**Noname manuscript No.**
(will be inserted by the editor)

# Space-efficient Substring Occurrence Estimation

**Alessio Orlandi · Rossano Venturini**

the date of receipt and acceptance should be inserted later

**Abstract** In this paper we study the problem of estimating the number of occurrences of substrings in textual data: A text $T$ on some alphabet $\Sigma = [\sigma]$ of length $n$ is preprocessed and an index $\mathcal{I}$ is built. The index is used in lieu of the text to answer queries of the form $\mathsf{Count}_{\approx}(P)$, returning an approximated number of the occurrences of an arbitrary pattern $P$ as a substring of $T$. The problem has its main application in selectivity estimation related to the *LIKE* predicate in textual databases. Our focus is on obtaining an algorithmic solution with guaranteed error rates and small footprint. To achieve that, we first enrich previous work in the area of compressed text-indexing providing an optimal data structure that, for a given additive error $\ell \geq 1$, requires $\Theta(\frac{n}{\ell}\log\sigma)$ bits. We also approach the issue of guaranteeing exact answers for sufficiently frequent patterns, providing a data structure whose size scales with the amount of such patterns. Our theoretical findings are supported by experiments showing the practical impact of our data structures.

The work is an extended version of the paper appeared in the *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS) 2011* [23]. This work has been partially supported by MIUR of Italy under projects PRIN ARS Technomedia 2012, the Midas EU Project, Grant Agreement no. 318786 and the InGeoCloudS EU Project, Grant Agreement no. 297300.
The work was conducted while the first author was a Ph.D. student at the Dept. of Computer Science, University of Pisa.

Alessio Orlandi
Google Switzerland.

Rossano Venturini
Department of Computer Science, University of Pisa, Italy.

E-mail: oalessio@google.com, rossano.venturini@unipi.it

## 1 Introduction

A large fraction of the data we process every day are texts: unformatted natural language documents, XML structured data, HTML collections, textual columns in relational databases, biological sequences, are just few examples. With nowadays growth of data it is common to perform operations on massive data sets. Thinking about text, the basic class of operations is simple pattern matching queries (or its variations, e.g., regular expressions). The challenge, especially on massive data sets, is to obtain low time complexities and little space requirements. On one hand, one would like to achieve the maximum speed in solving matching queries on the text, and thus indexing the data is mandatory. On the other hand, when massive data sets are involved, the cost for extra index data may be non-negligible, and thus compressing the data is mandatory too. It is not surprising that the last decade has seen a trending growth of *compressed text indexes* [9,11,12,15,22]. Their main role is to match both requirements at the same time, allowing textual data to be stored in compressed format while being able to efficiently perform pattern matching queries on the indexed text itself.

Nonetheless, there exist lower bounds on the compression ratio these indexes can achieve, usually expressed in terms of the $k$th order empirical entropy of the underlying text. Without making assumptions on the text at hand, such limits can be only surpassed by allowing pattern matching operations to have approximated results, namely, a small amount of error may be present in the output. In this paper we proceed on this path by addressing the following problem.

Substring Occurrence Estimation Problem. Given a text $T[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$ and any fixed error parameter $\ell \geq 1$, we would like to design an index that, without the need of accessing/storing the original text, is able to count the number of occurrences of any pattern $P[1,p]$ in $T$. In the *uniform error* case, the index is allowed to err by at most $\ell$: precisely, the reported number of occurrences of $P$ is in the range $[\mathsf{Count}(P), \mathsf{Count}(P)+\ell-1]$, where $\mathsf{Count}(P)$ is the actual number of occurrences of $P$ in $T$. This operation is referred to as $\mathsf{Count}{\approx_\ell}(P)$. In the *lower-sided error* case, instead, we consider the stronger operation $\mathsf{Count}{\geq_\ell}(P)$ such that $\mathsf{Count}{\geq_\ell}(P) = \mathsf{Count}(P)$ whenever $\mathsf{Count}(P) \geq \ell$, and $\mathsf{Count}{\geq_\ell}(P) = \ell - 1$ otherwise.

A relative of additive error is the multiplicative error, i.e., when the estimation lays in $[\mathsf{Count}(P), (1+\varepsilon)\mathsf{Count}(P)]$ for some fixed $\varepsilon > 0$, which provides better estimates for low frequency patterns. Unfortunately, solving the multiplicative error problem would imply an index able to discover for sure whether a pattern $P$ appears in $T$ or not (set $\mathsf{Count}(P) = 0$ in the above expression). This turns out to be the hard part of estimation and, indeed, we prove (Theorem 5) that an index with multiplicative error would require as much as $T$ to be represented. Hence, the forthcoming discussion will focus solely on additive error.

Occurrence estimation finds its main application in *Substring Selectivity Estimation*: given a textual column of a database, create a limited space index that finds (approximately) the percentage of rows satisfying the predicate *LIKE '%P%'* for any pattern $P$. Provided with a data structure for substring occurrence estimation with lower-sided error, solutions in literature [6,18,19] try to reduce the error when the data structure is not able to guarantee a correct answer, i.e., $\mathsf{Count}(P) < \ell$. This phase, called *error reduction*, usually involves splitting $P$ into pieces appearing in the data structure and using a probabilistic model to harness such information to generate a selectivity estimate for the whole pattern. Apart from providing an effective model, solutions for substring selectivity incur in a space/error trade-off: the more space-efficient is the underlying data structure, the more information can be stored, hence yielding a more accurate estimate. Most of the data structures used in selectivity estimation are simple and waste space; therefore, we can indirectly boost selectivity accuracy by studying space-efficient substring occurrence estimation.

In the forthcoming discussion, we focus on occurrence estimation on whole texts only. Nonetheless, the techniques immediately apply to collections of strings (i.e., rows in a database column): given the content of strings $R_1, R_2, \ldots, R_n$ we introduce a new special symbol $\triangleright$ and create the text $T(R) = \triangleright R_1 \triangleright R_2 \triangleright \cdots \triangleright R_n \triangleright$. A substring query is then performed directly on $T(R)$.

To date, the main data structure for occurrence estimation is the *pruned suffix tree* $\mathsf{PST}_\ell(T)$ [19]. Here, we briefly review its main characteristics and defer a full explanation to the next section. For a fixed error $\ell \geq 1$, the $\mathsf{PST}_\ell(T)$ is obtained from the suffix tree of $T$ by pruning away all nodes having less than $\ell$ leaves in their subtrees. This suffices for solving the substring occurrence estimation problem with lower-sided error. However, the space occupancy of $\mathsf{PST}_\ell$ is a serious issue, both in theory and practice: it requires a total of $O(m \log n + g \log \sigma)$ bits, where $m$ is the number of nodes surviving the pruning phase and $g$ is the amount of symbols that label the edges of such nodes. This is problematic for two reasons: 1) the number of nodes in the pruned tree may be very high (up to $\Theta(n - \ell)$); 2) the number of symbols on the edges may exceed the length of the text itself. Thus, the space occupancy of the pruned suffix tree may be not sublinear in the text size and can be very far from the $\Omega(\frac{n \log \sigma}{\ell})$ bits space lower bound for the problem (see Theorem 4).

Solutions with smaller footprint can be obtained by resorting to the compressed full-text indexes [9, 11, 15, 22], which are well known in the field of succinct data structures. These indexes deliver a framework to keep a copy of text $T$ compressed together with auxiliary information for efficient substring search. Such solutions, however, work on the entire text and are not designed to allow errors or pruning of portions of the string, yet they provide a good baseline for our work.

In this paper we show (Section 4) how to build an index (called $\mathsf{APX}_\ell$) that requires $O(\frac{n \log(\sigma\ell)}{\ell})$ bits of space. This is the first index that has both guaranteed space, sublinear with respect to the size of the indexed text, and provable error bounds. It turns out (Theorem 4) that such index is space-optimal up to constant factors for sufficiently small values of $\ell$ (namely, $\log\ell = O(\log\sigma)$).

We also provide a data structure ($\mathsf{CPST}_\ell$) for the lower-sided error version of the problem (Section 5) that presents a space bound of $O(m\log(\sigma\ell))$ bits where $m$ is the number of nodes in the $\mathsf{PST}_\ell(T)$. Hence, our $\mathsf{CPST}_\ell$ does not require to store the labels (the $g\log\sigma$ term above), which account for most of the space in practice. Such data structure outperforms our previous solution only when $m = O(n/\ell)$, a condition that is satisfied in many real data sets[1]. Both the $\mathsf{APX}_\ell$ and $\mathsf{CPST}_\ell$ data structures heavily rely on the Burrows-Wheeler Transform [5], which proves to be an effective tool to tackle the problem.

In Section 6 we support our claims with tests on real data sets. We show the improvement in space occupancy of both $\mathsf{APX}_\ell$ and $\mathsf{CPST}_\ell$, both ranging from 5x to 60x w.r.t. to $\mathsf{PST}_\ell$, and we show our sharp advantage over compressed text indexing solutions. As an example, for an English text of about 512 MB, it suffices to set $\ell = 256$ to obtain an index of 5.1 MB (roughly, 1%). We also confirm that $m$ and $n/\ell$ are close most of the times. In such sense we also note that the main component in $\mathsf{PST}_\ell$'s space is given by the labels, hence guaranteeing to our $\mathsf{CPST}_\ell$ a clear advantage over $\mathsf{PST}_\ell$.

Concerning the selectivity estimation problem, we illustrate the gain in estimation quality given by employing our indexes as underlying data structure. For such purposes we employ the `MOL` algorithm (see [18]). We study the average additive error by choosing a first threshold for $\mathsf{PST}_\ell$ and a second one for $\mathsf{CPST}_\ell$ such that the resulting indexes occupy roughly the same space. Results show that $\mathsf{CPST}_\ell$ induces an improvement ranging from 5x to 790x depending on the source of data. Combining `MOL` and our `CPST` with reasonably small $\ell$, it is possible to solve the selectivity estimation problem with an average additive error of 1 by occupying (on average) around 1/7 of the original text size.

## 2 Preliminaries

A basic concept of text compression is the empirical entropy of a text. Let $T[1,n]$ be a string drawn from the alphabet $\Sigma = [\sigma]$.[2] For each $c \in \Sigma$, we let $n_c$ be the number of occurrences of $c$ in $T$. The zero-th order *empirical* entropy of $T$ is defined as

$$H_0(T) = (1/n)\sum_{c\in\Sigma} n_c \log(n/n_c).$$

The quantity $nH_0(T)$ bits provides an information-theoretic lower bound to the output size of any compressor that encodes each symbol of $T$ with a fixed code [20, 25].

For any string $w$ of length $k$, we denote by $w_T$ the string of single symbols following the occurrences of $w$ in $T$, taken from left to right. For example, if

[1] Recall that the condition on $m$ is not enough to obtain a small $\mathsf{PST}_\ell$ due to the edge labels.

[2] In the following we will adopt the common assumption that $\sigma = O(n)$.

