Space-efficient Substring Occurrence Estimation

Alessio Orlandi
Dipartimento di Informatica
University of Pisa, Italy
aorlandi@di.unipi.it

Rossano Venturini
ISTI-CNR
Pisa, Italy
venturini@isti.cnr.it

ABSTRACT

We study the problem of estimating the number of occurrences of substrings in textual data: A text T on some alphabet Σ of size σ 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 [15, 14, 5]. 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 [8, 11, 6, 17] providing an optimal data structure that requires $\Theta(\frac{|T| \log \sigma}{l})$ bits where $l \geq 1$ is the additive error on any answer. 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 sustained by experiments showing the practical impact of our data structures.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Arrays, Tables; E.4 [**Coding and Information Theory**]: Data compaction and compression; E.5 [**Files**]: Sorting/searching; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical algorithms and Problems—*Pattern matching*; H.3 [**Information Storage and Retrieval**]: Content Analysis and Indexing, Information Storage, Information Search and Retrieval.

General Terms

Algorithms, Design, Theory.

Keywords

Compressed Full-Text indexes, Pattern Matching, Full-text Indexing, Data structures

1. INTRODUCTION

A large fraction of the data we process every day consists of a sequence of symbols from an alphabet, i.e., a text. 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 not uncommon to have massive data sets at hand, on which operations must be performed. Thinking about text, the basic class of operations are simple pattern matching queries (or 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* [8, 11, 6, 17, 10]. 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 exists a bound on the compression ratio they can achieve. Such a limit can be surpassed by allowing pattern matching operations to have approximated results. This is a realistic scenario, as with massive amounts of data and answers that provide millions of strings, a small absolute error is clearly tolerable in many situations. In this paper we follow such idea by studying the problem called **Substring Occurrence Estimation**:

Given a text $T[1, n]$ drawn from an alphabet Σ of size σ and fixed any error parameter l, 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. The index is allowed to err by at most l: precisely, the reported number of occurrences of P is in the range $[\mathsf{Count}(P), \mathsf{Count}(P) + l - 1]$ where $\mathsf{Count}(P)$ is the actual number of occurrences of P in T. In the following we will refer this operation, which we say has *uniform error range* with $\mathsf{Count}{\approx}_l(P)$. We also consider a stronger version of the problem denoted $\mathsf{Count}{\geq}_l(P)$, namely having *lower-sided error range*, where $\mathsf{Count}{\geq}_l(P) = \mathsf{Count}(P)$ whenever $\mathsf{Count}(P) \geq l$, and $\mathsf{Count}{\geq}_l(P) \in [0, l - 1]$ otherwise.

A relative of additive error is multiplicative error, i.e. when the estimation lays in $[\mathsf{Count}(P), (1 + \varepsilon)\mathsf{Count}(P)]$ for some fixed $\varepsilon > 0$. In theory, such an error could provide better estimates for low frequency patterns. 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 formulas). This turns out to be the hard part of estimation. In fact, we are able to prove (Theorem 4) 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 [15, 14, 5] try to reduce the error when the data structure is not able to guarantee a correct answer, i.e., $\mathsf{Count}(P) < l$. 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. To date, most 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 discuss, we will focus on occurrence estimation on whole texts only. Nonetheless, the techniques immediately apply to collections of strings (i.e., rows in a db 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 $T(R)$.

The main data structure for occurrence estimation, and the one used in [15, 14], is the *pruned suffix tree* $\mathcal{PST}_l(T)$. Here, we briefly review it and defer a full explanation of related work to Section 7. For a fixed error $l \geq 1$, the $\mathcal{PST}_l(T)$ is obtained from the suffix tree [12] of T by pruning away all nodes of suffixes that appear less than l times in T. It is immediate to see that the resulting data structure has, indeed, lower-sided error. However, the space occupancy of $\mathcal{PST}_l$ 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. The number of nodes in the pruned tree could raise to $O(n - l)$ and could slowly decrease as the error l 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 l, the space required to store its pruned suffix tree is at least $O((n - l) \log n)$ 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. It goes without saying that the number of symbols we need to store can exceed the length of the text itself. One could resort to techniques like blind search over compacted tries [7] to remove the need of storing full labels for the edges. However, it would incur in an uncontrollable error when the pattern is not in the $\mathcal{PST}_l$, since solutions based on compacted tries require the original text to perform membership queries. Thus, the space occupancy of the pruned suffix tree may be not sublinear w.r.t. the text. Moreover, the lower bound of Theorem 3 formally proves that the space complexity for an index with threshold l is $\Omega(n \log(\sigma)/l)$ bits, hence stating that a pruned suffix tree is highly non-optimal.

To provide solutions with smaller footprint, one can resort to compressed full-text indexes [8, 11, 6, 17], which are well known in the field of succinct data structures. They deliver a framework to keep a copy of text T compressed together with auxiliary information for efficient (i.e., without decompressing the whole T) 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. Our objective is to heavily reduce the space of compressed text indexes as l increases.

We provide two different solutions: one in the uniform error model and one in the lower-sided error model. Section 4 illustrates the former and shows how to build an index (called $\mathcal{APX}_l$) that requires $O(n \log(\sigma l)/l)$ 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 3) that such index is space-optimal up to constant factors for sufficiently small l (namely, $\log l = O(\log \sigma)$).

We also provide a data structure ($\mathcal{CPST}_l$) for the lower-sided error problem (Section 5) that presents a space bound of $O(m \log(\sigma l))$ where m is the number of nodes in the $\mathcal{PST}_l(T)$. Hence, our $\mathcal{CPST}_l$ does not require to store the labels (the $g \log \sigma$ factor), which account for most of the space in practice. Such data structure outperforms our previous solution only when $m = O(n/l)$; surprisingly, many real data sets exhibit the latter property[1]. Both the $\mathcal{APX}_l$ and $\mathcal{CPST}_l$ data structures heavily rely on the Burrows-Wheeler Transform (BWT), which proves to be an effective tool to tackle the problem. As part of our contribution, we prove how the pruning of Suffix Trees

In Section 6 we support our claims with tests on real data sets. We show the improvement in space occupancy of both $\mathcal{APX}_l$ and $\mathcal{CPST}_l$, both ranging from 5 to 60 w.r.t. to $\mathcal{PST}_l$, 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 $l = 256$ to obtain an index of 5.1 MB (roughly, 1%). We also confirm that m and n/l are close most of the times. In such sense we also note that the main component in $\mathcal{PST}_l$'s space is given by the labels, hence guaranteeing to our $\mathcal{CPST}_l$ a clear advantage over $\mathcal{PST}_l$.

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 [14]). Given two thresholds yielding similar space occupancies between $\mathcal{PST}_l$ and $\mathcal{CPST}_l$, we exhibit an improvement factor ranging from 5 to 790. Combining `MOL` and our `CPST` with reasonably small l, 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.

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

2. NOTATION

Let $T[1,n]$ be a string drawn from the alphabet Σ of size σ.[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: $H_0(T) = (1/n) \sum_{c \in \Sigma} n_c \, \log(n/n_c)$.

Note that $|T|H_0(T)$ provides an information-theoretic lower bound to the output size of any compressor that encodes each symbol of T with a fixed code [19]. 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 $T =$ `abracadabra` and $w =$ `br`, we have $w_T =$ `aa` since the two occurrences of `br` in T are followed by the symbol `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 $k \geq 0$. As usual in data compression [16], we will adopt $nH_k(T)$ as 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.

