers of 2 and use the sliding window property. This allows us to compute, in $\mathcal{O}(n)$ time, the local utility of all fragments of the same length (that is a power of 2) *at the same time*. (2) We use string indexing data structures, in order to evaluate the global utility of all the substrings considered in an iteration of Phase I or II, and to test maximality in Phase III, in $\mathcal{O}(n)$ time. Since there are $\log n$ powers of 2, the time complexity of MIA is $\mathcal{O}(n \log n)$.

Example 4 explains how the maximal useful substring $\mathtt{TACCCC}$ in Example 2 is produced by MIA.

EXAMPLE 4. *Recall that $\mathcal{V} = 1.4$. In Phase I,* MIA *considers position $i = 1$ and finds that $2^{p_1} = 2^2$ is the largest power of* 2 *that is equal to the length of the useful string* $\mathtt{TACC}$ *occurring at i. This is because the immediately larger power, 2^3, corresponds to* $\mathtt{TACCCCGA}$*, which is not useful. In Phase II,* MIA *considers a potential extension of length $2^{p_1 - 1} = 2^1$ to the fragment* $\mathtt{TACC}$ *(i.e., appending* $\mathtt{CC}$ *to* $\mathtt{TACC}$*). As* $\mathtt{TACCCC}$ *is useful (its utility is $0.7 + 0.9 = 1.6 \geq \mathcal{V}$),* $\mathtt{TACC}$ *is extended to* $\mathtt{TACCCC}$*. Then,* MIA *considers 2^0, the next smaller*

power of 2*, which corresponds to* G*. However,* TACCCCG *is* not *useful and thus* TACCCC *is* not *extended. In Phase III,* MIA *outputs* TACCCC*, as it is maximal useful, and discards* ACCC*, which occurs at* $i = 2$*, along with all other non-maximal substrings.*

We now discuss MIA in detail; see also Algorithm 1.

Utility Functions. MIA works for any global utility function $U \in \mathbb{U}_M$, but we describe it using our "sum of products" function (see Section 3). MIA exploits the fact that the local utility u of this function has the sliding window property: For any non-negative integer $d \leq \log n$, MIA computes an array of local utilities lu_d such that $lu_d[i] = u(i, 2^d)$, for all $i \in [0, n)$, in $\mathcal{O}(n)$ time by Lemma 3.1.

Utility Check. Each iteration of Phase I and II relies on the computation of the following *utility check* (lines 9 and 22): For a candidate substring R and a subset $I_R \subseteq \mathsf{occ}_S(R)$ of its occurrences, it verifies whether the fragments corresponding to I_R satisfy $\sum_{i \in I_R} u(i, |R|) \geq \mathcal{V}$. Note that when $I_R = \mathsf{occ}_S(R)$, this condition is actually a check on the *global* utility of R (i.e., on $U(R) \geq \mathcal{V}$). We will later prove (Lemma 4.4) that: (1) this is always the case when R is maximal; and (2) checking for $I_R \subseteq \mathsf{occ}_S(R)$ is necessary for correctness.

Lemma 4.1 states that the utility check can be performed efficiently, using two string indexing data structures: suffix array (SA) [16] and LCP array (LCP) [17].

LEMMA 4.1. *At each iteration of* MIA*, the utility check for all fragments can be computed in* $\mathcal{O}(n)$ *time using the* SA *and* LCP *array of* S*, which occupy* $\Theta(n)$ *space.*

To keep track of the outcome of the utility check at each iteration, MIA uses arrays *len* and *prod*; each is of size n and initialized with zeros. For each position $i \in [0, n)$ of S, $len[i]$ is the length of the *longest* fragment $\mathsf{frag}_S(i, len[i])$ that *passed* the utility check at some previous iteration, and $prod[i] = u(i, len[i])$ is the local utility of this fragment. Also, in Phase III, *len* will identify $\mathcal{O}(n)$ substrings as candidates for being maximal (compared to the $\Theta(n^2)$ possible candidates).