$T = \mathsf{banabananab}$ and $w = \{\mathsf{an}\}$, we have $w_T = \{\mathsf{aaa}\}$ since the three occurrences of $\mathsf{an}$ in $T$ are followed by the symbol $\mathsf{a}$. The $k$-th order *empirical* entropy of $T$ is defined as

$$H_k(T) = (1/n) \sum_{w \in \Sigma^k} |w_T| \, H_0(w_T).$$

We have $H_k(T) \geq H_{k+1}(T)$ for any $k \geq 0$. The quantity $nH_k(T)$ bits provides an information-theoretic lower bound to the output size of any compressor that encodes each symbol of $T$ with a code that depends on the symbol itself and on the $k$ immediately preceding symbols [20].

## 2.1 Rank and Select queries

Our solutions rely on basic data structures that can answer $\mathsf{Rank}$ and $\mathsf{Select}$ queries. Let $T[1,n]$ be a text drawn from an alphabet $\Sigma = [\sigma]$. The query $\mathsf{Rank}_c(T,i)$ for $c \in \Sigma$ counts the number of occurrences of the symbol $c$ in the prefix $T[0:i]$. The query $\mathsf{Select}_c(T,i)$ for $c \in \Sigma$ returns the position of the $i$th occurrence of the symbol $c$ in $T$, or $-1$ if not existing. The space/time complexities of the best solutions on the RAM model with word size $w$ are summarized in the following theorem (see [4] and references therein).

**Theorem 1** *Given a text $T[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$, there exist data structures storing $T$ within $nH_0(T) + o(n)$ bits and supporting* $\mathsf{Rank}$ *and* $\mathsf{Select}$ *queries in $O(1 + \log \frac{\log \sigma}{\log w})$ time.*

Notice that, since $w = \Omega(\log n)$ bits, $\mathsf{Rank}/\mathsf{Select}$ queries require constant time for $\sigma = O(polylog(n))$.

In our solutions we also employ the so-called Elias-Fano representation [7,8] that supports the query $\mathsf{Select}_1$ over a binary vector $B[1,n]$ in constant time. The main advantage of this representation is that, depending on the relation between $n$ and the number $m$ of bits set to 1, it may be more space efficient than the results in Theorem 1. Its time/space complexities are reported in the following theorem.

**Theorem 2** *There exists a data structure encoding a bitvector $B$ of length $n$ with $m$ bits set to 1 in $m \log \frac{n}{m} + O(m)$ bits, supporting* $\mathsf{Select}_1$ *in $O(1)$ time.*

## 2.2 Suffix trees

The *Suffix Tree* of a text $T$, denoted as $\mathsf{ST}(T)$, is the compacted trie built on all the $n$ suffixes of $T$ (see e.g., [16]). We assume that a special symbol, say \$, terminates the text $T$ to ensure that no suffix is a proper prefix of another suffix. The symbol \$ does not appear anywhere else in $T$ and is assumed to be lexicographically smaller than any other symbol in $\Sigma$. In this way each suffix of $T$ has its own unique leaf in the suffix tree, since any two suffixes of $T$ will eventually follow separate branches in the tree. The label of any edge $e = (u,v)$

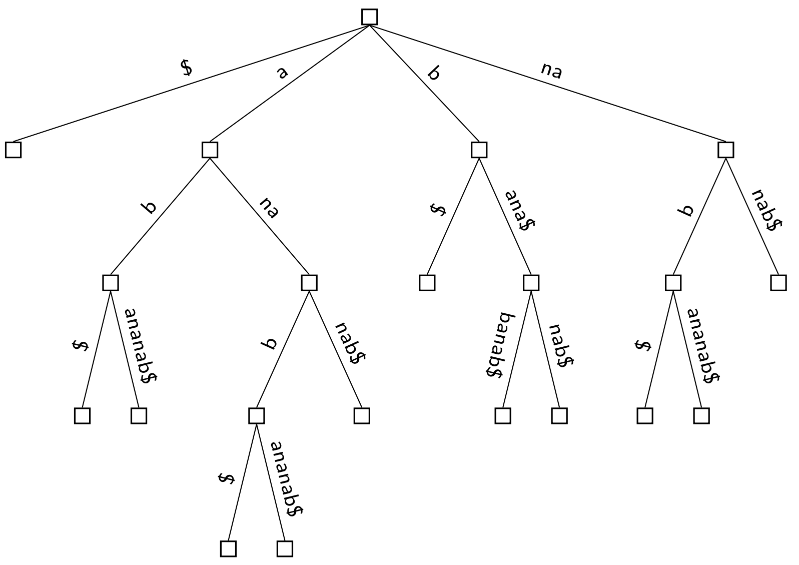


Fig. 1: The suffix tree for the text banabananab$.

in $\mathsf{ST}(T)$, denoted as $\mathsf{label}(u,v)$, is always a non-empty substring of $T$. For a given node $u$ in $\mathsf{ST}(T)$, its *path label* (denoted $\mathsf{pathlabel}(u)$) is defined as the concatenation of edge labels on the path from the root to $u$. The *string depth* of node $u$ is $|\mathsf{pathlabel}(u)|$. See Figure 1 for the suffix tree built on the text $T = \mathsf{banabananab\$}$.

The search of a pattern $P[1,p]$ in $T$ has to identify the highest node $u$ in $\mathsf{ST}(T)$ such that $P$ prefixes $\mathsf{pathlabel}(u)$, if any. The search starts from the root of $\mathsf{ST}(T)$ and follows a path trying to match symbols of $P$. The search stops as soon as a mismatch occurs or $P$ is completely matched. In the former case $P$ does not occur in $T$. In the latter case, each leaf in the subtree below the matching position gives an occurrence of $P$. The number of these occurrences can be obtained in constant time by simply storing in each node $u$ the number $C(u)$ of leaves in its subtree. Therefore, the number of occurrences of any pattern $P[1,p]$ is computed in $O(p \log \sigma)$ time. This time complexity can be reduced up to $O(p)$ by placing a (minimal) perfect hashing function [17] in each node to speed up percolation, while increasing the space only by a constant factor.

## 2.3 Burrows-Wheeler Transform

Burrows and Wheeler [5] introduced a new compression algorithm based on a reversible transformation, now called the *Burrows-Wheeler Transform* (Bwt from now on). The Bwt permutes the symbols of the input string $T$ to obtain a new string which is easier to compress. We assume, again, that $T$ is terminated

| |
|---|
| banabananab$ |
| anabananab$b |
| nabananab$ba |
| abananab$ban |
| bananab$bana |
| ananab$banab |
| nanab$banaba |
| anab$banaban |
| nab$banabana |
| ab$banabanan |
| b$banabanana |
| $banabananab |

$\Longrightarrow$

| F | | L |
|---|---|---|
| $ | banabanana | b |
| a | b$banabana | n |
| a | bananab$ba | n |
| a | nab$banaba | n |
| a | nabananab$ | b |
| a | nanab$bana | b |
| b | $banabanan | a |
| b | anabananab | $ |
| b | ananab$ban | a |
| n | ab$banaban | a |
| n | abananab$b | a |
| n | anab$banab | a |

Fig. 2: Example of Burrows-Wheeler transform for the string $T = \mathsf{banabananab\$}$. The matrix on the right has the rows sorted in lexicographic order. The output of the Bwt is the column $L = \mathsf{bnnnbba\$aaaa}$.

by an extra symbol \$ smaller than any other symbol of $\Sigma$. The Bwt of $T$, hereafter denoted by $\mathsf{Bwt}(T)$, is built with two basic steps (see Figure 2):

1. form a *conceptual* matrix $\mathcal{M}(T)$ whose rows are the cyclic rotations of string $T$ in lexicographic order;
2. construct string $L$ by taking the last column of the sorted matrix $\mathcal{M}(T)$. We set $\mathsf{Bwt}(T) = L$.

Every column of $\mathcal{M}(T)$, hence also the transformed string $L$, is a permutation of $T\$$. In particular the first column of $\mathcal{M}(T)$, call it $F$, is obtained by lexicographically sorting the symbols of $T$ (or, equally, the symbols of $L$). Note that the sorting of the rows of $\mathcal{M}(T)$ is essentially equal to the sorting of the suffixes of $T$, because of the presence of the special symbol \$. This shows that: (1) symbols following the same substring (*context*) in $T$ are grouped together in $L$, and thus give raise to clusters of nearly identical symbols; (2) there is an obvious relation between $\mathcal{M}(T)$ and suffix tree/array of $T$. Property 1 is the key for devising modern data compressors (see e.g., [20]); Property 2 is crucial for designing compressed indexes (see e.g., [9, 22]) and, additionally, suggests a way to compute the Bwt in linear time through the construction of the suffix array of $T$.

Burrows and Wheeler [5] observed two properties that guarantee the invertibility of the Bwt:

**(a)** Since the rows in $\mathcal{M}(T)$ are cyclically rotated, $L[i]$ *precedes* $F[i]$ in the original string $T$.

**(b)** For any $c \in \Sigma$, the $j$-th occurrence of $c$ in $F$ and the $j$-th occurrence of $c$ in $L$ correspond to the *same* symbol of the string $T$.

As a result, the original text $T$ can be obtained backwards from $L$ by resorting to function $LF$ (also called LF-mapping) that maps row indexes to row indexes, and is defined as

$$LF(i) = C[L[i]] + \mathsf{Rank}_{L[i]}(L, i),$$

**Algorithm** Count($P[1,p]$)

```
1. $i = p$, $c = P[p]$, First $= C[c]+1$, Last $= C[c+1]$;
2. while ((First $\leq$ Last) and ($i \geq 2$)) do
3.     $c = P[i-1]$;
4.     First $= C[c] + \mathsf{Rank}_c(L, \mathsf{First}-1)+1$;
5.     Last $= C[c] + \mathsf{Rank}_c(L, \mathsf{Last})$;
6.     $i = i-1$;
7. if (Last $<$ First) then return "no rows prefixed by $P$"
8. else return [First, Last].
```

Fig. 3: The algorithm to find the range [First, Last] of rows of $\mathcal{M}(T)$ prefixed by $P[1,p]$.

where $C[L[i]]$ counts the number of occurrences in $T$ of symbols smaller than $L[i]$. We talk about LF-mapping because the symbol $c = L[i]$ is located in the first column, $F$, of $\mathcal{M}(T)$ at position $LF(i)$. The LF-mapping allows one to navigate $T$ backwards: if $T[k] = L[i]$, then $T[k-1] = L[LF(i)]$ because row $LF(i)$ of $\mathcal{M}(T)$ starts with $T[k]$ and, thus, ends with $T[k-1]$. In this way, we can reconstruct $T$ backwards by starting at the first row, equal to $\$T$, and repeatedly applying $LF$ for $n$ steps. As an example, observe that the second a in $L$ of Figure 2 lies onto the row which starts with bananab\$ and, correctly, the second a in $F$ lies onto the row which starts with abananab\$. That symbol a is $T[4]$.