Both our solutions rely on basic data structures that can answer `rank` and `select` queries. In the binary version, let B be a bit vector of length u, having m bits set to $\mathbf{1}$. Here, $\texttt{rank}_b(B, x)$ for $b \in \{0, 1\}$ counts the number of occurrences of bit b in the prefix $B[0..x-1]$. $\texttt{select}_b(B, x)$ for $b \in \{0, 1\}$ returns the position of the xth occurrence of bit b in B, or -1. Among all available solutions, we employ Elias Fano sequences (also known as SDarrays) ([18, Section 6]):

Theorem 1. *There exists a data structure encoding a bitvector B of length u with m bits set to $\mathbf{1}$ in $m \log(u/m) + O(m)$ bits, supporting $\texttt{select}_1$ in $O(1)$ time and $\texttt{select}_0$, $\texttt{rank}_1$ and $\texttt{rank}_0$ in $O(\log(\min\{u/m, m\}))$ time.*

The idea of `rank` and `select` can be extended from binary to arbitrary alphabets in the natural way. The best solutions to date have been presented in [1, 2, 9]:

Theorem 2. *Given a text $T[1,n]$ drawn from an alphabet of size σ there exists data structures storing T and supporting `rank` and `select` over it with the following trade-offs:*

Ref.	*space (bits)*	`rank`/ `select` *(time)*	σ
[9]	$nH_0(T) + o(n)$	$O(1)$	$\log^{O(1)} n$
[9]	$nH_0(T) + o(n) \cdot \log \sigma$	$O(1 + \frac{\log \sigma}{\log \log n})$	$o(n)$
[2]	$nH_0(T) + n \cdot o(\log \sigma)$	$O(\log \log \sigma)$ / $O(1)$	$O(n)$
[1]	$(n + o(n))H_0(T) + o(n)$	$O(\log \log \sigma)$ / $O(1)$	$O(n)$

3. LOWER BOUNDS

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

Theorem 3. *For a fixed additive error $l \geq 1$, an index built on a text $T[1,n]$ drawn from an alphabet Σ of size σ that approximates the number of occurrences of any pattern P in T within l must use $\Omega(n \log(\sigma)/l)$ bits of space.*

Proof. Assume that there exists an index answering any approximate counting query within an additive error l by requiring $o(n \log(\sigma)/l)$ bits of space. Given any text $T[1,n]$, we derive a new text $T'[1, (l+1)(n+1)]$ that is formed by repeating the string $T\$$ for $l+1$ times, where \$ is a symbol that does not belong to Σ. Then, we build the index on T' that requires $o((l+1)(n+1)\log(\sigma+1)/l) = o(n \log \sigma)$ bits. We observe that we can recover the original text T by means of this index: we search all possible strings of length n drawn from Σ followed by a \$, the only one for which the index answers with a value greater than l is T. A random text has entropy $\log(\sigma^n) - O(1) = n \log \sigma - O(1)$ bits. Hence, the index would represent a random text using too few bits, a contradiction. □

```
                                  F          L
abracadabra$                      $ abracadabr a
bracadabra$a                      a $abracadab r
racadabra$ab                      a bra$abraca d
acadabra$abr                      a bracadabra $
cadabra$abra                      a cadabra$ab r
adabra$abrac         ⟹            a dabra$abra c
dabra$abraca                      b ra$abracad a
abra$abracad                      b racadabra$ a
bra$abracada                      c adabra$abr a
ra$abracadab                      d abra$abrac a
a$abracadabr                      r a$abracada b
$abracadabra                      r acadabra$a b
```

Figure 1: Example of Burrows-Wheeler transform for the string $T =$ `abracadabra$`. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the column $L =$ `ard$rcaaaabb`.

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

Theorem 4. *For a fixed multiplicative error $(1 + \varepsilon) > 1$, an index built on a text $T[1,n]$ drawn from an alphabet Σ of size σ that approximates the number of occurrences of any pattern P in T within $(1 + \epsilon)$ must use $\Omega(n \log \sigma)$ bits of space.*

By similar arguments we are able to prove that even when restricting to pattern of fixed, sufficiently large length, i.e. $\geq 2 \log n$, the problem remains within the same space complexity. On the other hand, for sufficiently shorter lengths, the problem becomes trivial. Details are deferred to the final version.

4. OPTIMAL ERROR/SPACE SOLUTION

In this section we describe our first solution which is able to report the number of occurrences of any pattern within an additive error at most l. Its error/space trade-off is provably optimal whenever the error l is such that $\log l = O(\log \sigma)$. In this section we will prove the following Theorem:

Theorem 5. *Given $T[1,n]$ drawn from an alphabet Σ of size σ and fixed an error threshold l, there exists an index that answers $\textsf{Count}{\approx_l}(P[1,p])$ in $O(p \times f(n, \sigma))$ time by using $O((n \log(\sigma l))/l + \sigma \log n)$ bits of space, where $f(n, \sigma)$ depends on the chosen `rank` and `select` data structure (see Theorem 2).*

In order to understand this solution we require some background related to compressed full-text indexes [17, 6]. We start by presenting the Burrows-Wheeler Transform [4] which

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

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

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

Figure 2: The algorithm to find the range $[\mathsf{First}_1, \mathsf{Last}_1]$ of rows of $\mathcal{M}(T)$ prefixed by $P[1,p]$ (if any).

is a tool originally designed for data compression that recently turned out to be fundamental for most of known compressed full-text indexes. Then, we present Backward Search [8] that efficiently supports searching operations by exploiting properties of the Burrows-Wheeler Transform.

4.1 Burrows-Wheeler Transform

Burrows and Wheeler [4] introduced a new compression algorithm based on a reversible transformation, now called the *Burrows-Wheeler Transform* (BWT from now on). The BWT transforms the input string T into a new string that is easier to compress. The BWT of T, hereafter denoted by $\mathsf{Bwt}(T)$, consists of three basic steps (see Figure 1): (1) append at the end of T a special symbol \$ smaller than any other symbol of Σ; (2) form a *conceptual* matrix $\mathcal{M}(T)$ whose rows are the cyclic rotations of string $T\$$ in lexicographic order; (3) construct string L by taking the last column of the sorted matrix $\mathcal{M}(T)$. It is $\mathsf{Bwt}(T) = L$.

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 when we sort the rows of $\mathcal{M}(T)$ we are essentially sorting the suffixes of T because of the presence of the special symbol \$. For our purposes, we hereafter concentrate on compressed indexes [17, 6]. They efficiently support the search of any pattern $P[1,p]$ as a substring of the indexed string $T[1,n]$ by requiring a space which is close to the one of best compressors. Two properties are crucial for their design [4]: (a) Given the cyclic rotation of rows in $\mathcal{M}(T)$, $L[i]$ *precedes* $F[i]$ in the original string T; (b) For any $c \in \Sigma$, the ℓ-th occurrence of c in F and the ℓ-th occurrence of c in L correspond to the *same* symbol of string T.

In order to map symbols in L to their corresponding symbols in F, [8] introduced the following function:

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

where $C[c]$ counts the number of symbols smaller than c in the whole string L. Given Property (b) and the alphabetic ordering of F, it is not difficult to see that symbol $L[i]$ corresponds to symbol $F[\mathsf{LF}(i)]$.