Phase I (lines 1–13). MIA runs at most $\log n + 1$ doubling iterations for increasing values of d between 0 and $\log n$, and performs the utility check on fragments of length 2^d (those in $Cand_I$ in line 4). Only the positions that pass the check (stored in set I') are then involved in the next iteration for fragments of length 2^{d+1}; for them, *len* and *prod* are overwritten (lines 11–12). Phase I can be implemented efficiently using string indexes:

LEMMA 4.2. *Phase I performs* $\Theta(\log k_{\max})$ *iterations, where* $k_{\max}$ *is the length of the longest useful substring of* S*. Each iteration takes* $\mathcal{O}(n)$ *time. Thus, Phase I requires* $\mathcal{O}(n \log k_{\max})$ *time in total and* $\Theta(n)$ *space.*

Algorithm 1 MIA $(S, w, n, U, u, \mathcal{V})$

```
    /* Phase I */
 1: len[0..n-1] ← 0s; prod[0..n-1] ← 0s;
 2: d ← 0; I ← {0, 1, ..., n-1};
 3: while d ≤ log n do
 4:     Cand_I ← {frag(i, 2^d) | i ∈ I};          ▷ set of distinct strings
 5:     for each i ∈ [0, n-2^d] do lu_d[i] ← u(i, 2^d); ▷ Lemma 3.1
 6:     I' ← ∅;
 7:     for each R ∈ Cand_I do
 8:         I_R ← {i ∈ I | R = frag(i, 2^d)};
 9:         if Σ_{i∈I_R} lu_d[i] ≥ V then I' ← I' ∪ I_R;
10:     if I' = ∅ then break;
11:     for each i ∈ I' do
12:         len[i] ← 2^d; prod[i] ← lu_d[i];
13:     d ← d + 1; I ← I';
    /* Phase II */
14: d ← d - 2;                  ▷ d and d-1 surely fail to extend
15: while d ≥ 0 do
16:     J ← {0 ≤ i ≤ n - 2^d - len[i] | len[i] > 2^d};
17:     Cand_J ← {frag(i, len[i] + 2^d) | i ∈ J};
18:     for each i ∈ [0, n-2^d] do lu_d[i] ← u(i, 2^d); ▷ Lemma 3.1
19:     J' ← ∅;
20:     for each R ∈ Cand_J do
21:         J_R ← {i ∈ J | R = frag(i, len[i] + 2^d)};
22:         if Σ_{i∈J_R} (prod[i] · lu_d[i + len[i]]) ≥ V then J' ← J' ∪ J_R;
23:     for each i ∈ J' do
24:         len[i] ← len[i] + 2^d;
25:         prod[i] ← prod[i] · lu_d[i + len[i]];
26:     d ← d - 1;
    /* Phase III */
27: K ← {0 ≤ i < n | len[i] > 0};
28: Cand_K ← {frag(i, len[i]) | i ∈ K};
29: sol ← ∅;
30: for each R ∈ Cand_K do   ▷ keep maximal useful substrings
31:     if R is not substring of any R' ∈ Cand_K \ {R} then
32:         sol ← sol ∪ {R};
33: return sol;
```

Phase II (lines 14–26). MIA applies a binary search for $d = \widehat{d} - 2, \widehat{d} - 3, \ldots, 1, 0$, similarly to Phase I, where $\widehat{d}$ is the value of d at the end of Phase I. At iteration d, MIA processes the set J of positions i whose current length is $len[i] > 2^d$ (line 16). For each position $i \in J$, it tries to extend $\mathsf{frag}_S(i, len[i])$ by 2^d letters, which occur at offset $len[i]$ from i (line 17), and thus have local utility stored in $lu_d[i + len[i]]$ (after line 18). Armed with this information, in line 22, MIA performs the utility check for J and stores the positions that pass the check in J'. The entries of *len* and *prod* are then updated only for J' accordingly (lines 24–25). Phase II can be implemented efficiently using string indexes:

LEMMA 4.3. *Phase II performs* $\Theta(\log k_{\max})$ *iterations, where* $k_{\max}$ *is the length of the longest useful substring of* S*. Each iteration takes* $\mathcal{O}(n)$ *time. Thus, Phase II requires* $\mathcal{O}(n \log k_{\max})$ *time in total and* $\Theta(n)$ *space.*

Phase III (lines 27–33) MIA considers the fragments

with positive values in *len* and collects their corresponding strings in set $Cand_K$ (lines 27–28). Note that $|Cand_K| \leq n$, as $|K| \leq n$ and the distinct strings cannot be more than the fragments they correspond to. Thus, Phase III is applied to $\mathcal{O}(n)$ substrings of S.