## 2.4 Compressed Full-text Indexing

Ferragina and Manzini [11] proved that data structures for supporting Rank queries on the string $L = \mathsf{Bwt}(T)$ suffice to search for an arbitrary pattern $P[1,p]$ as a substring of $T$. Their searching procedure, called *backward search*, is reported in Figure 3. It works in $p$ phases, each preserving a simple invariant: *At the end of the i-th phase,* [First, Last] *is the range of contiguous rows in* $\mathcal{M}(T)$ *which are prefixed by* $P[i,p]$. Count starts with $i = p$ so that First and Last are determined via the array $C$ (step 1). Ferragina and Manzini proved that the pseudo-code in Figure 3 maintains the invariant above for all phases, so [First, Last] delimits at the end the rows prefixed by $P$ (if any). Steps 4 and 5 are dominant costs of each iteration of Count. While array $C$ is small and occupies $O(\sigma \log n)$ bits, efficient support of Rank queries over Bwt is guaranteed by the data structures described in Subsection 2.1. The space occupancy of a solution achieving bounds in terms of $H_0(T)$ on an arbitrary text $T$ can be bounded in terms of $H_k(T)$ when applied to $L = \mathsf{Bwt}(T)$.

By plugging these considerations into Count algorithm (see e.g., [2]), we obtain the following theorem.

**Theorem 3** *Given a text* $T[1,n]$ *drawn from an alphabet* $\Sigma = [\sigma]$, $\sigma \leq n$, *there exists a compressed index that takes* $O(p(1+\log\frac{\log\sigma}{\log w}))$ *time to support*

$\mathsf{Count}(P[1,p])$ *and requires* $nH_k(T)+o(n)(H_k(T)+1)$ *bits of space, for any* $k \leq (\alpha \log_\sigma n)-1$ *and constant* $0<\alpha<1$.

Notice that compressed indexes support also other operations, like locate and display of pattern occurrences, which are slower than Count in that they require $O(polylog(n))$ time per occurrence [9, 22]. We do not go into further details on these operations because they are not required in our solution.

## 2.5 Solutions for the Substring Occurrence Estimation Problem

The main data structure for occurrence estimation, and the one used in `KVI` and `MO` [18, 19], is the *pruned suffix tree* $\mathsf{PST}_\ell(T)$. For a fixed error $\ell \geq 1$, the $\mathsf{PST}_\ell(T)$ is obtained from the suffix tree of $T$ by pruning away all nodes of suffixes that appear less than $\ell$ times in $T$.

It is immediate to see that the resulting data structure solves the Substring Occurrence Estimation Problem with lower-sided error in time $O(|P|)$. However, the space occupancy of $\mathsf{PST}_\ell$ may be a serious issue, both in theory and practice. Indeed, the $\mathsf{PST}_\ell$ requires a total of $O(m\log n + g\log\sigma)$ bits, where $m$ is the number of nodes surviving the pruning phase and $g$ is the amount of symbols that label the edges of such nodes. This space complexity may be problematic because both $m$ and $g$ may be $\Omega(n)$ even for very large values of $\ell$ (i.e., the error). Indeed, depending on the text $T$, the number of nodes in the pruned tree could raise to $\Theta(n-\ell)$ and could slowly decrease as the error $\ell$ increases: observe that we require to increase the error up to $n/2$ just to halve the number of nodes in the tree. Consider the text $T = a^n\$$. The shape of its suffix tree is a long chain of $n-1$ nodes with two children each. Therefore, for any value of $\ell$, the space required to store explicitly its pruned suffix tree is at least $\Omega((n-\ell)\log n)$ bits. This quantity further increases due to the need of storing explicitly edges' labels. We point out that the number of these symbols is at least equal to the number of nodes but can significantly increase whenever the suffixes represented in the tree share long common prefixes and it may even exceed the length of the text itself. This implies that the space occupancy of the pruned suffix tree may be not sublinear in the text size.

An alternative to the pruning strategy above consists in building a pruned Patricia trie [21] $\mathsf{PT}_{\ell/2}(T)$ indexing just a suffix every $\ell/2$ suffixes of $T$ sorted lexicographically. More formally, let $S_1, S_2, \ldots, S_n$ denote the $n$ suffixes of $T$ sorted lexicographically, $\mathsf{PT}_{\ell/2}(T)$ is the Patricia trie of the set of $O(n/\ell)$ strings $\mathcal{S} = \{S_i \mid i \equiv 1 \pmod{\ell/2}\}$. The pruned Patricia trie $\mathsf{PT}_{\ell/2}(T)$ can be stored in $O(\frac{n(\log\sigma+\log n)}{\ell}) = O(\frac{n\log n}{\ell})$ bits. We use the blind search [10] to search for a pattern $P[1,p]$ in time $O(p)$. Such algorithm returns a node $u$ that either corresponds to $P$, if $P$ is a prefix of some string in $\mathcal{S}$, or another node otherwise (whereas there is a connection between such node and $P$, it is not possible to exploit it without the original text). Once we identify the node $u$, we return the number of leaves descending from that node multiplied by $\ell$. If $P$ occurs at least $\ell/2$ times in $T$, it is easy to see that the number of reported occurrences is a correct approximation of its number of occurrences in $T$. However, if $P$ occurs less than $\ell/2$ times in $T$, the algorithm may fail by reporting as result a number of occurrences which may be arbitrarily far from the correct one.

A similar solution resorts to a data structure presented by Belazzougui *et al.* [3]. Their solution solves via (perfect) hash functions a problem somehow related to ours, called *Weak Prefix Search*. The problem is as follows: We have a set $\mathcal{V}$ of $v$ strings and want to build an index on them. Given a pattern $P[1,p]$, the index outputs the ranks (in lexicographic order) of the strings that have $P$ as prefix; if such strings do not exist the output of the index is arbitrary. Their main solution needs just $O(p\log(\sigma)/w + \log p + \log\log\sigma)$ time and $O(v\log(L\log\sigma))$ bits of space, where $L$ is the average length of the strings in the set and $w$ is the machine word size. We can use their data structure to index the set of suffixes $\mathcal{S}$, so that we can search $P[1,p]$ and report its number of occurrences multiplied by $\ell$. Since in our case $L = \Theta(n)$, the index requires $O(\frac{n\log(n\log\sigma)}{\ell}) = O(\frac{n\log n}{\ell})$ bits of space. As in the case of pruned Patricia trie, the answer is arbitrary when $P$ is not prefix of any suffix in $\mathcal{S}$ (i.e., it occurs less that $\ell$ times). Hence, this solution improves the time complexity of the previous approach but retains the same important drawback. We conclude by mentioning a solution [13] that solves the weak prefix search problem efficiently in the Cache-Oblivious Model [14], and, thus, makes the above approach suitable for this model. Applying this solution we can search for a pattern $P[1,p]$ in $O(\log_B n + (p\log\sigma)/B)$ I/Os in the Cache-Oblivious Model by using $O(\frac{n\log n}{\ell})$ bits of space, where $B$ is the number of bits that fit in a block.

## 2.6 Algorithms for the Substring Selectivity Estimation

In this section we present in detail the three main algorithms for substring selectivity estimation: `KVI` [19], the class of `MO`-based estimators [18] and `CRT` [6], in chronological order. All the algorithms we will describe rely on an underlying solution for the substring occurrence estimation problem and, thus, represent an application for our solutions.

For a given threshold $\ell$, `KVI` starts by assuming to have a data structure answering correctly to queries $\mathsf{Count}(P)$ when $\mathsf{Count}(P) \geq \ell$ and strives to obtain a one-sided error estimate for infrequent ($< \ell$) strings. It also assumes the data structure can detect if $\mathsf{Count}(P) < \ell$. Its main observation is as follows: let $P = \alpha\beta$ where $\mathsf{Count}(P) < \ell$ and assume $\mathsf{Count}(\alpha) \geq \ell$ and $\mathsf{Count}(\beta) \geq \ell$, then one can estimate $\mathsf{Count}(P)$ from $\mathsf{Count}(\alpha)$ and $\mathsf{Count}(\beta)$ in a probabilistic way, using a model in which the probability of $\beta$ appearing in the text given that $\alpha$ appears is roughly the same of $\beta$ appearing by itself. Generalizing this concept, `KVI` starts from $P$ and retrieves the longest prefix of $P$, say $P'$, such that $\mathsf{Count}(P') > \ell$, and then repeats on the remaining suffix.

Requiring the same kind of data structure beneath, the `MO` class starts by observing that, instead of splitting the pattern $P$ into known fragments of information, one can rely on the concept of maximum overlap: given two strings $\alpha$ and $\beta$, the maximum overlap $\alpha \oslash \beta$ is the longest prefix of $\beta$ that is also a suffix of $\alpha$. Hence, instead of estimating $\mathsf{Count}(P)$ from $\mathsf{Count}(\alpha)$ and $\mathsf{Count}(\beta)$ alone, it also computes and exploits the quantity $\mathsf{Count}(\alpha \oslash \beta)$. In probabilistic terms, this is equivalent to introducing a light form of conditioning between pieces of the string, hence yielding better estimates. The change is justified by an empirically proved Markovian property that makes maximum overlap estimates very significant. `MO` is also presented in different variants: `MOC`, introducing constraints from the strings to avoid overestimation, `MOL`, performing a more thorough search of substrings of the pattern, and `MOLC`, combining the two previous strategies.

In particular, `MOL` relies on the *lattice* $\mathcal{L}_P$ of the pattern $P$. For a string $P = a \cdot \alpha \cdot b$ ($|\alpha| \geq 0$), the *l-parent* of $P$ is the string $\alpha \cdot b$ and the *r-parent* of $P$ is $a \cdot \alpha$. The lattice $\mathcal{L}_P$ is described recursively: $P$ is in the lattice and for any string $\zeta$ in the lattice, also its $l$-parent and $r$-parent are in the lattice. Two nodes $\beta$ and $\zeta$ of the lattice are connected if $\beta$ is an $l$-parent or an $r$-parent of $\zeta$ or viceversa. To estimate $\mathsf{Count}(P)$, the algorithm starts by identifying all nodes in the lattice for which $\mathsf{Count}(\alpha)$ can be found in the underlying data structure and retrieve it, so that $Pr(\alpha) = \mathsf{Count}(\alpha)/N$, where $N$ is a normalization factor. For all other nodes, it computes $Pr(a \cdot \alpha \cdot b) = Pr(a \cdot \alpha) \times Pr(\alpha \cdot b)/Pr(a \cdot \alpha \oslash \alpha \cdot b)$ recursively. In the end, it obtains $Pr(P)$, i.e., the normalized ratio of occurrences of $P$ in $T$.

The `CRT` method was presented to circumvent underestimation, a problem that may afflict estimators with limited probabilistic knowledge as those above. The first step is to build an a-priori knowledge of which substrings are highly distinctive in the database: in that, they rely on the idea that most patterns exhibit a short substring that is usually sufficient to identify the pattern itself. Given a pattern to search for, they retrieve all distinctive substrings of the pattern and use a machine learning approach to combine their value. At construction time, they train a regression tree over the distinctive substrings by using a given query log; the tree is then exploited at query time to obtain a final estimate.

## 3 Substring Occurrence Estimation Problem: Lower bounds

The following lower bound establishes the minimum amount of space needed to solve the substring occurrence estimation problem for both error types, uniform and lower-sided.