4.2 Backward search

The *backward search* [8] is a surprisingly simple algorithm that, given a pattern $P[1,p]$, is able to identify the range of rows in $\mathcal{M}(T)$ prefixed by P in $O(p)$ steps. In particular, the authors proved that data structures for supporting `rank` queries on the string L are enough to search for an arbitrary pattern $P[1,p]$ as a substring of the indexed text T. The resulting search procedure is illustrated in Figure 2. It works in p phases. In each phase it is guaranteed that the following invariant is kept: *At the end of the i-th phase,* $[\mathsf{First}_i, \mathsf{Last}_i]$ *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_p and Last_p are determined via the array C (step 1). At any other phase, the algorithm (see pseudo-code in Figure 2) has inductively computed First_{i+1} and Last_{i+1}, and thus it can derive the next interval of suffixes prefixed by $P[i,p]$ by setting $\mathsf{First}_i = C[P[i]] + \mathtt{rank}_{P[i]}(L, \mathsf{First}_{i+1} - 1) + 1$ and $\mathsf{Last}_i = C[P[i]] + \mathtt{rank}_{P[i]}(L, \mathsf{Last}_{i+1})$. These two computations are actually mapping (via LF) the first and last occurrences (if any) of symbol $P[i]$ in the substring $L[\mathsf{First}_{i+1}, \mathsf{Last}_{i+1}]$ to their corresponding occurrences in F. As a result, the backward-search algorithm requires to solve $2p$ rank queries on $L = \mathsf{Bwt}(T)$ in order to find out the (possibly empty) range $[\mathsf{First}_1, \mathsf{Last}_1]$ of text suffixes prefixed by P. The number of occurrences of P in T is, thus, $occ(P) = \mathsf{Last}_1 - \mathsf{First}_1 + 1$.

The data structures to support `rank` and `select` queries of Theorem 2 achieve better space bounds when they are built on strings which are the result of the Burrows-Wheeler transform (i.e., $L = \mathsf{Bwt}(T)$). In this cases the achieved upper bounds are in terms of the k-th order entropy of the original text for sufficiently small values of k. It follows:

THEOREM 6. *Given a text* $T[1,n]$ *drawn from an alphabet* Σ *of size* σ*, there exists a compressed index that takes* $p \times t_{\mathtt{rank}}$ *time to support* $\mathsf{Count}(P[1,p])$ *where* $t_{\mathtt{rank}}$ *is the time required to perform a* `rank` *query. The following are the best space/time complexities depending on* σ.

Ref.	*space (bits)*	$t_{\mathtt{rank}}$	σ
[9]	$nH_k(T) + o(n)$	$O(1)$	$\log^{O(1)} n$
[9]	$nH_k(T) + o(n) \cdot \log\sigma$	$O(1 + \frac{\log\sigma}{\log\log n})$	$o(n)$
[2]	$nH_k(T) + n \cdot o(\log\sigma)$	$O(\log\log\sigma)$	$O(n)$

The space bounds hold for any $k \le \alpha \log_\sigma n$ *and* $0 < \alpha < 1$.

Notice that compressed indexes support also other operations, like locate and display of pattern occurrences, which are slower than Count in that they require `polylog(n)` time per occurrence (See [17, 6]). We do not enter into details since these kind of operations are out of our scope.

4.3 Our solution

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 $l/2$ of its occurrences). Similarly to backward search, our algorithm searches a pattern $P[1,p]$ by performing p phases. In each of them, it computes two indexes of rows of $\mathcal{M}(T)$ ($\mathsf{First}{\approx}_i$ and $\mathsf{Last}{\approx}_i$) which are obtained by performing rank queries on the sampled BWT and then by applying a correction mechanism. Corrections are required to guarantee that both indexes are within a distance $l/2$ from the actual indexes First_i and Last_i (i.e., the indexes that the backward search would compute for P in phase i). More formally, in each phase it is guaranteed that $\mathsf{First}{\approx}_i \in [\mathsf{First}_i - (l/2)) - 1), \mathsf{First}_i]$ and $\mathsf{Last}{\approx}_i \in [\mathsf{Last}_i, \mathsf{Last}_i + (l/2) - 1]$. Clearly, also the last step obeys to the invariant, hence all rows in $[\mathsf{First}{\approx}_1, \mathsf{Last}{\approx}_1]$ contain suffixes prefixed by P, with the possible exception of the first and last $l/2$ ones. Hence, the maximum error such algorithm can commit is l.

```
Algorithm Count≈_l(P[1,p])
 1. $i = p$, $c = P[p]$, $\mathsf{First}{\approx}_p = C[c] + 1$, $\mathsf{Last}{\approx}_p = C[c+1]$;
 2. while (($\mathsf{First}{\approx}_i \leq \mathsf{Last}{\approx}_i$) and ($i \geq 2$)) do
 3.     $c = P[i-1]$;
 4.     $\mathsf{DiscrFirst}_i = \mathsf{Succ}(\mathsf{First}{\approx}_i, \mathrm{D}_c)$
 5.     $RL = \min(\mathsf{DiscrFirst}_i - \mathsf{First}{\approx}_i, l/2 - 1)$
 6.     $\mathsf{First}{\approx}_{i-1} = \mathsf{LF}(\mathsf{DiscrFirst}_i) - RL$;
 7.     $\mathsf{DiscrLast}_i = \mathsf{Pred}(\mathsf{Last}{\approx}_i, \mathrm{D}_c)$
 8.     $RR = \min(\mathsf{Last}{\approx}_i - \mathsf{DiscrLast}_i, l/2 - 1)$
 9.     $\mathsf{Last}{\approx}_{i-1} = \mathsf{LF}(\mathsf{DiscrLast}_i) + RR$;
10.     $i = i - 1$;
11. if ($\mathsf{Last}{\approx}_i < \mathsf{First}{\approx}_i$) then return "no occurrences of $P$"
    else return $[\mathsf{First}{\approx}_i, \mathsf{Last}{\approx}_i]$.
```

Figure 3: Our algorithm to find the approximate range $[\mathsf{First}_1, \mathsf{Last}_1]$ of rows of $\mathcal{M}(T)$ prefixed by $P[1,p]$ (if any).

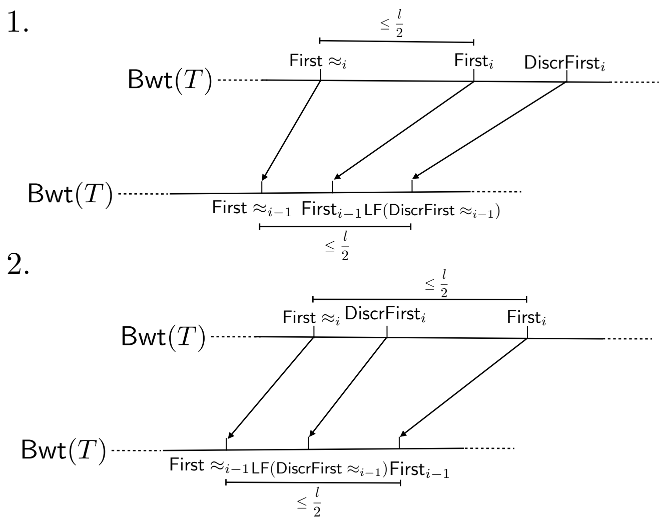


Figure 4: How $\mathsf{First}{\approx}_i$, $\mathsf{DiscrFirst}_i$ and First_i interact.