MIA hinges on the crucial property that the maximal useful strings in S are *all* contained in $Cand_K$ (see Lemma 4.4), so that a test of substring inclusion identifies the maximal ones (lines 30-32). This is performed efficiently using string indexes; see Lemma 4.5.

LEMMA 4.4. *If a maximal useful substring R of S occurs at position i, then $len[i] > 0$ and* MIA *finds that $U(R) \geq \mathcal{V}$.*

LEMMA 4.5. *Phase III requires $\mathcal{O}(n)$ time and $\Theta(n)$ space and it correctly returns the set of maximal useful substrings of S.*

Lemmas 4.1 to 4.5 yield *directly* Theorem 3.3.

5 MSA: A Scan-Based Algorithm for MUSM

MIA uses $\Theta(n)$ space for a string of length n with significant constant factors, due to the use of the SA and LCP array. Thus, it requires too much space when S is very long. To address this, we: (1) consider a parameterized version of MUSM asking for all maximal useful substrings of an input weighted string (S, w) that *have length at least* L, for a given integer $L > 0$; and (2) develop MSA (for MUSM *Scan-based Algorithm*), which solves this parameterized version, in far less space than MIA in practice, while being equally efficient.

MSA is a data-scan based algorithm that is similar to MIA but implements Phases I and II differently: (1) It uses a hash dictionary to store some distinct useful strings in main memory with random access to it. (2) It maintains all other information in external memory, using few arrays of size n each. These arrays are not resident in main memory but are loaded in constant-size buffers through sequential access. (3) It operates in a streaming fashion for these arrays, as only small parts of them are needed in the main memory at any time.

Data Structures. MSA relies on hashing strings with the Karp-Rabin method [20] (a.k.a. rolling hash). We denote by $KR(X)$ the integer obtained by hashing string X. Without loss of generality, we assume that $KR(X) = KR(Y)$ if and only if $X = Y$, as the Karp-Rabin method can be tuned to have no collision *with high probability*.

MSA maintains arrays *len* and *prod* to keep track of the utility check, described in Section 4. Along with them, it keeps two other arrays *lim* and *hval* of size n each, such that $hval[i] = KR(\mathsf{frag}(i, len[i]))$ stores the Karp-Rabin hash value of the fragment $\mathsf{frag}(i, len[i])$, and $lim[i] > len[i]$ is a strict upper limit on any value that $len[i]$ can get. All four arrays are read or written in blocks of B entries, where $B \ll n$ is a user-defined parameter that specifies the block size.

MSA uses a dictionary D which stores pairs (h, v), where h is a Karp-Rabin hash value and v is a utility value (i.e., $\mathsf{D}[h] = v$), initially set to 0. D implements the utility check $\sum_{i \in I_R} u(i, |R|) \geq \mathcal{V}$ of Section 4 for a substring R and a set I_R of its occurrences (positions): Given $i \in I_R$, MSA increments $\mathsf{D}[hval[i]]$ by $u(i, |R|)$, as we guarantee that $hval[i] = KR(R)$. When all positions in I_R are examined, $\mathsf{D}[KR(R)] = \sum_{i \in I_R} u(i, |R|)$ holds. Thus, the utility check is $\mathsf{D}[KR(R)] \geq \mathcal{V}$.