**Theorem 4** *For any fixed additive error $\ell \geq 1$, an index built on a text $T[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$ that approximates the number of occurrences of any pattern $P$ in $T$ within $\ell$ must use $\Omega(\frac{n \log \sigma}{\ell})$ bits of space.*

*Proof* Assume there exists an index requiring $o(\frac{n \log \sigma}{\ell})$ bits of space that is able to answer any approximate counting query within an additive error $\ell$. Given any text $T[1,n]$, we derive a new text $T'[1,(2\ell+1)(n+1)]$ that is formed by repeating the string $T\$$ for $2\ell+1$ times, where \$ is a symbol that does not belong to $\Sigma$. Then, we build the above index on $T'$, which would require $o(\frac{(2\ell+1)(n+1)\log(\sigma+1)}{\ell}) = o(n\log\sigma)$ bits. We observe that we can recover the original text $T$ by means of this index. We search for all possible texts of length $n$ drawn from $\Sigma$ followed by a \$. The original text $T$ will be the only one for which the index answers with a value greater than $\ell$. However, since there must exist at least a text of length $n$ from the alphabet $[\sigma]$ requiring $\log\sigma^n = n\log\sigma$ bits to be represented, the index built on this text would require too few bits, a contradiction. $\square$

Using similar considerations we can prove the following Theorem, which justifies the need of focusing on additive errors.

**Theorem 5** *For any fixed multiplicative error $(1+\varepsilon) \geq 1$, an index built on a text $T[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$ that approximates the number of occurrences of any pattern $P$ in $T$ within $(1+\epsilon)$ must use $\Omega(n\log\sigma)$ bits of space.*

*Proof* The proof uses a standard argument. Assume there exists an index that, for any $T[1,n]$, requires fewer than $n\log\sigma$ bits and answers any approximate counting query within a multiplicative error $(1+\varepsilon) \geq 1$. We can recover the original text $T$ by means of this index. We search for all possible texts of length $n$ drawn from $\Sigma$. The text $T$ will be the only one for which the index answers with a value greater than 0. The existence of such index would imply a contradiction, since at least a text of length $n$ must require $n\log\sigma$ bits to be represented. $\square$

## 4 Uniform error solution

In this section we describe our first data structure, which is able to report the number of occurrences of any pattern within an additive error of at most $\ell$ by using $O(\frac{n\log(\sigma\ell)}{\ell})$ bits of space. Accordingly to Theorem 4, this error/space trade-off is optimal whenever the error $\ell$ is such that $\log\ell = O(\log\sigma)$. Formally, we will prove the following theorem.

**Theorem 6** *Given $T[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$, and a fixed error threshold $\ell$, there exists an index that answers $\mathsf{Count}_{\approx\ell}(P[1,p])$ in $O(p(1+\log\frac{\log\sigma}{\log w}))$ time by using $O(\frac{n\log(\sigma\ell)}{\ell})$ bits of space.*

The idea behind our solution is that of sparsifying the string $L = \mathsf{Bwt}(T)$ by removing most of its symbols (namely, for each symbol we just keep track of one every $\ell/2$ of its occurrences). Then, we provide an algorithm that, even though, can provide sufficiently good results on this sampled $\mathsf{Bwt}$. Similarly to the backward search, our algorithm searches for a pattern $P[1,p]$ by performing $p$ phases. In each of them, it computes two indexes of rows of $\mathcal{M}(T)$: $\mathsf{AFirst}_i$ and $\mathsf{ALast}_i$. These two indexes are obtained by first performing $\mathsf{Rank}$ queries on

**Algorithm** $\mathsf{Count}{\approx_\ell}(P[1,p])$

```
1.  $i = p$, $c = P[p]$, $\mathsf{AFirst}_p = C[c] + 1$, $\mathsf{ALast}_p = C[c+1]$;
2.  while (($\mathsf{AFirst}_i \le \mathsf{ALast}_i$) and ($i \ge 2$)) do
3.      $c = P[i-1]$;
4.      $\mathsf{DiscrFirst}_i = \mathsf{Succ}(\mathsf{AFirst}_i, \mathcal{D}_c)$
5.      $RL = \min(\mathsf{DiscrFirst}_i - \mathsf{AFirst}_i, \ell/2 - 1)$
6.      $\mathsf{AFirst}_{i-1} = \mathsf{LF}(\mathsf{DiscrFirst}_i) - RL$;
7.      $\mathsf{DiscrLast}_i = \mathsf{Pred}(\mathsf{ALast}_i, \mathcal{D}_c)$
8.      $RR = \min(\mathsf{ALast}_i - \mathsf{DiscrLast}_i, \ell/2 - 1)$
9.      $\mathsf{ALast}_{i-1} = \mathsf{LF}(\mathsf{DiscrLast}_i) + RR$;
10.     $i = i - 1$;
11. if ($\mathsf{ALast}_i < \mathsf{AFirst}_i$) then return "no occurrences of $P$"
12. else return [$\mathsf{AFirst}_i, \mathsf{ALast}_i$].
```

Fig. 4: Our algorithm to find the approximate range $[\mathsf{AFirst}_1, \mathsf{ALast}_1]$ of rows of $\mathcal{M}(T)$ prefixed by $P[1,p]$ (if any).

the sampled $\mathsf{Bwt}$ and, then, by applying a correction mechanism. Corrections are required to guarantee that both indexes are within a distance $\ell/2$ from the actual indexes $\mathsf{First}_i$ and $\mathsf{Last}_i$ that would be computed by the backward search in phase $i$. More formally, in each phase it is guaranteed that $\mathsf{AFirst}_i \in [\mathsf{First}_i - (\ell/2) - 1, \mathsf{First}_i]$ and $\mathsf{ALast}_i \in [\mathsf{Last}_i, \mathsf{Last}_i + (\ell/2) - 1]$. Clearly, also the last step obeys to this invariant, hence all rows in $[\mathsf{AFirst}_1, \mathsf{ALast}_1]$ contain suffixes prefixed by $P$, with the possible exception of the first and last $\ell/2$ ones. It follows that the algorithm errs for at most $\ell$ occurrences reporting $\mathsf{ALast}_1 - \mathsf{AFirst}_1 + 1$ as the number of occurrences of $P$.

For each symbol $c$, the sampling of $L = \mathsf{Bwt}(T)$ keeps track of a set $\mathcal{D}_c$ of positions, called *discriminant positions* (for symbol $c$), containing:

- the position of the first occurrence of $c$ in $L$;
- the positions $x$ of the $i$th occurrence of $c$ in $L$ where $i \bmod \ell/2 = 0$;
- the position of the last occurrence of $c$ in $L$.

In Figure 4 we report the procedure for searching with this sampled $\mathsf{Bwt}$. The algorithm searches for a pattern $P[1,p]$ by performing predecessor and successor queries on sets $D$s[3]. The crucial steps are lines $4-9$ where the algorithm computes the values of $\mathsf{AFirst}_{i-1}$ and $\mathsf{ALast}_{i-1}$ using the values computed in the previous phase. To understand the intuition behind these steps, let us focus on the computation of $\mathsf{AFirst}_{i-1}$ and assume that we know the value of $\mathsf{First}_i$. The original backward search would compute the number of occurrences, say $v$, of symbol $c$ in the prefix $L[1 : \mathsf{First}_i - 1]$. Since our algorithm does not have the whole $L$, the best it can do is to identify the rank, say $r$, of the position in $D_c$ closest to (but larger than) $\mathsf{First}_i$. Clearly, $r \cdot \ell/2 - \ell/2 < v \le r \cdot \ell/2$. Thus, setting $\mathsf{AFirst}_{i-1} = C[c] + r \cdot \ell/2 - \ell/2 - 1$ would suffice to guarantee that $\mathsf{AFirst}_{i-1} \in [\mathsf{First}_{i-1} - (\ell/2 - 1), \mathsf{First}_{i-1}]$. Notice that

[3] We recall that a predecessor query $\mathsf{Pred}(x, A)$ returns the predecessor of $x$ in a set $A$, namely, $\max\{y \mid y \le x \wedge y \in A\}$. A successor query is similar but finds the minimum $y$ such that $y \ge x$.

we are using the crucial assumption that the algorithm knows $\mathsf{First}_i$. If we replace $\mathsf{First}_i$ with its approximation $\mathsf{AFirst}_i$, this simple argumentation cannot be applied since the error would grow phase by phase. In this way the final error would grow up to $\ell/2 \cdot p$. Surprisingly, it is enough to use the simple correction retrieved at line 5 and applied at line 6 to fix this problem. The following Lemma provides a formal proof of our claim.

**Lemma 1** *For any fixed $\ell \geq 0$ and any phase $i$, both $\mathsf{AFirst}_i \in [\mathsf{First}_i - (\ell/2 - 1), \mathsf{First}_i]$ and $\mathsf{ALast}_i \in [\mathsf{Last}_i, \mathsf{Last}_i + \ell/2 - 1]$ hold.*

*Proof* We prove only that $\mathsf{AFirst}_i \in [\mathsf{First}_i - (\ell/2-1), \mathsf{First}_i]$ or, equivalently, that $0 \leq \mathsf{First}_i - \mathsf{AFirst}_i < \ell/2$. A similar reasoning applies for $\mathsf{ALast}_i$. The proof is by induction. For the first step $p$, we have that $\mathsf{AFirst}_p = \mathsf{First}_p$, thus the thesis immediately follows. For the inductive step, we assume that $0 \leq \mathsf{First}_i - \mathsf{AFirst}_i < \ell/2$ and we prove that $0 \leq \mathsf{First}_{i-1} - \mathsf{AFirst}_{i-1} < \ell/2$. Recall that $\mathsf{First}_{i-1}$ is computed as $C[c] + \mathsf{Rank}_c(L, \mathsf{First}_i - 1) + 1$ by the backward search and that $\mathsf{LF}(\mathsf{DiscrFirst}_i) = C[c] + \mathsf{Rank}_c(L, \mathsf{DiscrFirst}_i - 1) + 1 = C[c] + \mathsf{Rank}_c(L, \mathsf{DiscrFirst}_i)$, where the latter equality follows by observing that $L[\mathsf{DiscrFirst}_i] = c$ by definition of discriminant position for $c$. We distinguish two cases depending on the relative (and unknown) order between $\mathsf{First}_i$ and $\mathsf{DiscrFirst}_i$.

**Case 1.** Assume $\mathsf{First}_i \leq \mathsf{DiscrFirst}_i$. Let $z$ be the number of occurrences of symbol $c$ in $L[\mathsf{First}_i, \mathsf{DiscrFirst}_i - 1]$. We have $\mathsf{First}_{i-1} = \mathsf{LF}(\mathsf{DiscrFirst}_i) - z$. Thus, the difference $\Delta = \mathsf{First}_{i-1} - \mathsf{AFirst}_{i-1}$ equals to $\mathsf{LF}(\mathsf{DiscrFirst}_i) - z - \mathsf{LF}(\mathsf{DiscrFirst}_i) + RL = RL - z$. Since there are at most $\ell/2 - 1$ occurrences of $c$ in $L[\mathsf{AFirst}_i, \mathsf{DiscrFirst}_i - 1]$ and since $\mathsf{AFirst}_i \leq \mathsf{First}_i \leq \mathsf{DiscrFirst}_i$, $z$ cannot be larger than $RL = \min(\mathsf{DiscrFirst}_i - \mathsf{AFirst}_i, \ell/2 - 1)$. Hence, $0 \leq \Delta < \ell/2$.