For each symbol c, the sampling of $L = \mathsf{Bwt}(T)$ keeps track of a set 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 ith occurrence of c in L where $i \bmod l/2 \equiv 0$;
- the position of the last occurrence of c in L.

Algorithm 3 searches a pattern $P[1,p]$ by performing predecessor and successor queries on sets Ds.[3] The crucial steps are lines $4-9$ where the algorithm computes the values of $\mathsf{First}{\approx}_{i-1}$ and $\mathsf{Last}{\approx}_{i-1}$ using the values computed in the previous phase. To understand the intuition behind these steps, let us focus on the computation of $\mathsf{First}{\approx}_{i-1}$ and assume that we know the value of 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 (but larger) to First_i. Clearly, $r \cdot l/2 - l/2 < v \leq r \cdot l/2$. Thus, setting $\mathsf{First}{\approx}_{i-1} = C[c] + r \cdot l/2 - l/2 - 1$ would suffice to guarantee that $\mathsf{First}{\approx}_{i-1} \in [\mathsf{First}_{i-1} - (l/2-1), \mathsf{First}_{i-1}]$. Notice that we are using the crucial assumption that the algorithm knows First_i. If we replace First_i with its approximation $\mathsf{First}{\approx}_i$, this simple argumentation cannot be applied since the error would grow phase by phase. Surprisingly, it is enough to use the simple correction computed at line 5 to fix this problem. The following Lemma provides a formal proof of our claims.

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

Proof. We prove only that $\mathsf{First}{\approx}_i \in [\mathsf{First}_i - (l/2 - 1), \mathsf{First}_i]$ (a similar reasoning applies for $\mathsf{Last}{\approx}_i$). The proof is by induction. For the first step p, we have that $\mathsf{First}{\approx}_p = \mathsf{First}_p$, thus the thesis immediately follows. Otherwise, we assume that $\mathsf{First}{\approx}_i \in [\mathsf{First}_i - (l/2 - 1), \mathsf{First}_i]$ is true and prove that $\mathsf{First}{\approx}_{i-1} \in [\mathsf{First}_{i-1} - (l/2-1), \mathsf{First}_{i-1}]$. Recall that First_{i-1} is computed as $C[c] + \mathtt{rank}_c(L, \mathsf{First}_i - 1) + 1$. We distinguish two cases: (1) $\mathsf{First}_i \leq \mathsf{DiscrFirst}_i$ and (2) $\mathsf{First}_i > \mathsf{DiscrFirst}_i$, both of which are illustrated also in Figure 4.

Case 1) Let z be the number of occurrences of symbol c in the substring $L[\mathsf{First}_i, \mathsf{DiscrFirst}_i - 1]$, so that $\mathsf{First}_{i-1} = \mathsf{LF}(\mathsf{DiscrFirst}_i) - z$[4]. Then, the difference $\mathsf{First}_{i-1} - \mathsf{First}{\approx}_{i-1}$ equals to $\mathsf{LF}(\mathsf{DiscrFirst}_i) - z - \mathsf{LF}(\mathsf{DiscrFirst}_i) + \min(\mathsf{DiscrFirst}_i - \mathsf{First}{\approx}_i, l/2 - 1) = \min(\mathsf{DiscrFirst}_i - \mathsf{First}{\approx}_i, l/2 - 1) - z \leq l/2$. Since (by inductive hypothesis) $0 \leq \mathsf{First}_i - \mathsf{First}{\approx}_i \leq l/2$ and $\mathsf{DiscrFirst}_i$ is the closest discriminant position for c larger than $\mathsf{First}{\approx}_i$, we have that $z \leq \min(\mathsf{DiscrFirst}_i - \mathsf{First}{\approx}_i, l/2 - 1)$. Thus, the difference is also always non negative.

Case 2) Let $k = |\mathsf{First}{\approx}_i - \mathsf{DiscrFirst}_i|$ and z be the number of occurrences of c in $L[\mathsf{DiscrFirst}_i, \mathsf{First}_i - 1]$. Start by noting that $z < l/2$ since $L[\mathsf{DiscrFirst}_i, \mathsf{First}_i - 1]$ contains at most $l/2$ symbols. Since (by inductive hypothesis) $\mathsf{First}_i - \mathsf{First}{\approx}_i < l/2$, we have that $k + z < l/2$; moreover, First_{i-1} can be rewritten as $\mathsf{LF}(\mathsf{DiscrFirst}_i) + z + 1$. Thus, $\mathsf{First}_{i-1} - \mathsf{First}{\approx}_{i-1} = \mathsf{LF}(\mathsf{DiscrFirst}_i) + z + 1 - \mathsf{LF}(\mathsf{DiscrFirst}_i) + k = z + k + 1 \leq l/2$. Finally, since k and z are non negative, $\mathsf{First}_{i-1} - \mathsf{First}{\approx}_{i-1}$ is non negative. □

By combining Lemma 1 with the proof of correctness of Backward Search (Lemma 3.1 in [8]) we easily obtain the following Theorem.

Theorem 7. *For any pattern $P[1,p]$ that occurs* $\mathsf{Count}(P)$ *times in T Algorithm 3 returns in $O(p)$ steps as result a value* $\mathsf{Count}{\approx}_l(P) \in [\mathsf{Count}(P), \mathsf{Count}(P) + l - 1]$.

Notice that, if $[\mathsf{First}, \mathsf{Last}]$ is the range of indexes corresponding to the consecutive suffixes that are prefixed by

[3]A predecessor query Pred(x,A) returns the predecessor of x in a set A i.e., $\max\{y \mid y \leq x \ \wedge \ y \in A\}$. A successor query is similar but finds the minimum of $y \geq x$.

[4]Observe that $L[\mathsf{DiscrFirst}_i] = c$ by definition of discriminant position for c.

P, then the algorithm identifies a range $[\mathsf{First}\approx, \mathsf{Last}\approx]$ such that $\mathsf{First} - l/2 < \mathsf{First}\approx \leq \mathsf{First}$ and $\mathsf{Last} \leq \mathsf{Last}\approx < \mathsf{Last} + l/2$.

It remains to show how to represent the sets of discriminant positions D_c to support predecessor and successor queries on them. We represent each of these sets by means of two different objects. We conceptually divide the string $L = \mathsf{Bwt}(T)$ into $\lceil 2n/l \rceil$ 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 D_c belonging to block i. Considering sets B_i as strings (with arbitrary order), we compute the string $B = B_0\#B_1\#\dots B_{2n/l}\#$ where $\#$ is a symbol outside Σ and augment it with `rank` and `select` data structures (see Theorem 2). 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 jth one in B, then $V[j] = x \bmod l/2$. The following Lemma states that a constant number `rank` and `select` queries on B and V suffice for computing $\mathsf{Pred}(x, \mathrm{D}_c)$ and $\mathsf{Succ}(x, \mathrm{D}_c)$.

Lemma 2. *$\mathsf{Pred}(x, D_c)$ and $\mathsf{Succ}(x, D_c)$ can be computed with a constant number of* `rank` *and* `select` *queries on B and V.*

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

Once we have computed the correct discriminant positions, Algorithms 3 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 D_c but the largest one, we have that $\mathsf{LF}(d) = C[c] + (i-1) \cdot l/2 + 1$ where i is such that D_c's ith element in left-to-right position is d. For the largest discriminant position d in 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.