Phase I. Instead of performing $\mathcal{O}(\log n)$ iterations, MSA examines only fragments of L letters in S using $\mathcal{O}(1)$ iterations, and it applies the utility check on these substrings using a dictionary D it constructs. For each position i, MSA sets $len[i] = L$ if and only if a useful string of length L occurs at i. If so, MSA sets $lim[i] = i' + L$, where $i' \geq i$ is the largest position such that $len[i] = len[i+1] = \cdots = len[i'] = L$. This encodes a run of consecutive occurrences of useful strings of length L. These runs determine the minimum value d' such that $2^{d'} > \max_{i \in [0,n)}\{lim[i] - len[i]\}$.

Phase II. MSA applies iterations $d = d' - 2, d' - 3, \ldots, 1, 0$. In each iteration d, MSA computes the sum of the local utilities of each fragment $\mathsf{frag}(i, len[i] + 2^d)$ at position $i \in [0, n)$ using D. After that, D is ready for the utility check. Thus, MSA performs the utility check on each fragment that was considered, using D: If the check succeeds, $len[i] = len[i] + 2^d$ (as in MIA). Otherwise, $lim[i] = len[i] + 2^d$ (new in MSA). The arrays *prod* and *hval* are consequently updated in $\mathcal{O}(1)$ time per entry as discussed. MSA uses array *lim* as a key step to eliminate positions that have no chance to be occurrences of maximal useful strings; this is crucial to reduce the size of D. Namely, if $[i, i + lim[i] - 1] \subseteq [i', i' + len[i']]$ holds for $i' \neq i$, then i can be safely removed from the set of positions, and the string of its fragment can be deleted from D.

Phase III. Performed as in MIA.

LEMMA 5.1. *For any (S, w) of length n,* MSA *takes $\mathcal{O}(n \log n)$ time, as each of the $\mathcal{O}(\log n)$ iterations requires $\mathcal{O}(n)$ time (and a linear number of I/Os). As for space, let $\mathsf{D}_{\max}$ be the largest size of dictionary D at any moment. For a given block size $B \ll n$, the total additional space (not accounting for S and w) is $\mathcal{O}(B + \mathsf{D}_{\max})$ in main memory, plus $\Theta(n)$ space in external memory.*

Reducing the Dictionary Size via Sketching. As L increases, the substrings of length L typically increase in number but those leading to useful strings may be few. We identify such candidates using small space by

a pre-filtering step in Phase I. We use a Count-Min Sketch [21], denoted by CMS. CMS supports one-side error queries [21] and is configured to trade off $u(i, |R|)$'s approximation with space usage: If $\mathsf{CMS}[KR(R)] < \mathcal{V}$, surely R is not useful and MSA discards R. Otherwise, there is an error with bounded probability, so MSA passes R to the dictionary. Thus, $\mathsf{D}_{\max}$ in Lemma 5.1 gets smaller. CMS works for a global utility function aggregating local utilities by sum; pre-filtering for other functions in $\mathbb{U}_M$ is left for future work.

6 Related Work

Many works focus on mining patterns either from a collection of strings (e.g., [13, 22]) or from a single long string (e.g., [12, 23]). Unlike these works, we consider strings with associated utilities.

As such, our work falls into utility-oriented pattern mining, an emerging direction in data mining that considers the utility of data elements [5]. There are algorithms for mining utility-oriented itemsets, association rules, and episodes (see [5] for a survey). These algorithms are not applicable to strings. There are also algorithms applied to a collection of short strings comprised of letters or itemsets [4, 6, 7]. However, these algorithms mine subsequences, which are not necessarily comprised of consecutive elements, as in our case. Since the set of these subsequences can be exponentially large, these algorithms are inapplicable to a realistically long string. Also, [4, 6, 7] use different utility functions than ours.

At each position of an *uncertain* string [24, 25], *every* letter of the alphabet is associated with a probability, and the sum of all probabilities at any position must be equal to one. Thus, a weighted string (S, w) drawn from an alphabet Σ can be converted into an uncertain string when each $w[i]$ is in $[0, 1]$: At each position i of the latter string, the letter $S[i]$ is associated with probability $w[i]$ and a letter $\# \notin \Sigma$ representing all other letters is associated with probability $1 - w[i]$. The mining of uncertain strings has been studied in [25]. However, the mining method in [25] cannot solve MUSM, as it employs a fundamentally different local utility function.