**Case 2.** Assume $\mathsf{First}_i > \mathsf{DiscrFirst}_i$. Let $z$ be the number of occurrences of $c$ in $L[\mathsf{DiscrFirst}_i, \mathsf{First}_i - 1]$. We have $\mathsf{First}_{i-1} = \mathsf{LF}(\mathsf{DiscrFirst}_i) + z$. Thus, the difference $\Delta = \mathsf{First}_{i-1} - \mathsf{AFirst}_{i-1}$ equals to $\mathsf{LF}(\mathsf{DiscrFirst}_i) + z - \mathsf{LF}(\mathsf{DiscrFirst}_i) + RL = z + RL$. Since both $z$ and $RL$ are non-negative and $z + RL$ cannot be larger than the number of positions between $\mathsf{AFirst}_i$ and $\mathsf{First}_i - 1$, it follows $0 \leq \Delta < \ell/2$. □

We easily obtain the following Theorem by combining Lemma 1 with the proof of correctness of Backward Search (Lemma 3.1 in [11]).

**Theorem 7** *For any pattern $P[1,p]$ that occurs $\mathsf{Count}(P)$ times in text $T$, Algorithm 4 returns in $O(p)$ steps as result a value $\mathsf{Count}{\approx_\ell}(P)$ which is in the range $[\mathsf{Count}(P), \mathsf{Count}(P) + \ell - 1]$.*

Notice that, if $[\mathsf{First}, \mathsf{Last}]$ is the range of indexes corresponding to the consecutive suffixes that are prefixed by $P$, then the algorithm identifies a range $[\mathsf{AFirst}, \mathsf{ALast}]$ such that $\mathsf{First} - \ell/2 < \mathsf{AFirst} \leq \mathsf{First}$ and $\mathsf{Last} \leq \mathsf{ALast} < \mathsf{Last} + \ell/2$.

It remains to show how to represent the sets of discriminant positions $\mathcal{D}_c$ to support predecessor and successor queries on them. We represent all of these sets by means of two different objects. We conceptually divide the string $L = \mathsf{Bwt}(T)$ into $\lceil 2n/\ell \rceil$ blocks of equal length and for each of them we create the characteristic set $B_i$, such that $B_i$ contains $c$ iff there exists a position in $\mathcal{D}_c$ belonging to block $i$. Note that since each block has length $\lfloor l/2 \rfloor$, the construction procedure for $\mathcal{D}_c$ guarantees that there can only be one discriminant position per symbol in any block. Considering sets $B_i$ as strings (with arbitrary order), we compute the string $B = B_0\#B_1\#\ldots B_{2n/\ell}\#$ where $\#$ is a symbol outside $\Sigma$ and augment it with Rank and Select data structures (see Theorem 1). Let $r$ be the total number of discriminant positions. We also create an array $V$ of $r$ cells, designed as follows. Let $x$ be a discriminant position and assume that it appears as the $j$th one in $B$, then $V[j] = x \bmod \ell/2$. The following lemma states that a constant number of Rank and Select queries on $B$ and $V$ suffices for computing $\mathsf{Pred}(x,\mathcal{D}_c)$ and $\mathsf{Succ}(x,\mathcal{D}_c)$.

**Lemma 2** $\mathsf{Pred}(x,\mathcal{D}_c)$ *and* $\mathsf{Succ}(x,\mathcal{D}_c)$ *can be computed with a constant number of* Rank *and* Select *queries on* $B$ *and* $V$*.*

*Proof* We show only how to support $\mathsf{Pred}(x,\mathcal{D}_c)$ since $\mathsf{Succ}(x,\mathcal{D}_c)$ is similar. Let $p = \mathsf{Rank}_c(B, \mathsf{Select}_\#(B, \lfloor 2x/\ell \rfloor))$, denoting the number of blocks containing a discriminant position of $c$ before the one addressed by $\lfloor 2x/\ell \rfloor$. Let $q = \mathsf{Select}_c(B,p) - \lfloor 2x/\ell \rfloor$ be the index of the discriminant position preceding $x$ (the subtraction removes the $\#$ spurious symbols). Then, by computing $g = \mathsf{Rank}_\#(B, \mathsf{Select}_c(B,p))$ we find the block preceding (or including) $\lfloor 2x/\ell \rfloor$ that has a discriminant position for $c$. Also, $V[q]$ contains the offset, within that block, of the discriminant position. Such position can be either in a block preceding $\lfloor 2x/\ell \rfloor$ or in the same block. In the former case, $\mathsf{Pred}(x,\mathcal{D}_c) = \lfloor 2x/\ell \rfloor g + V[q]$. In latter case we have an additional step to make, as we have so far retrieved a position that just belongs to the same block of $x$ but could be greater than $x$. If that happens, we decrease $p$ by 1 and repeat all the calculations. Note that, since the first occurrence of $c$ is also a discriminant, this procedure can never fail. $\square$

Once we have computed the correct discriminant positions, Algorithm 4 requires to compute an LF-step from them (lines 7 and 9). The following Lemma states that this task is simple.

**Fact 1** *For any symbol $c$, given any discriminant position $d$ in $\mathcal{D}_c$ but the largest one, we have that* $\mathsf{LF}(d) = C[c] + (i-1)\cdot \ell/2 + 1$ *where $i$ is such that $\mathcal{D}_c$'s $i$th element in left-to-right position is $d$. For the largest discriminant position $d$ in $\mathcal{D}_c$ we have* $\mathsf{LF}(d) = C[c+1]$.

It follows immediately that while performing the calculations of Lemma 2 we can also compute the LF mapping of the discriminant position retrieved.

To conclude the proof of Theorem 6 it remains to bound the time and the space complexities of our solution. Since the query algorithm requires $O(p)$ applications of Lemma 2, the claim on the time complexity easily follows. The space complexity is given by three elements. The array $C$, containing counters for each symbol, requires $O(\sigma \log n)$ bits. The number of discriminant positions is easily seen to be at most $2n/\ell$ in total, hence the array $V$ requires at most $O(n/\ell)$ cells of $O(\log \ell)$ bits each. Finally, the string $B$ requires one symbol per block plus one symbol per discriminant position, accounting for $O(\frac{n \log \sigma}{\ell})$ bits in total. Thus, the overall space occupancy of the data structure is $O(\frac{n \log(\sigma\ell)}{\ell} + \sigma \log n)$ bits. The first term dominates as long as $\sigma = O(n/\ell)$, and, in this case, the space bound of Theorem 6 holds. In order to match this bound also for larger alphabet sizes (i.e., $\sigma = \omega(n/\ell)$) we use a simple pre-filter with the aim of reducing the alphabet size to $O(n/\ell)$ without altering the approximation guarantees. Observe that the number of symbols that occur at least $\ell$ times in $T$ are at most $n/\ell$. Let $\Sigma'$ be the set of these symbols. We build a minimal perfect hash function $M$ to map each symbol in $\Sigma'$ to a unique value in $[|\Sigma'|]$. The minimal perfect hash function requires $O(|\Sigma'|)$ bits of space, and, given a symbol $c$, $M(c)$ can be evaluated in constant time [17]. We also store an array of size $|\Sigma'|$ that stores the symbol $c \in \Sigma'$ in position $M(c)$. This array requires $O(|\Sigma'| \log \sigma) = O(\frac{n \log \sigma}{\ell})$ bits. In this way, given a symbol $c$ in $\Sigma$, we can check if $c$ belongs to $\Sigma'$ and, in case, remap it to $M(c)$ in constant time. We now construct the text $T'$ of length $n$ drawn from an alphabet of size $|\Sigma'| + 1$. The $i$th symbol of $T'[i]$ is equal to $M(T[i])$ if $T[i] \in \Sigma'$, or to the special symbol \$ otherwise. We build the index described in this section over the text $T'$, which, thanks to the alphabet remapping, requires $O(\frac{n \log(\sigma\ell)}{\ell})$ bits of space. At query time, given a pattern $P$, we remap its symbols by using the minimal perfect hash function in $O(p)$ time. If at least one of its symbols does not belong to $\Sigma'$, then the pattern cannot occur more than $\ell - 1$ times in $T$ and, thus, we can safely report $\ell - 1$ as its number of occurrences. Otherwise, we search the remapped pattern with the index for $T'$ so that its number of occurrences is correctly estimated.

## 5 Lower-side error solution

Let $\mathsf{PST}_\ell(T)$ be the pruned suffix tree as discussed in Subsection 2.5, and let $m$ be the number of its nodes. Recall that $\mathsf{PST}_\ell(T)$ is obtained from the suffix tree of $T$ by removing all the nodes with less than $\ell$ leaves in their subtrees, and hence constitutes a solution to the lower-sided error problem: when $\mathsf{Count}(P) \geq \ell$, the answer is correct, otherwise the value $\ell - 1$ is returned. Thus, compared with the solution of previous section, it has the great advantage of being perfectly correct if the pattern appears frequently enough, but it is extremely space inefficient. Our objective in this section is to present a compact version of $\mathsf{PST}_\ell(T)$, by means of proving the following theorem.

**Theorem 8** *Given $T[1,n]$ drawn from an alphabet $\Sigma = [\sigma]$, $\sigma \leq n$, and given an error threshold $\ell$, there exists a representation of $\mathsf{PST}_\ell(T)$ using $O(m \log(\sigma\ell))$ bits that can answer to $\mathsf{Count}{\geq_\ell}(P[1,p])$ in $O(p(1 + \log \frac{\log \sigma}{\log w}))$ time, where $m$ is the number of nodes of $\mathsf{PST}_\ell(T)$.*

To appreciate Theorem 8, consider that the original $\mathsf{PST}_\ell(T)$ representation requires, apart from node pointers, to store labels together with their lengths, for a total of $O(m \log n + g \log \sigma)$ bits. The predominant space complexity is given by the edge labels, since it can reach $n \log \sigma$ bits even when $m$ is small. Therefore, our objective is to build an alternative search algorithm that does not require all the labels to be stored.

## 5.1 Computing counts

As a crucial part of our explanation, we will refer to nodes using their preorder traversal times. The branching symbol of a child of a node $u$ is the first symbol of child's edge label. During the visit we are careful to descend into children in ascending lexicographical order over their branching symbols. Therefore, $u < v$ if and only if $u$ is either an ancestor of $v$ or their corresponding path labels have the first mismatching symbols, say in position $k$, such that $\mathsf{pathlabel}(u)[k] < \mathsf{pathlabel}(v)[k]$.

We begin by explaining how to store and access a basic information that our algorithm must recover: Given any node $u \in \mathsf{PST}_\ell(T)$, we would like to compute $C(u)$, the number of occurrences of $\mathsf{pathlabel}(u)$ as a substring in $T$.[4] A straightforward storage of such data would require $m \log n$ bits for a tree of $m$ nodes. We can obtain a more space efficient representation and still able to compute $C(u)$ in $O(1)$ time. Our approach is based on the following simple observation.