Proof of Theorem 5. Correctness has been proved. The time complexity is easily seen to be $O(|P|)$ instances of Lemma 2, hence the claim. 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 $2n/l$ in total, hence the array V requires at most $O(n/l)$ cells of $O(\log l)$ bits each. Finally, the string B requires one symbol per block plus one symbol per discriminant positions, accounting for $O(n \log(\sigma)/l)$ bits in total. The theorem follows. □

5. COMPACT PRUNED SUFFIX TREE

Let $\mathcal{PST}_l(T)$ be the pruned suffix tree as discussed in the introduction, and let m be the number of its nodes. Recall that $\mathcal{PST}_l(T)$ is obtained from the suffix tree of T by removing all the nodes with less than l leaves in their subtrees, and hence constitutes a good solution to our lower-sided error problem: when $\mathsf{Count}(P) \geq l$, the answer is correct, otherwise an arbitrary number below l can be returned. Compared with the solution of Section 4 it has the great advantage of being perfectly correct if the pattern appears frequently enough, but is extremely space inefficient. Our objective in this section is that explaining a compact version of the $\mathcal{PST}_l(T)$, by means of proving the following:

Theorem 8. *Given $T[1, n]$ drawn from an alphabet Σ of size σ and given an error threshold l, there exists a representation of $\mathcal{PST}_l(T)$ using $O(m \log(\sigma l) + \sigma \log n)$ bits that can answer to $\mathsf{Count}_{\geq l}(P)$ in $O(|P| \times f(n, \sigma))$ time where m is the number of nodes of $\mathcal{PST}_l(T)$ and $f(n, \sigma)$ is the chosen* `rank` *and* `select` *time complexity summed up (see Theorem 2).*

To appreciate Theorem 8, as also observed in the introduction, consider that the original $\mathcal{PST}_l(T)$ representation requires, apart from node pointers, labels together with their length for a total of $O(m \log n + g \log \sigma)$. 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 written.

5.1 Suffix trees

We now review the suffix tree[12] in greater detail, introducing useful notation and some of its properties. The *suffix tree* [12] of a text T is the compacted trie, i.e., a trie in which all unary nodes omitted, denoted as $\mathcal{ST}(T)$ or simply $\mathcal{ST}$, built on all the n suffixes of T. We ensure that no suffix is a proper prefix of another suffix by simply assuming that a special symbol, say \$, terminates the text T. The symbol \$ does not appear anywhere else in T and is assumed to be lexicographically smaller than any other symbol in Σ. This constraint immediately implies that 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. For a given edge, the *edge label* is simply the substring in T corresponding to the edge. For edge between nodes u and v in $\mathcal{ST}$, the edge label (denoted $\mathsf{label}(u, v)$) is always a non-empty substring of T. For a given node u in the suffix tree, 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 simply $|\mathsf{pathlabel}(u)|$. In order to allow a linear space representation of the tree, each edge label is usually represented by a pair of integers denoting, respectively, the starting position in T of the substring describing the edge label and its length. In this way, the suffix tree can be stored in $\Theta(n \log n)$ bits of space. It is very well-known that to search a pattern $P[1, p]$ in T we have to identify, if any, the highest node u in $\mathcal{ST}$ such that P prefixes $\mathsf{pathlabel}(u)$. To do this, we start from the root of $\mathcal{ST}$ and follow the path matching symbols of P, until 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 any node u the number $C(u)$ of leaves in its subtree. Therefore, this algorithm counts the occurrences of any pattern $P[1,p]$ in time $O(p \log \sigma)$. This time complexity can be reduced up to $O(p)$ by placing a (minimal) perfect hashing function [13] in each node to speed up percolation. This will increase the space just by a constant factor.

5.2 Computing counts

As a crucial part of our explanation, we will refer to nodes using their preorder traversal times, with an extra requirement. Recall the branching symbol in a set of children of node u is the first symbol of children edge labels. During the visit we are careful to descend into children in ascending lexicographical order over their branching symbols. Therefore, $u < v$ iff 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 the basic information that our algorithm must recover: Given a node $u \in \mathcal{PST}_l(T)$ we would like to compute $C(u)$, the number of occurrences of $\mathsf{pathlabel}(u)$ as a substring in T [5]. A straightforward storage of such data would require $m \log n$ bits for a tree of m nodes. We prove we can obtain better bounds and still compute $C(u)$ in $O(1)$ time, based on the following simple observation:

Observation 1. *Let u be a node in $\mathcal{PST}_l(T)$ and let $v_1, v_2, \ldots, v_k$ be children of u in $\mathcal{PST}_l(T)$ that have been pruned away. Denote by $g(u)$ the sum $C(v_1) + C(v_2) + \cdots + C(v_k)$. Then $g(u) < \sigma l$.*

Proof. Each of the v_is represents a suffix that has been pruned away, hence, for any i, $C(v_i) < l$ in T by definition. Since each node can have at most σ children, the observation follows. $\square$

Note that Observation 1 applies in a stronger form to leaves, where for a leaf x, $C(x) = g(x)$. We refer to the $g(\cdot)$ values as *correction factors* (albeit for leaves they are actual counts). For an example refer to Figure 5. It is easy to see that to obtain $C(v)$ it suffices to sum all correction factors of all descendants of v in $\mathcal{PST}_l(T)$. Precisely, it suffices to build the binary string $G = 0^{g(0)}10^{g(1)}1 \cdots 0^{g(m-1)}1$ together with support for binary `select` queries.

Lemma 3. *Let $v \in \mathcal{PST}_l(T)$ and let z be the identifier of its rightmost leaf, Define $\mathsf{CNT}(u,z) = \mathtt{select}_1(G,z) - z - \mathtt{select}_1(G,u) + u$. Then $C(u) = \mathsf{CNT}(u,z)$.*

Proof. By our numbering scheme, $[u,z]$ contains all values in G for nodes in the subtree of u. $\mathtt{select}_1(G,x) - x$ is equivalent to $\mathtt{rank}_0(G, \mathtt{select}_1(G,x))$, i.e. it sums up all correction factors in nodes before x in the numbering scheme. Computing the two prefix sums and subtracting is sufficient. $\square$

Lemma 4. *Let m be the number of nodes in $\mathcal{PST}_l(T)$, then G can be stored using at most $m \log(\sigma l) + O(m)$ bits and each call $\mathsf{CNT}(u,z)$ requires $O(1)$ time.*

Proof. Each correction factor has size σl at most, hence the number of **0**s in G is at most $m\sigma l$. The number of **1**s in G is m. The thesis follows by storing G with the structure of Lemma 1. $\square$

5.3 Finding the correct node

In our solution we will resort to the concepts of suffix links and inverse suffix links in a suffix tree. For each node u of $\mathcal{PST}_l(T)$, the *suffix link* $\mathsf{SL(u)}$ is v iff we obtain $\mathsf{pathlabel}(v)$ from $\mathsf{pathlabel}(u)$ by removing its first symbol. The *inverse suffix link* of v for some symbol c, denoted $\mathsf{ISL(v,c)}$, is u iff $u = SL(v)$ and the *link symbol* is c. 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 $\mathsf{ISL(u,c) = v}$ exists only if $\mathsf{pathlabel}(v) = c \cdot \mathsf{pathlabel}(u)$, 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 descendant (including w) has an inverse suffix link for c. The value of $\mathsf{VISL}(w,c)$ is equal to $\mathsf{ISL}(u,c)$ where u 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 pathlabel is prefixed by $c \cdot \mathsf{pathlabel}(w)$.