7 Experimental Evaluation

Datasets. We used 4 large-scale datasets; see Table 1. The strings IoT and ECOLI in Table 1 are associated to real utilities. In IoT, the utilities are RSSIs normalized in $[0, 1]$, and in ECOLI, confidence scores [8] in $[0, 1]$; see Section 1. In XML and CHR, there are no real utilities. Thus, we selected each utility $w[i]$, $i \in [0, n)$, uniformly at random from $\{0.7, 0.75, \ldots, 1\}$.

Setup. We compared MIA and MSA to an Apriori-like [15], $\mathcal{O}(n\, k_{\max})$-time baseline, called MBA (for MUSM BAseline). MBA considers substrings of increasing length in each iteration, stops when no substring is useful, and discards non-maximal useful substrings as in MIA. It uses SA and LCP to compute the global utility of all substrings of length k in $\Theta(n)$ time. We did not compare MIA and MSA to utility-oriented subsequence mining algorithms [4, 6, 7], as these algorithms are not alternative to ours; see Section 6. We used the "sum of products" function; see Section 3.

Table 1: Dataset properties and values for $\mathcal{V}$ and L.

Dataset	Length n	Alphabet Size $\lvert\Sigma\rvert$	Dataset Size (GBs)	Available at	$\mathcal{V}$	L
IoT	$1.9 \cdot 10^7$	63	0.171	`https://bit.ly/3X3rpmE`	$[10^1, 10^5]$	$[1, 32]$
XML	$2 \cdot 10^8$	95	1.8	`http://pizzachili.dcc.uchile.cl/texts.html`	$[10^1, 10^5]$	$[1, 32]$
ECOLI	$4.6 \cdot 10^9$	6	41.4	`https://bit.ly/3pcU0d4`	$[10^1, 10^5]$	$[1, 32]$
CHR	$2.5 \cdot 10^9$	4	22.5	`https://github.com/koeppl/phoni`	$[10^1, 10^5]$	$[1, 32]$

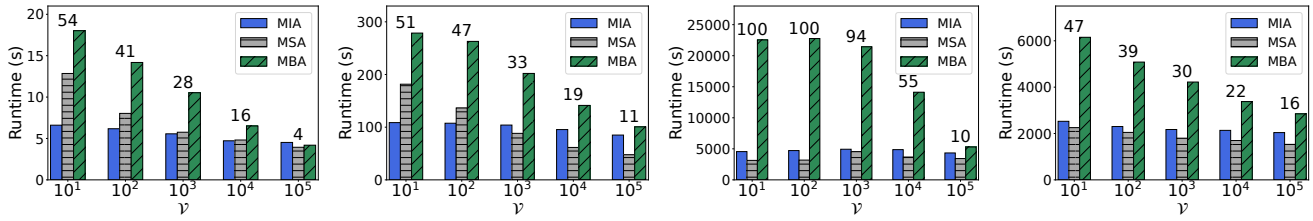
All our experiments ran on an Intel Xeon Gold 5318Y CPU at 2.10GHz with 512GB RAM. We implemented all algorithms in C++. *Our code is available at* `https://github.com/gloukides/usm`.

Results. We report the runtime and peak memory usage of all algorithms, noting that the external memory usage of MSA was exactly $24n$ bytes. We configured MIA and the baseline MBA to solve the same (parameterized) problem as MSA. In MSA, we set $B = 256$KB and the error probability threshold to $\frac{1}{e^{11}}$.

Impact of $\mathcal{V}$. Fig. 2 shows the impact of $\mathcal{V}$ on the runtime of all algorithms. MIA and MSA were *more than* 2.3 *times faster* than MBA on average (over all datasets). Note that $k_{\max}$ (the length of the longest useful substring) decreases as $\mathcal{V}$ increases, which makes MBA much faster, as it takes $\mathcal{O}(n\, k_{\max})$ time. MSA also becomes faster, although it takes $\mathcal{O}(n \log n)$ time, as its dictionary D gets smaller and thus it is accessed more efficiently. MIA becomes only slightly faster, as it takes $\mathcal{O}(n \log k_{\max})$ time.