**Observation 1** *Let $u$ be a node in $\mathsf{PST}_\ell(T)$ and let $v_1, v_2, \ldots, v_k$ be the children of $u$ in the suffix tree of $T$ that have been pruned away. We have $g(u) = \sum_{1 \le i \le k} C(v_i) < \sigma\ell$.*

*Proof* For any $i \le k$, since $v_i$'s subtree has been pruned away, it holds $C(v_i) < \ell$. Since node $u$ has at most $\sigma$ children, the observation follows. □

Note that Observation 1 applies in a stronger form to leaves of the suffix tree where $C(x) = g(x)$, for any leaf $x$. We refer to the $g(\cdot)$ values as *correction terms*. For an example refer to Figure 5. It is easy to see that to obtain $C(v)$ it suffices to sum all the correction terms of all the descendants of $v$ in $\mathsf{PST}_\ell(T)$. Precisely, it suffices that, at index construction time, we build[5] the binary string $G = 0^{g(0)}10^{g(1)}1 \cdots 0^{g(m-1)}1$ together with support for binary $\mathsf{Select}$ queries.

**Lemma 3** *Let $v \in \mathsf{PST}_\ell(T)$ and let $z$ be the identifier of the rightmost leaf in the subtree rooted at $v$. Define $\mathsf{CNT}(v,z) = \mathsf{Select}_1(G,z) - z - \mathsf{Select}_1(G,v) + v$. Then $C(v) = \mathsf{CNT}(v,z)$.*

*Proof* By the numbering scheme, it follows that a consecutive range of $G$ contains the values of all the nodes in the subtree of $v$. $\mathsf{Select}_1(G,x) - x$ is equivalent to $\mathsf{Rank}_0(G, \mathsf{Select}_1(G,x))$, i.e., it sums up all correction terms in nodes before $x$ in the numbering scheme. Computing the two prefix sums and subtracting gives the value of $C(v)$. □

[4] Notice that $C(u)$ is the number of leaves in the subtree of $u$ in the *original* suffix tree.

[5] We use the notation $0^x$ to denote the binary value 0 repeated $x$ times.

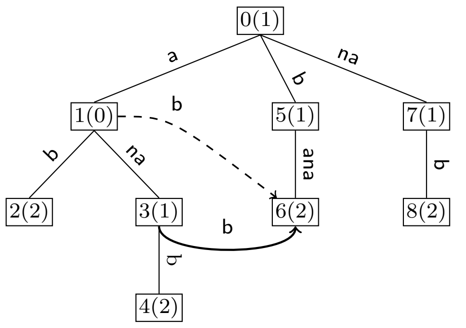


Fig. 5: The pruned suffix tree of the text banabananab\$ with threshold 2. Each node contains its preorder traversal id and, in parentheses, its correction factor. The solid arrow denotes an inverse suffix link for b, the dashed arrow a virtual one.

**Lemma 4** *Let $m$ be the number of nodes in $\mathsf{PST}_\ell(T)$, then $G$ can be stored using at most $m\log(\sigma\ell)+O(m)$ bits. The computation of $\mathsf{CNT}(v,z)$ requires $O(1)$ time.*

*Proof* Each correction factor is at most $\sigma\ell$, hence the number of 0s in $G$ is at most $m\sigma\ell$. The number of 1s in $G$ is $m$. The thesis follows by storing $G$ with the binary Elias-Fano's data structure of Theorem 2. □

## 5.2 Finding the correct node

Our solution relies on the concepts of suffix links and inverse suffix links in a suffix tree. For each node $u$ of $\mathsf{PST}_\ell(T)$, the *suffix link* $\mathsf{SL}(u)$ is $v$ iff the string $\mathsf{pathlabel}(v)$ is obtained from $\mathsf{pathlabel}(u)$ by removing the its first symbol. The *inverse suffix link* (also referred to as Weiner link) of $v$ for some symbol $c$, denoted $\mathsf{ISL}(v,c)$, is $u$ iff $v=\mathsf{SL}(u)$ and the *link symbol* is $c$ (i.e., $c\cdot\mathsf{pathlabel}(v)=\mathsf{pathlabel}(u)$). We say that $v$ possesses an inverse suffix link for $c$ if $\mathsf{ISL}(v,c)$ is defined. We also refer to the lowest common ancestor of two nodes $u$ and $v$ as $\mathsf{LCA}(u,v)$. An inverse suffix link $\mathsf{ISL}(v,c)=u$ exists only if $c\cdot\mathsf{pathlabel}(v)=\mathsf{pathlabel}(u)$, however many search algorithms require also *virtual* inverse suffix links to be available. We say a node $w$ has a virtual inverse suffix link for symbol $c$ (denoted $\mathsf{VISL}(w,c)$) if and only if at least one of its descendants (including $w$) has an inverse suffix link for $c$. The value of $\mathsf{VISL}(w,c)$ is equal to $\mathsf{ISL}(v,c)$, where $v$ is the highest descendant of $w$ having an inverse suffix link for $c$.[6] As we will see in Lemma 7, it is guaranteed that this highest descendant is unique and, thus, this definition is always well formed. The intuitive meaning of virtual suffix links is the following: $\mathsf{VISL}(w,c)$ links node $w$ to the highest node $w'$ in the tree whose path label is prefixed by $c\cdot\mathsf{pathlabel}(w)$. An example is illustrated in Figure 5.

[6] Notice that $w$ and $v$ coincide whenever $w$ has an inverse suffix link for $c$.

Our interest in virtual inverse suffix links is motivated by an alternative interpretation of the classic backward search. When the backward search is performed, the algorithm virtually starts at the root of the suffix tree, and then traverses (*virtual*) inverse suffix links using the pattern to induce the link symbols, prefixing a symbol at the time to the suffix found so far. The use of virtual inverse suffix links is necessary to accommodate situations in which the pattern $P$ exists but only an extension $P \cdot \alpha$ of it appears as a node in the suffix tree. Note that the algorithm can run directly on the suffix tree if one has access to virtual inverse suffix links, and such property can be directly extended to pruned suffix trees. Storing virtual inverse suffix links explicitly is prohibitive since there can be up to $\sigma$ of them outgoing from a single node, therefore we plan to store real inverse suffix links and provide a fast search procedure to evaluate the $\mathsf{VISL}$ function.

In the remaining part of this section we will show a few properties of (virtual) suffix links that allow us to store/access them efficiently and to derive a proof of correctness of the search algorithm sketched above.

The following two lemmas state that inverse suffix links preserve the relative order between nodes.

**Lemma 5** *Let* $w, z$ *be nodes in* $\mathsf{PST}_\ell(T)$ *such that* $\mathsf{ISL}(w,c) = w'$ *and* $\mathsf{ISL}(z,c) = z'$. *Let* $u = \mathsf{LCA}(w,z)$ *and* $u' = \mathsf{LCA}(w',z')$. *Then,* $\mathsf{ISL}(u,c) = u'$.

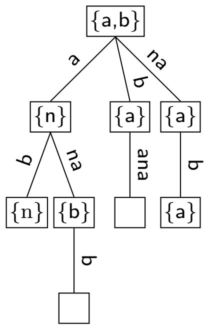
*Proof* If $w$ is a descendant of $z$ or viceversa, the lemma is proved. Hence, we assume $u \neq w$ and $u \neq z$. Let $\alpha = \mathsf{pathlabel}(u)$. Since $u$ is a common ancestor of $w$ and $z$, it holds $\mathsf{pathlabel}(w) = \alpha \cdot \beta$ and $\mathsf{pathlabel}(z) = \alpha \cdot \zeta$ for some non-empty strings $\beta$ and $\zeta$. By definition of inverse suffix link, we have that $\mathsf{pathlabel}(w') = c \cdot \alpha \cdot \beta$ and $\mathsf{pathlabel}(z) = c \cdot \alpha \cdot \zeta$. Since $w$ and $z$ do not share the same path below $u$, the first symbols of $\beta$ and $\zeta$ must differ. This implies the existence of a node $v$ whose path label is $\mathsf{pathlabel}(v) = c \cdot \alpha$ which is the lowest common ancestor between $w'$ and $z'$. Again by definition of inverse suffix link, it follows that $\mathsf{ISL}(u,c) = u' = v$.[7] □

**Lemma 6** *Given any pair of nodes* $u$ *and* $v$ *with* $u < v$ *such that both have an inverse suffix link for symbol* $c$, *it holds* $\mathsf{ISL}(u,c) < \mathsf{ISL}(v,c)$.

*Proof* Since $u < v$, we have that $\mathsf{pathlabel}(u)$ is lexicographically smaller than $\mathsf{pathlabel}(v)$. Thus, obviously $c \cdot \mathsf{pathlabel}(u)$ is lexicographically smaller than $c \cdot \mathsf{pathlabel}(v)$. Since $c \cdot \mathsf{pathlabel}(u)$ is the path label of $u' = \mathsf{ISL}(u,c)$ and $c \cdot \mathsf{pathlabel}(v)$ is the path label of $v' = \mathsf{ISL}(v,c)$, $u'$ precedes $v'$ in the preorder traversal of $\mathsf{PST}_\ell(T)$. □

Computing the virtual inverse suffix link of node $u$ for symbol $c$ requires to identify the highest descendant of $u$ (including $u$) having an inverse suffix link for $c$. If such a node does not exist we conclude that the virtual inverse suffix link is undefined. The following lemma states that such node, say $v$, must be unique, meaning that if there exists another descendant of $u$ having an inverse suffix link for $c$, then this node must also be a descendant of $v$.

[7] We notice that this lemma is also observed by Russo *et al.* [24] (Lemma 5.1).



$S$ =ab#n#n#b##a##a#a#

Fig. 6: The same PST of Figure 5, with information associated with Theorem 9. Each node is given the set of symbols for which an inverse suffix link is defined. The string $S$ contains the separated encoding of these symbols collected by visiting the tree in preorder.

**Lemma 7** *For any node $u$ in $\mathsf{PST}_\ell(T)$, the highest descendant of $u$ (including $u$) having an inverse suffix link for a symbol $c$, if existing, is unique.*

*Proof* Pick any pair of nodes that descend from $u$ having an inverse suffix link for the symbol $c$. By Lemma 5 their common ancestor must also have an inverse suffix link for $c$. Thus, there must exist a unique node that is a common ancestor of all of these nodes. $\square$

In our solution we conceptually associate each node $u$ in $\mathsf{PST}_\ell(T)$ with the set of symbols $\mathcal{E}_u$ for which $u$ has an inverse suffix link. We represent each set with a string $\mathrm{Enc}(\mathcal{E}_u)$ built by concatenating the symbols in $\mathcal{E}_u$ in any order and ending with a special symbol # not in $\Sigma$. We then build a string $S$ as $\mathrm{Enc}(\mathcal{E}_0)\mathrm{Enc}(\mathcal{E}_1)\cdots\mathrm{Enc}(\mathcal{E}_{m-1})$ so that the encodings follow the preorder traversal of the tree[8]. We also define the array $C[1,\sigma]$ whose entry $C[c]$ stores the number of nodes of $\mathsf{PST}_\ell(T)$ whose pathlabels start with a symbol lexicographically smaller than $c$. The next theorem proves that string $S$ together with Rank and Select capabilities is sufficient to compute VISL. This is crucial to prove that our data structure works, proving virtual inverse suffix links can be recreated from real ones.