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 linking 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 σ 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 VISL function.

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

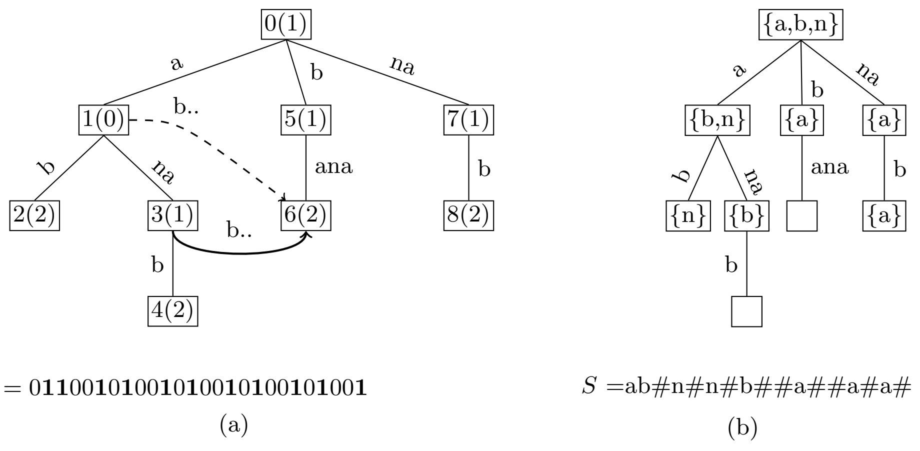
The following two Lemmas state that inverse suffix links preserve the relative order between nodes.

Lemma 5. *Let w, z be nodes in $\mathcal{PST}_l(T)$ 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 β and ζ. By definition of inverse suffix link, we have that $\mathsf{pathlabel}(w') = c \cdot \alpha \cdot \beta$ and $\mathsf{pathlabel}(z) =$

[5]Here, $C(u)$ is the number of leaves in the subtree of u in the *original* suffix tree.

[6]Notice that w and u are the same node whenever w has an inverse suffix link for c.



$G = 0\mathbf{1}1\mathbf{0}0\mathbf{1}0\mathbf{1}0\mathbf{0}1\mathbf{0}1\mathbf{0}0\mathbf{1}0\mathbf{1}0\mathbf{0}1\mathbf{0}1\mathbf{0}0\mathbf{1}$

(a)

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

(b)

Figure 5: The pruned suffix tree of banabananab with threshold 2. Each node contains its preorder traversal id and in brackets, its correction factor. Arrow denotes an inverse suffix link for b; dashed arrow a virtual one. (b) The same PST of (a), with information associated to Theorem 9. Each node is given the set of symbols for which a virtual inverse suffix link is defined. The binary string G contains corrections factor in unary format (separators are in bold for clarity). The string S contains the separated encoding, in preorder traversal, of suffix links chosen by our procedure.

$c \cdot \alpha \cdot \zeta$. Since w and z do not share the same path below u, the first symbols of β and ζ 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 $\mathsf{ISL(u,c)} = u' = v$. □

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 is the path label of $v' = \mathsf{ISL}(v,c)$, u' precedes v' in the preorder traversal of $\mathcal{PST}_l(T)$. □

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 an other descendant of u having an inverse suffix link for c, then this node must be also descendant of v.

Lemma 7. *For any node u in the $\mathcal{PST}_l(T)$, if exists, the highest descendant of u (including u) having an inverse suffix link for a symbol c 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 an unique node which is common ancestor of all of these nodes. □

In our solution we conceptually associate each node u in $\mathcal{PST}_l(T)$ with the set of symbols $\mathcal{D}_u$ for which u has an inverse suffix link. We represent each set with a string $\text{Enc}(\mathcal{D}_u)$ built by concatenating the symbols in $\mathcal{D}_u$ in any order and ending with a special symbol # not in Σ. We then build a string S as $\text{Enc}(\mathcal{D}_0)\text{Enc}(\mathcal{D}_1)\cdots\text{Enc}(\mathcal{D}_{m-1})$ so that the encodings follow the preorder traversal of the tree. We also define the array $C[1,\sigma]$ whose entry $C[c]$ stores the number of nodes of $\mathcal{PST}_l(T)$ whose path label starts 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 induced from real ones.

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

Proof. Let $\mathcal{A}$ be the set of nodes of $\mathcal{PST}_l(T)$ whose pathlabel is lexicographically smaller than the pathlabel 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, $\text{Enc}(\mathcal{D}_w)$ for $w \in \mathcal{A}$ and $\text{Enc}(\mathcal{D}_w)$ 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 $\mathtt{select}_\#(S,u)$ and $\mathtt{select}_\#(S,z)$ 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 by preserving their relative order. Thus, the first node in $\mathcal{B}$ that has an inverse suffix link for c is mapped to node $C[c] + c_u + 1$. By the numbering of node, this first node is obviously also the highest one. Thus, proposition (b) is proved.

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

Figure 5 illustrates the whole situation: from a pruned suffix tree, we illustrate the resulting string containing correction factors and the data to rebuild inverse suffix links.

```
Algorithm Count$_{\geq l}(P[1,p])$
  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.     $u_{i-1} = \mathsf{VISL}(u_i, c) = C[c] + \mathtt{rank}_c(S, \mathtt{select}_{\#}(S, u_i)) + 1$;
  5.     $z_{i-1} = \mathsf{VISL}(z_i, c) = C[c] + \mathtt{rank}_c(S, \mathtt{select}_{\#}(S, z_i))$;
  6.     $i = i - 1$;
  7. if $(u_i = z_i)$ then return "no occurrences of $P$" else return
     $\mathsf{CNT}(u_1, z_1)$
```

Figure 6: Our algorithm to report the number of occurrences of a pattern $P[1,p]$ in our Compact Pruned Suffix Tree.

Exploiting VISL, Algorithm 6 searches a pattern $P[1,p]$ backward. The algorithm starts by setting u_p to be $C[P[p]]+1$. At the ith step, we inductively assume that u_{i+1} is known, and its pathlabel to be 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 $\mathsf{VISL}(u_{i+1}, P[i])$ 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 6 retrieves $C(u)$, where u is the highest node of $\mathcal{PST}_l(T)$ 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 $P[p]$.

By hypothesis, we assume that u_{i+1} is the highest node in $\mathcal{PST}_l(T)$ whose path label is prefixed by $P[i+1,p]$, and we want to prove the same for $u_i = \mathsf{VISL}(u_{i+1}, P[p-i])$. The fact that $\mathsf{pathlabel}(u_i)$ is prefixed by $P[p-i,p]$ 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 an other 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 VISL is also the highest one in u'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 the claimed node u does not exist (i.e. P occurs in T less than l times). Once u is found, also z is present, hence we resort to Lemma 3 to obtain $C(u) = \mathsf{CNT}(u, z)$. □

By combining this Theorem with the discussion of the previous section, we derive the proof of our main Theorem.