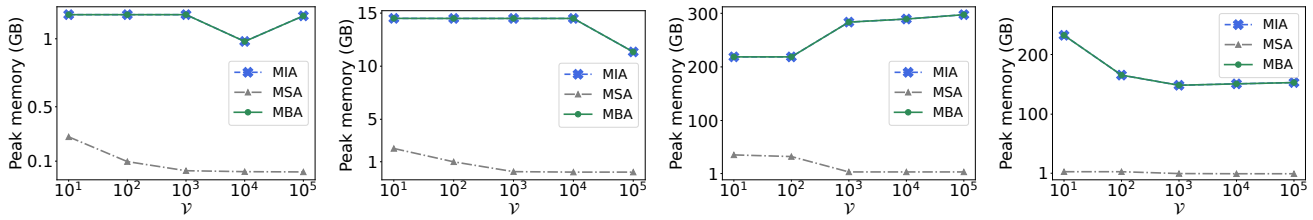
Fig. 3 shows the impact of increasing $\mathcal{V}$ on the peak memory usage of all algorithms. MSA uses *205 times less memory on average* (over all datasets) than MIA and MBA, which use a similar amount of memory, as they employ the same $\Theta(n)$-space indexes. Unlike MSA, the amount of memory that MIA and MBA require is several times larger than the dataset size (e.g., 10.3 times larger in the case of CHR). In addition, MSA needs less memory as $\mathcal{V}$ increases since its dictionary D gets smaller. On the other hand, the indexes of MIA and MBA occupy $\Theta(n)$ space, for any $\mathcal{V}$.

Impact of L. Fig. 4 shows the impact of increasing L on the runtime of all algorithms. MSA was *more than two times faster* than MBA and comparable to (or sometimes slower than) MIA, as the small $\mathcal{V} = 10$ that we used led to a large dictionary for MSA, especially for small L. When $\mathcal{V}$ was larger, MSA was 21% faster than MIA on average; see Figs. 6a and 6b. Note that the runtime of MIA and MBA was not affected by L, as they simply check whether a maximal useful pattern



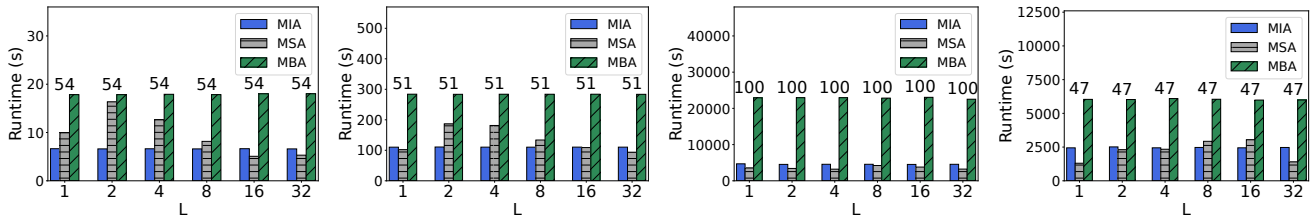
(a) IoT (b) XML (c) ECOLI (d) CHR

Figure 2: Runtime in seconds vs $\mathcal{V}$ for $L = 4$: (a) IoT, (b) XML, (c) ECOLI, and (d) CHR. The number on the top of each group of bars denotes $k_{\max}$ (length of the longest useful substring), which is the same for all algorithms.



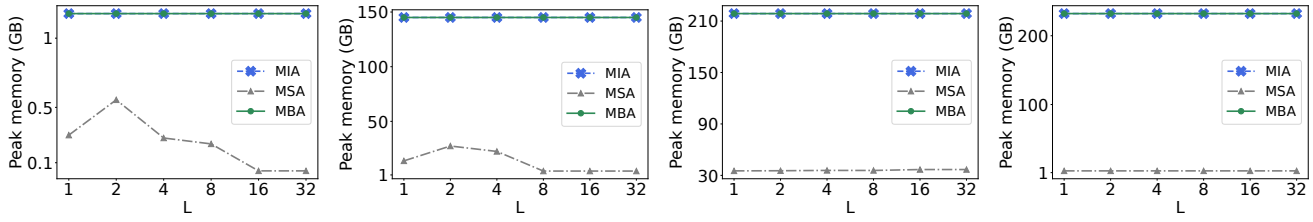
(a) IoT (b) XML (c) ECOLI (d) CHR

Figure 3: Peak memory usage in GBs vs $\mathcal{V}$ for $L = 4$: (a) IoT, (b) XML, (c) ECOLI, and (d) CHR.