**Theorem 9** *Let $u \in \mathsf{PST}_\ell(T)$ and let $z$ be the rightmost leaf descending from $u$. For any symbol $c \in \Sigma$, let $c_u = \mathsf{Rank}_c(S, \mathsf{Select}_\#(S, u-1))$ and, similarly, let $c_z = \mathsf{Rank}_c(S, \mathsf{Select}_\#(S, z))$. Then (a) if $c_u = c_z$, $\mathsf{VISL}(u,c)$ is undefined. Otherwise, (b) $\mathsf{VISL}(u,c) = C[c] + c_u + 1$ and (c) $C[c] + c_z$ is the rightmost leaf descending from $\mathsf{VISL}(u,c)$.*

[8] A similar method of traversing a suffix tree by means of inverse suffix links encoded in a string has been proposed by Arroyuelo *et al.* [1].

**Algorithm** $\mathsf{Count}_{\geq \ell}(P[1,p])$

```
1. $i = p$, $c = P[p]$, $u_p = C[c]+1$, $z_p = C[c+1]$;
2. while $((u_i \neq z_i)$ and $(i \geq 2))$ do
3.    $c = P[i-1]$;
4.    $u_{i-1} = \mathsf{VISL}(u_i, c) = C[c] + \mathsf{Rank}_c(S, \mathsf{Select}_{\#}(S, u_i)) + 1$;
5.    $z_{i-1} = \mathsf{VISL}(z_i, c) = C[c] + \mathsf{Rank}_c(S, \mathsf{Select}_{\#}(S, z_i))$;
6.    $i = i - 1$;
7. if $(u_i = z_i)$ then return "no occurrences of $P$" else return $\mathsf{CNT}(u_1, z_1)$
```

Fig. 7: The algorithm reports the number of occurrences of a pattern $P[1,p]$ in the Compact Pruned Suffix Tree.

*Proof* Let $\mathcal{A}$ be the set of nodes of $\mathsf{PST}_\ell(T)$ whose path label is lexicographically smaller than the path label of $u$ and let $\mathcal{B}$ be the set of nodes in the subtree of $u$. Let $S(\mathcal{A})$ and $S(\mathcal{B})$ be the concatenations of, respectively, $\mathrm{Enc}(\mathcal{E}_w)$ for $w \in \mathcal{A}$ and $\mathrm{Enc}(\mathcal{E}_w)$ for $w \in \mathcal{B}$. Due to the preorder numbering of nodes, we know that $\mathcal{A} = [0, u-1]$ and $\mathcal{B} = [u, z]$. Thus, $S(\mathcal{A})$ is a prefix of $S$ that ends where $S(\mathcal{B})$ begins. Notice that the operations $\mathsf{Select}_{\#}(S, u-1)$ and $\mathsf{Select}_{\#}(S, z)$ return respectively the ending positions of $S(\mathcal{A})$ and $S(\mathcal{B})$ in $S$. Thus, $c_u$ counts the number of inverse suffix links of nodes in $\mathcal{A}$ while $c_z$ includes also the number of inverse suffix links of nodes in $\mathcal{B}$. Hence, if $c_u = c_z$ no node of $\mathcal{B}$ has an inverse suffix link and, thus, proposition $(a)$ is proved.

By Lemma 6 we know that inverse suffix links map nodes preserving their relative order. Thus, the first node in $\mathcal{B}$ that has an inverse suffix link for $c$ is mapped to node $C[c] + c_u + 1$.[9] By the node numbering, this first node is obviously also the highest one. Thus, proposition (b) is proved.

Proposition (c) is proved by resorting to similar considerations. □

Figure 6 illustrates the whole situation. Exploiting $\mathsf{VISL}$, Algorithm 7 searches for a pattern $P[1,p]$ backwards. The algorithm starts by setting $u_p$ to be $C[P[p]] + 1$. At the $i$th step, we inductively assume that $u_{i+1}$ is known, and that its path label is prefixed by $P[i+1,p]$. Similarly, we keep $z_{i+1}$, the address of the rightmost leaf in $u$'s subtree. Using $u_{i+1}$ and $z_{i+1}$ we can evaluate if $\mathsf{VISL}(u_{i+1}, P[i])$ exists and, in such case, follow it. In the end, we have to access the number of suffixes of $T$ descending from $u_1$. The next theorem formally proves the whole algorithm correctness:

**Theorem 10** *Given any pattern $P[1,p]$, Algorithm 7 retrieves $C(u)$, where $u$ is the highest node of $\mathsf{PST}_\ell(T)$ such that $\mathsf{pathlabel}(u)$ is prefixed by $P$. If such node does not exist, it terminates reporting $-1$.*

*Proof* We start by proving that such node $u$, if any, is found, by induction. It is easy to observe that $C[P[p]] + 1$ is the highest node whose path label is prefixed by the single symbol $P[p]$.

[9] There is a caveat: in case the first node of the subtree of $c$ has an edge label with length greater than 1, then the $+1$ factor must be eliminated, since that same node becomes a destination.

By hypothesis, we assume that $u_{i+1}$ is the highest node in $\mathsf{PST}_\ell(T)$ whose path label is prefixed by $P[i+1,p]$, and we want to prove the same for $u_i = \mathsf{VISL}(u_{i+1}, P[i])$. The fact that $\mathsf{pathlabel}(u_i)$ is prefixed by $P[i,p]$ easily follows by definition of inverse suffix link. We want to prove that $u_i$ is the highest one with this characteristic: by contradiction assume there exists another node $w'$ higher that $u_i = \mathsf{VISL}(u_{i+1}, P[i])$. This implies that there exists a node $w = \mathsf{SL}(w')$, prefixed by $P[i+1,p]$. Also, the virtual inverse suffix link of $u_{i+1}$ is associated with a proper one whose starting node is $z = \mathsf{SL}(u_{i+1})$, which by definition of $\mathsf{VISL}$ is also the highest one in $u_{i+1}$'s subtree. Thus, by Lemma 7, $w$ is a descendant of $z$. Hence, $w > z$ but $\mathsf{ISL}(w',c) < \mathsf{ISL}(z,c)$, contradicting Lemma 6.

Finally, if at some point of the procedure a node $u_{i+1}$ does not have a virtual inverse suffix link, then it is straightforward that the claimed node $u$ does not exist (i.e., $P$ occurs in $T$ less than $\ell$ times). Once $u$ is found, also $z$ is present, hence we resort to Lemma 3 to obtain $C(u) = \mathsf{CNT}(u,z)$. $\square$

In order to conclude the proof of Theorem 8 we are left with proving the time/space complexities of the index. The solution has to store: the $C$ array, holding the count of nodes in $\mathsf{PST}_\ell(T)$ whose path label is prefixed by each of the $\sigma$ symbols; the string $G$, together with binary $\mathsf{Select}$ capabilities, and the string $S$, together with arbitrary alphabet $\mathsf{Rank}$ and $\mathsf{Select}$ capabilities. Let $m$ be the number of nodes in $\mathsf{PST}_\ell(T)$. We know $C$ occupies at most $O(\sigma \log n)$ bits. By Lemma 4, $G$ occupies at most $m\log(\sigma\ell) + O(m)$ bits. String $S$ can be represented in different ways, related to $\sigma$, picking a choice from Theorem 1, but the space is always limited by $m\log\sigma + o(m\log\sigma)$. Hence the total space is $O(\sigma\log n) + m\log(\sigma\ell) + O(m) + O(m\log(\sigma)) = O(m\log(\sigma\ell))$, as claimed[10]. For the time complexity, at each of the $p$ steps, we perform four $\mathsf{Rank}$ and $\mathsf{Select}$ queries on arbitrary alphabets which we cost $O(1+\log\frac{\log\sigma}{\log w})$ time each. The final step on $G$ takes $O(1)$ time, hence the bound follows.

## 6 Experiments

In this section we show an experimental comparison among the solutions presented above and those previously known. We use four different data sets downloaded from `Pizza&Chili` corpus [9] that correspond to four different types of texts: DNA sequences, structured text (XML), natural language and source code.

– `Dna`. This file contains bare DNA sequences without descriptions, separated by `newline`, collected from files available at the Gutenberg Project site. Each of the four DNA bases is coded as an uppercase letter A,G,C,T, and there are a few occurrences of other special symbols.

[10] Notice that we can use the same pre-filter described at the end of Section 4 whenever the term $O(\sigma\log n)$ bits is dominant.

- `Dblp`. This file is in XML format and provides bibliographic information on major computer science journals and proceedings. It was downloaded from the DBLP archive at `dblp.uni-trier.de`.
- `English`. This file contains the first 512 Mbytes of the concatenation of English texts selected from the Gutenberg Project site.
- `Sources`. This file is formed by C/Java source codes obtained by concatenating all the *.c*, *.h*, *.C* and *.java* files of the *linux-2.6.11.6* (`ftp.kernel.org`) and *gcc-4.0.0* (`ftp.gnu.org`) distributions.

Text and alphabet sizes of the above datasets are reported in the first two columns in Figure 8.

| | | | $\ell = 8$ | | |
|---|---|---|---|---|---|
| Dataset | Size (in Mbytes) | $\sigma$ | $n/\ell$ | $\lvert\mathsf{PST}_\ell\rvert$ | $\sum_i \lvert\mathsf{edge}(i)\rvert$ |
| `Dblp` | 275 | 96 | 36,064 | 28,017 | 1,034,016 |
| `Dna` | 292 | 15 | 38,399 | 42,361 | 814,993 |
| `English` | 501 | 225 | 65,764 | 53,600 | 660,957 |
| `Sources` | 194 | 229 | 25,475 | 25,474 | 11,376,730 |

| | $\ell = 64$ | | | $\ell = 256$ | | |
|---|---|---|---|---|---|---|
| Dataset | $n/\ell$ | $\lvert\mathsf{PST}_\ell\rvert$ | $\sum_i \lvert\mathsf{edge}(i)\rvert$ | $n/\ell$ | $\lvert\mathsf{PST}_\ell\rvert$ | $\sum_i \lvert\mathsf{edge}(i)\rvert$ |
| `Dblp` | 4,508 | 3,705 | 103,383 | 1,127 | 941 | 20,200 |
| `Dna` | 4,799 | 5,491 | 102,127 | 1,199 | 1,317 | 19,194 |
| `English` | 8,220 | 6,491 | 64,500 | 2,055 | 1,616 | 14,316 |
| `Sources` | 3,184 | 3,264 | 9,430,627 | 796 | 982 | 8,703,817 |

Fig. 8: Statistics on the datasets. The second column denotes the original text in MBytes. Each subsequent group of three columns describe $\mathsf{PST}_\ell$ information for a choice of $\ell$: expected amount of nodes, $n/\ell$; real amount of nodes in $\mathsf{PST}_\ell(T)$; sum of length of labels in $\mathsf{PST}_\ell(T)$. All numbers are expressed in thousands.

In our experimental evaluation we compare the following solutions.