Proof of Theorem 8. We need to store: the C array, holding the count of nodes in $\mathcal{PST}_l(T)$ whose pathlabel prefixed by each of the σ characters; the G string, together with binary `select` capabilities and the S string, together with arbitrary alphabet `rank` and `select` capabilities. Let m be the number of nodes in $\mathcal{PST}_l(T)$. We know C occupies at most $\sigma \log n$ bits. By Lemma 4 G occupies at most $m \log(\sigma l) + O(m)$ bits. String S can be represented in different ways, related to σ, picking a choice from Theorem 2, but the space is always limited by $m \log \sigma + o(m \log \sigma)$. Hence the total space is $\sigma \log n + m \log(\sigma l) + O(m) + O(m \log(\sigma)) = O(m \log(\sigma l) + \sigma \log n)$, as claimed. For the time complexity, at each of the p steps, we perform four `rank` and `select` queries on arbitrary alphabets which we account as $f(\sigma)$. The final step on G takes $O(1)$ time, hence the bound. □

6. EXPERIMENTS

In this section we show an experimental comparison among the known solutions and our solutions. We use four different datasets downloaded from `Pizza&Chili` corpus [6] that correspond to four different types of texts: DNA sequences, structured text (XML), natural language and source codes. Text and alphabet size for the texts in the collection are reported in the first column of Figure 7.

We compare the following solutions:

- `FM-index`. This is an implementation of a compressed full-text index available at the `Pizza&Chili` site [6]. 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` $-\, l$. This is the implementation of our solution presented in Section 4.
- `PST`$-\,l$. This is an implementation of the Pruned Suffix Tree as described in [15].
- `CPST` $-\, l$. This is the implementation of our Compact Pruned Suffix Tree described in Section 5.

Recall that `APPROX` $-\, l$ reports results affected by an error of at most l while `PST` $-\, l$ and `CPST` $-\, l$ are always correct whenever the pattern occurs at least l times in the indexed text. Due to the lack of space we leave details regarding the various implementations to the final version.

The plots in Figure 8 show the space occupancies of the four indexes depending on the chosen threshold l. We do not plot space occupancies worse than `FM-index`, since in those cases `FM-index` is clearly the index to choose.

As anticipated in the previous sections, it turns out that in all texts of our collection the number of nodes in the pruned suffix tree is small (even smaller than n/l) (these statistics are reported in Figure 7). 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 l times. Even though, the latter remains interesting due to its better theoretical guarantees. In both solutions by halving the error threshold, we obtain indexes that are between 1.75 and 1.95 times smaller. Thus, we can obtain very small indexes by setting relatively small values of l. As an example, `CPST` with $l = 256$ 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 value of l. As an example, `FM-index` requires 232.5 Mbytes on `english` which is roughly 45 times larger than `CPST`-256.

Dataset	Size	σ	$l = 8$			$l = 64$			$l = 256$		
			$\|T\|/l$	$\|\mathcal{PST}_l\|$	$\sum_i \|\mathsf{edge}(i)\|$	$\|T\|/l$	$\|\mathcal{PST}_l\|$	$\sum_i \|\mathsf{edge}(i)\|$	$\|T\|/l$	$\|\mathcal{PST}_l\|$	$\sum_i \|\mathsf{edge}(i)\|$
`dblp`	275	96	36064	28017	1034016	4508	3705	103383	1127	941	20200
`dna`	292	15	38399	42361	814993	4799	5491	102127	1199	1317	19194
`english`	501	225	65764	53600	660957	8220	6491	64500	2055	1616	14316
`sources`	194	229	25475	25474	11376730	3184	3264	9430627	796	982	8703817

Figure 7: Statistics on the datasets. The second column denotes the original text in MBytes. Each subsequent group of three columns describe $\mathcal{PST}_l$ information for a choice of ℓ: expected amount of nodes, $|T|/l$; real amount of nodes in $\mathcal{PST}_l(T)$; sum of length of labels in $\mathcal{PST}_l(T)$. All numbers are expressed in thousands.

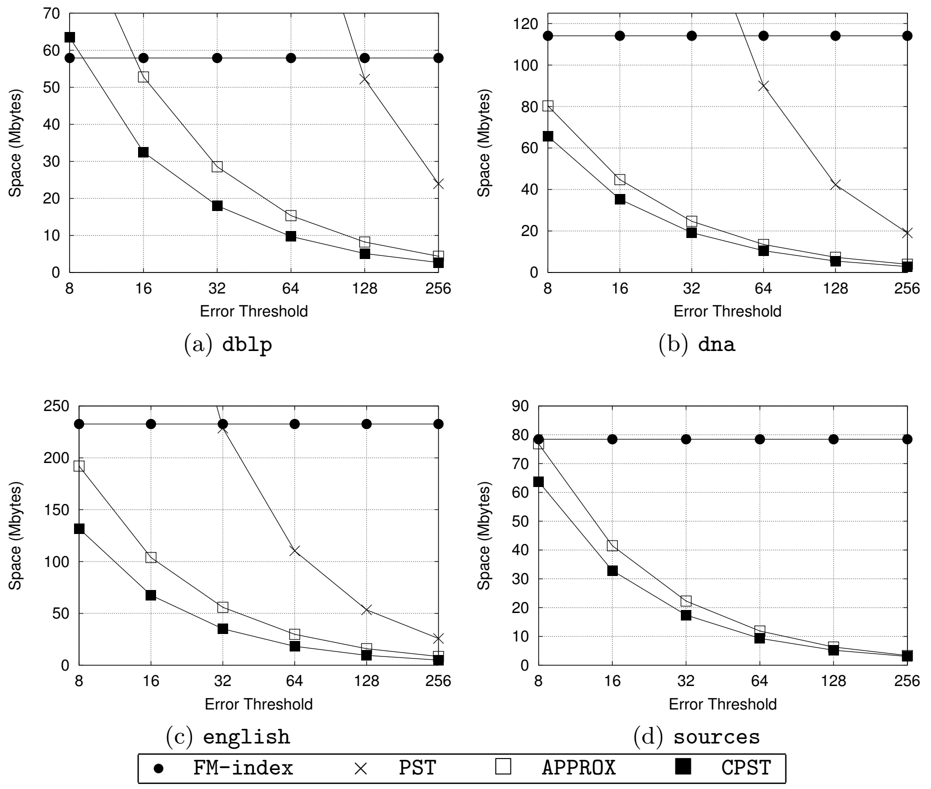


Figure 8: Space occupancies of indexes as a function of the error threshold l.

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 l decreases (see Figure 7). Moreover, this quantity strictly depends on the indexed text while the number of nodes is more stable. Thus, the performances of `PST` are erratic: worse than `CPST` by a factor 6 on `english` that 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 $l = 8$.

For what concerns applications, we use our best index, i.e. `CPST` together with one estimation algorithm: `MOL`. The reader can find a lightweight explanation of the algorithm in Section 7.2; the algorithm is oblivious of the underlying data structure as long as a lower-sided error one is used. We performed (details omitted) a comparison between `MO`, `MOL` and `KVI` [15, 14] 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 term of running memory. Finally, we tried to compare with `CRT` [5]; however, we lacked the original implementation and a significative training set for our data sets. Hence, we discarded the algorithm from our comparison.