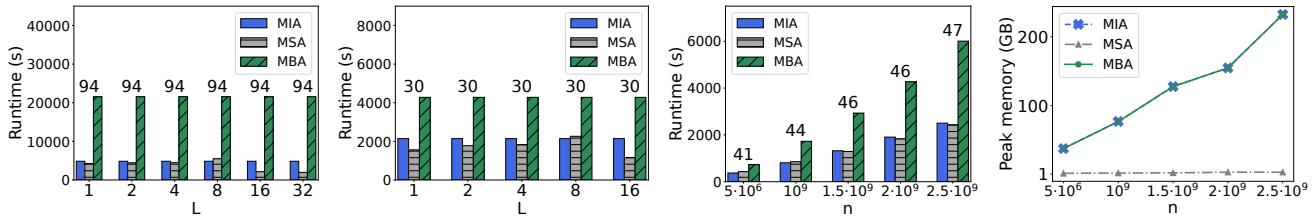
(a) IoT (b) XML (c) ECOLI (d) CHR

Figure 4: Runtime in seconds vs L for $\mathcal{V} = 10$: (a) IoT, (b) XML, (c) ECOLI and (d) CHR. The number on the top of each group of bars denotes $k_{\max}$ (length of the longest useful substring), which is the same for all algorithms.



(a) IoT (b) XML (c) ECOLI (d) CHR

Figure 5: Peak memory usage in GBs vs L for $\mathcal{V} = 10$: (a) IoT, (b) XML, (c) ECOLI, and (d) CHR.



(a) ECOLI (b) CHR (c) Prefixes of CHR (d) Prefixes of CHR

Figure 6: Runtime in seconds vs L for $\mathcal{V} = 1000$: (a) ECOLI and (b) CHR. (c) Runtime in seconds vs n and (d) peak memory usage in GBs vs n for $\mathcal{V} = 10$ and $L = 4$ on prefixes of CHR. The number on the top of each group of bars denotes the $k_{\max}$ (length of the longest useful substring) of all algorithms.

has length at least L, which is very fast.

Fig. 5 shows the impact of increasing L on the peak memory usage of all algorithms. MSA uses *25 times less memory on average* than MIA and MBA. For instance, MSA needed less than 37 GBs of RAM to process ECOLI, whose size is 41.4 GBs, while MIA and MBA needed more than 230 GBs of RAM. In general, MSA needs less memory as L increases, as we see the benefit of CMS on making dictionary D smaller: although there are potentially many substrings of length L, only few of them pass the filtering performed by CMS. On the other hand, MIA and MBA need a similar amount of memory, as the space occupied by their $\Theta(n)$-space indexes is independent of L. The amount of memory they use is several times larger than the dataset size.

Impact of n. Fig. 6c shows the impact of increasing n on the runtime of all algorithms. Both our algorithms substantially outperformed MBA, as they both scale quasi-linearly with n. MSA is comparable to MIA but much more memory-efficient (see Fig. 6d). For instance, it needed less than 4 GBs of RAM to process the CHR dataset, whose size is 22.5 GBs, while MIA and MSA needed more than 232 GBs of RAM.

Acknowledgments

GB is supported by the microgrant J93C22001380002; AC, RG, VG, NP by NextGeneration EU programme PNRR ECS00000017 Tuscany Health Ecosystem; AC, RG, VG are partially supported by MUR PRIN 20174LF3T8 AHeAD; AC and RG by MUR 2022TS4Y3N EXPAND; NP by MUR PRIN 2022YRB97K PINC; SPP by the PANGAIA project (GA 872539); and NP and SPP by the ALPACA project (GA 956229).