- `FM-index`. This is an implementation of a compressed full-text index available at the `Pizza&Chili` site [9][11]. Since it is the compressed full-text index that achieves the best compression ratio, it is useful to establish which is the minimum space required by known solutions to answer to counting queries without errors.
- `APPROX`-$\ell$. This is the implementation of the solution presented in Section 4.
- `PST`-$\ell$. This is an implementation of the Pruned Suffix Tree as described by Krishnan *et al.* [19].
- `CPST`-$\ell$. This is the implementation of the Compact Pruned Suffix Tree described in Section 5.

[11] This implementation can be downloaded at the address `http://pizzachili.dcc.uchile.cl/indexes/FM-indexV2`.

Recall that APPROX-$\ell$ reports results affected by an error of at most $\ell$ while PST-$\ell$ and CPST-$\ell$ are always correct whenever the pattern occurs at least $\ell$ times in the indexed text.

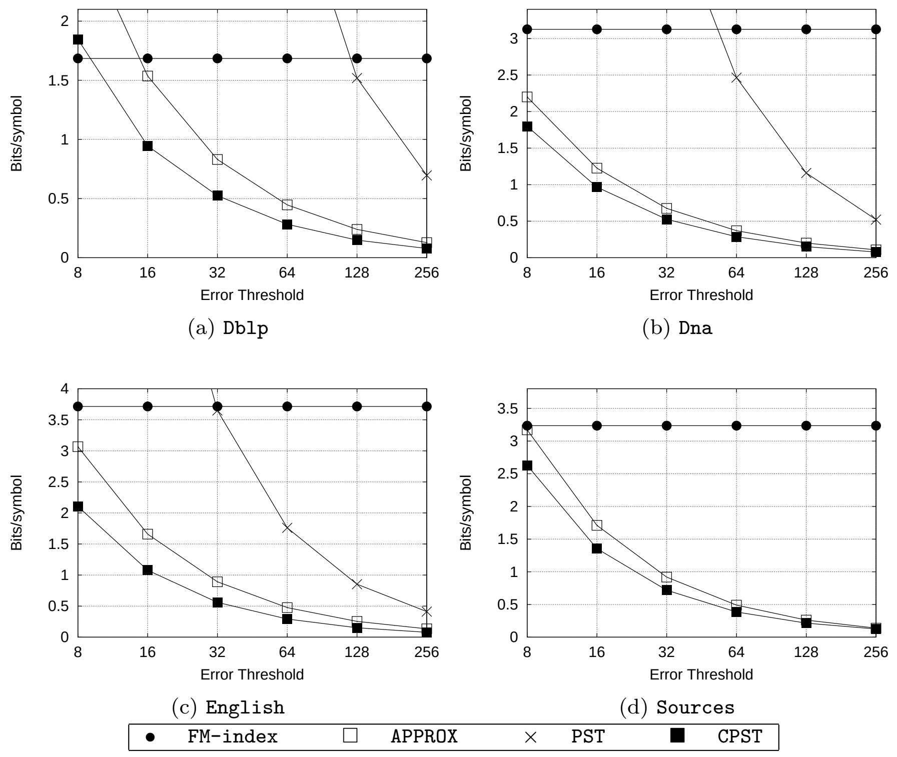


Fig. 9: Occupancies of indexes as a function of the error threshold $\ell$, as bits per symbol of the original text.

The plots in Figure 9 show the space occupancies of the four indexes depending on the chosen threshold $\ell$. We do not plot space occupancies worse than FM-index, since in those cases FM-index is clearly the index to choose. In fact, Figure 9(d) does not contain a plot for PST, since its space performance was always worse than FM-index.

It turns out that in all the texts of our collection the number of nodes in the pruned suffix tree is small (even smaller than $n/\ell$): these statistics are reported in Table 8. This is the reason why our CPST is slightly more space-efficient than APPROX. In practice, the former should be indubitably preferred with respect to the latter: it requires less space and it is always correct for patterns that occur at least $\ell$ times. Even though, the latter remains interesting due to its better theoretical guarantees. In both solutions, by halving the error

| Dataset | Indices | $\|P\| = 6$ | $\|P\| = 8$ |
|---|---|---|---|
| `Dblp` | PST-256 | 10.06 ± 32.372 | 12.43 ± 34.172 |
| | CPST-16 | 0.68 ± 1.456 | 0.86 ± 1.714 |
| `Dna` | PST-256 | 0.47 ± 1.048 | 0.49 ± 2.433 |
| | CPST-32 | 0.47 ± 0.499 | 0.43 ± 0.497 |
| `English` | PST-256 | 7.03 ± 27.757 | 12.45 ± 31.712 |
| | CPST-32 | 0.80 ± 2.391 | 1.40 ± 3.394 |
| `Sources` | PST-11,000 | 816.06 ± 1,646.57 | 564.94 ± 1,418.53 |
| | CPST-8 | 0.70 ± 1.028 | 0.93 ± 1.255 |

| Dataset | Indices | $\|P\| = 10$ | $\|P\| = 12$ | Avg Improvement |
|---|---|---|---|---|
| `Dblp` | PST-256 | 14.20 ± 35.210 | 15.57 ± 36.044 | 19.03× |
| | CPST-16 | 1.00 ± 1.884 | 1.14 ± 2.009 | |
| `Dna` | PST-256 | 4.26 ± 15.732 | 11.09 ± 19.835 | 5.51× |
| | CPST-32 | 0.52 ± 0.904 | 1.77 ± 2.976 | |
| `English` | PST-256 | 13.81 ± 28.897 | 11.43 ± 23.630 | 9.68× |
| | CPST-32 | 2.07 ± 3.803 | 2.45 ± 3.623 | |
| `Sources` | PST-11000 | 400.62 ± 1,229.35 | 313.68 ± 1,120.94 | 792.52× |
| | CPST-8 | 1.13 ± 1.367 | 1.28 ± 1.394 | |

Fig. 10: Comparison of error (difference between number of occurrences and estimate) for `MOL` estimates over different pattern lengths. `PST` and `CPST` parameters are chosen to obtain close index sizes. Tests performed on one Million random patterns appearing in the text. The last column shows the average factor of improvement obtained by using our `CPST` instead of `PST`.

threshold, we obtain indexes that are between 1.75 (`CPST`) and 1.95 (`APPROX`) times smaller. Thus, we can obtain very small indexes by setting relatively small values of $\ell$. As an example, `CPST` with $\ell = 256$ on text `English` requires 5.1 Mbytes of space, which is roughly 100 times smaller than the original text. We observe that both `CPST` and `APPROX` are in general significantly smaller than `FM-index` and remain competitive even for small values of $\ell$. As an example, `FM-index` requires 232.5 Mbytes on `English`, which is roughly 45 times larger than `CPST`−256.

As far as `PST` is concerned, it is always much worse than `CPST` and `APPROX`. As expected, its space inefficiencies are due to the need of storing edge labels since their amount grows rapidly as $\ell$ decreases (see Table 8). Moreover, this quantity depends on the indexed text, while the number of nodes is more stable. Thus, the performance of `PST` is erratic: worse than `CPST` by a factor 6 on `English`, which becomes 60 on `Sources`. It is remarkable that on `sources` we have to increase `PST`'s error threshold up to 11,000 to achieve a space occupancy close to our `CPST` with $\ell = 8$ .

For what concerns applications, we use our best index, i.e., `CPST`, together with one estimation algorithm: `MOL`, briefly explained in Section 2.6. The algorithm is oblivious to the underlying data structure as long as a lower-sided error one is used. We performed a comparison between `MO`, `MOL` and `KVI` [18,19] and found out that `MOL` delivered the best estimates. We also considered `MOC` and MOLC, but for some of our data sets the creation of the constraint network was prohibitive in terms of memory. Finally, we tried to compare with CRT [6]; however, we lacked the original implementation and a significative training set for our data sets. Hence, we discarded the algorithm from our comparison.

Figure 10 shows the average error of the estimates obtained with MOL on our collection by using either CPST or PST as the base data structure. For each set we identified two pairs of thresholds such that our CPST and PST have roughly the same space occupancy. For each text, we searched for 4 Million patterns of different lengths that we randomly extracted from the text. Thus, this figure depicts the significant boost in accuracy that one can achieve by replacing PST with our solution. As an example, consider the case of Sources where the threshold of PST is considerably high due to its uncontrollable space occupancy. In this case the factor of improvement that derives by using our solution is more than 790. The improvements for the other texts are less impressive but still considerable.

## 7 Conclusion and future work

We presented two different solutions to the problem of substring occurrence estimation. Our first solution is a space-optimal data structure when the index is allowed to have a uniform error on the reported number of occurrences. Our second solution can be seen as a very succinct version of the classical Pruned Suffix Tree for the harder problem of having one-sided errors. It guarantees better space complexities with respect to the pruned suffix tree both in theory and in practice. It is not clear if the latter solution is space-optimal or not, thus, proving a lower bound for the latter problem would provide greater insight into the problem.

As a second open problem, we note that the entire article is forced to deal with additive errors due to the space lower bound. A natural question is: is there a way to relax the model, in order to circumvent the multiplicative lower bound (Theorem 5)? For example, what if we allow non-existing substrings to have an arbitrary estimation error, forcing all others with a multiplicative bound?

## References

1. Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1):54–101, 2012.
2. Jérémy Barbay, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proceedings of the 21st International Symposium on Algorithms and Computation (ISAAC)*, pages 315–326, 2010.
3. Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Fast prefix search in little space, with applications. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA)*, pages 427–438, 2010.
4. Djamal Belazzougui and Gonzalo Navarro. New lower and upper bounds for representing sequences. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA)*, pages 181–192, 2012.

5. Michael Burrows and David Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
6. Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Proceedings of the 20th International Conference on Data Engineering (ICDE)*, pages 227–, 2004.
7. Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.
8. Robert M. Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass.*, 1971.
9. Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.
10. Paolo Ferragina and Roberto Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46:236–280, March 1999.
11. Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
12. Paolo Ferragina and Rossano Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):10, 2010.
13. Paolo Ferragina and Rossano Venturini. Compressed cache-oblivious String B-tree. In *Proceedings of 21th Annual European Symposium on Algorithms (ESA)*, pages 469–480, 2013.
14. Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.
15. Roberto Grossi and Jeffrey S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
16. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
17. Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 317–326, 2001.
18. H.V. Jagadish, Raymond T. Ng, and Divesh Srivastava. Substring selectivity estimation. In *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of database systems (PODS)*, pages 249–260, 1999.
19. P. Krishnan, Jeffrey S. Vitter, and Balakrishna R. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 282–293, 1996.
20. Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
21. Donald R. Morrison. PATRICIA - practical algorithm to retrieve coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
22. Gonzalo Navarro and Veli Mäkinen. Compressed full text indexes. *ACM Computing Surveys*, 39(1), 2007.
23. Alessio Orlandi and Rossano Venturini. Space-efficient substring occurrence estimation. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 95–106, 2011.
24. Luís M. S. Russo, Gonzalo Navarro, and Arlindo Oliveira. Fully-compressed suffix trees. *ACM Transactions on Algorithms*, 7(4), 2011.
25. Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1999.