References

[1] D. C. Koboldt, K. M. Steinberg, D. E. Larson, R. K. Wilson, and E. R. Mardis, "The next-generation sequencing revolution and its impact on genomics," *Cell*, vol. 155, no. 1, pp. 27–38, 2013.

[2] Y. Yan, L. Cao, S. Madden, and E. A. Rundensteiner, "Swift: Mining representative patterns from large event streams," *PVLDB*, vol. 12, no. 3, p. 265–277, 2018.

[3] Y. Zhang, Y. Wei, and J. Ren, "Multi-touch attribution in online advertising with survival theory," in *ICDM*, 2014.

[4] J. Yin, Z. Zheng, and L. Cao, "USpan: an efficient algorithm for mining high utility sequential patterns," in *KDD*, 2012.

[5] W. Gan, J. Lin, P. Fournier-Viger, H. Chao, V. S. Tseng, and P. S. Yu, "A survey of utility-oriented pattern mining," *TKDE*, vol. 33, no. 4, pp. 1306–1327, 2021.

[6] O. K. Alkan and P. Karagoz, "Crom and huspext: Improving efficiency of high utility sequential pattern extraction," *TKDE*, vol. 27, no. 10, pp. 2645–2657, 2015.

[7] W. Gan, J. C.-W. Lin, J. Zhang, H.-C. Chao, H. Fujita, and P. S. Yu, "Proum: Projection-based utility mining on sequence data," *Inf. Scie.*, vol. 513, pp. 222–240, 2020.

[8] B. Ewing, L. Hillier, M. C. Wendl, and P. Green, "Base-calling of automated sequencer traces using phred. i. accuracy assessment," *Gen. Res.*, vol. 8, pp. 175–185, 1998.

[9] A. Vlavianos, L. K. Law, I. Broustis, S. V. Krishnamurthy, and M. Faloutsos, "Assessing link quality in ieee 802.11 wireless networks: Which is the right metric?," in *PIMRC*, 2008.

[10] P. W. Farris, N. T. Bendle, P. E. Pfeifer, and D. J. Reibstein, *Marketing Metrics*. Wharton School Publishing, 2nd ed., 2010.

[11] M. Xu and Z. Su, "A novel alignment-free method for comparing transcription factor binding site motifs," *PLOS One*, vol. 1, no. e8797, 2010.

[12] H. Arimura and T. Uno, "An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence," *J. Comb. Optim.*, vol. 13, no. 3, pp. 243–262, 2007.

[13] J. Fischer, V. Heun, and S. Kramer, "Fast frequent string mining using suffix arrays," in *ICDM*, 2005.

[14] J. Dhaliwal, S. J. Puglisi, and A. Turpin, "Practical efficient string mining," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 4, pp. 735–744, 2012.

[15] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB*, p. 487–499, 1994.

[16] U. Manber and E. W. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.

[17] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," in *CPM*, 2001.

[18] S. Vinga and J. Almeida, "Alignment-free sequence comparison – a review," *Bioinformatics*, vol. 19, no. 4, pp. 513–523, 2003.

[19] `www.ncbi.nlm.nih.gov/nuccore/556503834`.

[20] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. R&D*, vol. 31, no. 2, pp. 249–260, 1987.

[21] G. Cormode and S. M. Muthukrishnan, "Approximating data with the count-min sketch," *IEEE Softw.*, vol. 29, no. 1, pp. 64–69, 2012.

[22] J. Fischer, V. Mäkinen, and N. Valimaki, "Space efficient string mining under frequency constraints," in *ICDM*, 2008.

[23] F. Zhu, X. Yan, J. Han, and P. S. Yu, "Efficient discovery of frequent approximate sequential patterns," in *ICDM*, 2007.

[24] J. Jestes, F. Li, Z. Yan, and K. Yi, "Probabilistic string similarity joins," in *SIGMOD*, 2010.

[25] Y. Li, J. Bailey, L. Kulik, and J. Pei, "Efficient matching of substrings in uncertain sequences," in *SDM*, 2014.