Figure 9 shows the average error of the estimates obtained with `MOL` on our collection by using either `CPST` or `PST` as base data structure. For each text, we searched 4 Millions patterns of different lengths that we randomly extracted from the text. For each set we identified two pairs of thresholds such that our `CPST` and `PST` have roughly the same space occupancy. Thus, this table gives the idea of 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.

Dataset	Indices	$\|P\| = 6$	$\|P\| = 8$	$\|P\| = 10$	$\|P\| = 12$	Avg improvement
`dblp`	PST-256	10.06 ± 32.372	12.43 ± 34.172	14.20 ± 35.210	15.57 ± 36.044	19.03×
	CPST-16	0.68 ± 1.456	0.86 ± 1.714	1.00 ± 1.884	1.14 ± 2.009	
`dna`	PST-256	0.47 ± 1.048	0.49 ± 2.433	4.26 ± 15.732	11.09 ± 19.835	5.51×
	CPST-32	0.47 ± 0.499	0.43 ± 0.497	0.52 ± 0.904	1.77 ± 2.976	
`english`	PST-256	7.03 ± 27.757	12.45 ± 31.712	13.81 ± 28.897	11.43 ± 23.630	9.68×
	CPST-32	0.80 ± 2.391	1.40 ± 3.394	2.07 ± 3.803	2.45 ± 3.623	
`sources`	PST-11000	816.06 ± 1646.57	564.94 ± 1418.53	400.62 ± 1229.35	313.68 ± 1120.94	792.52×
	CPST-8	0.70 ± 1.028	0.93 ± 1.255	1.13 ± 1.367	1.28 ± 1.394	

Figure 9: Comparison of error (difference between number of occurrences and estimate) for `MOL` estimates over different pattern lengths. `PST` and `CPST` parameters chosen as to obtain close index sizes. Tests performed on $1M$ random patterns appearing in the text. The last column shows the average factor of improvement obtained by using our `CPST` instead of `PST`.

7. RELATED WORK

In this section we introduce known solutions (or unpublished simple solutions that use known techniques) for the Substring Occurrence Estimation problem. We also review the literature about our main application, Substring Selectivity Estimation. Unfortunately, they suffer from two important drawbacks: 1) they are not space optimal since they require $\Theta(n \log(n)/l)$ bits of space; 2) they actually solve a relaxed version of our problem in which we do not care on the results whenever the patterns occur less than l times (in these cases, the result may be arbitrarily far from $\mathsf{Count}(P)$, i.e., ignoring the rules of lower-sided and uniform error).

7.1 Occurrence estimation

$\mathcal{PST}_l$ has been already explored in previous sections, hence we skip it and move to other solutions. An alternative pruning strategy consists in building a pruned Patricia Trie $\mathcal{PT}_{l/2}(T)$ that stores just a suffix every $l/2$ suffixes of T sorted lexicographically and resort to Blind Search. A plain Patricia Trie $\mathcal{PT}(T)$ [12] coincides with $\mathcal{ST}(T)$ in which we replace each substring labeling an edge by its first symbol only, called *branching symbol*. More formally, let $T_1, T_2, \ldots, T_n$ denote the n suffixes of T sorted lexicographically, $\mathcal{PT}_{l/2}(T)$ is the patricia trie of the set of $O(n/l)$ strings $\mathcal{S} = \{T_i \mid i \equiv 1 \pmod{l/2}\}$. The pruned patricia trie $\mathcal{PT}_{l/2}(T)$ can be stored in $O(n/l \cdot (\log\sigma + \log n)) = O(n \log n/l)$ bits. We use the blind search described in [7] to search 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, without the original text it is not possible to exploit it). Once we identify the node u, we return the number of leaves descending from that node multiplied by l. If P occurs at least $l/2$ times in T, it is easy to see that the number of reported occurrences is a correct approximation of its occurrences in T. Instead, if P occurs less than $l/2$ times in T, the blind search returns a different node. Thus, in such cases the algorithm may fail reporting as result a number of occurrences that may be arbitrarily far from the correct one.

A similar solution resorts to a recent data structure presented by Belazzougui *et al.* [3]. Their solution solves via hashing 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, 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 l. Since in our case $L = \Theta(n)$, the index requires $O(n\log(n\log\sigma)/l) = O(n\log n/l)$ bits of space. As in the case of pruned patricia tries, the answer is arbitrary when P is not prefix of any suffix in $\mathcal{S}$ (i.e., it occurs less that l times). Hence, this solution improves the time complexity but has the same drawback of the previous one.

7.2 Selectivity estimation

In this section we present in more detail the three main algorithms for substring selectivity estimation: `KVI` [15], the class of `MO`-based estimators [14] and `CRT` [5], in chronological order. For a given threshold l, the work of `KVI` starts by assuming to have a data structure answering correctly to queries $\mathsf{Count}(P)$ when $\mathsf{Count}(P) \geq l$ and strives to obtain a one-sided error estimate for infrequent ($< l$) strings. It also assume the data structure can detect if $\mathsf{Count}(P) < l$. Their main observation is as follows: let $P = \alpha\beta$ where $\mathsf{Count}(P) < l$ and assume $\mathsf{Count}(\alpha) \geq l$ and $\mathsf{Count}(\beta) \geq l$, 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 β appearing in the text given that α appears is roughly the same of β appearing by itself. Generalizing this concept, `KVI` starts from P and retrieves the longest prefix of P, say P', such that $\mathsf{Count}(P') > l$, and then reiterates 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 α and β, the maximum overlap $\alpha \oslash \beta$ is the longest prefix of β that is also a suffix of α. 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 constraint network 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 ζ in the lattice, also its l-parent and r-parent are in the lattice. Two nodes β and ζ of the lattice are connected if β is an l-parent or an r-parent of ζ 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, 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.

8. 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 lower bounds. A natural question is: is there a way to relax the model, in order to circumvent the multiplicative lower bound (Theorem 4)? For example, what if we allow non-existing substrings to have an arbitrary estimation error, forcing all others with a multiplicative bound?

Acknowledgments

The authors would like to thank Roberto Grossi for fruitful discussions on the topic of this paper.

This work was partially supported by the EU-PSP-BPN-250527 (ASSETS), the POR-FESR 2007-2013 No 63748 (VISITO Tuscany) projects, the MIUR of Italy under project AlgoDEEP prot. 2008TFBWL4, and the PRIN MadWeb 2008.

9. REFERENCES

[1] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *ISAAC (2)*, pages 315–326, 2010.

[2] J. Barbay, M. He, J.I. Munro, and S. Srinivasa Rao. Succinct indexes for string, binary relations and multi-labeled trees. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[3] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Fast prefix search in little space, with applications. In *ESA (1)*, pages 427–438, 2010.

[4] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[5] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 227–, 2004.

[6] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[7] P. Ferragina and R. Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46:236–280, March 1999.

[8] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[9] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[10] P. Ferragina and R. Venturini. Compressed permuterm index. In *SIGIR*, pages 535–542, 2007.

[11] R. Grossi and J. 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.

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

[13] T. Hagerup and T. 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.

[14] H.V. Jagadish, R. T. Ng, and D. Srivastava. Substring selectivity estimation. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '99, pages 249–260, 1999.

[15] P. Krishnan, J. S. Vitter, and B. R. Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *SIGMOD Conference*, pages 282–293, 1996.

[16] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[17] G. Navarro and V. Mäkinen. Compressed full text indexes. *ACM Computing Surveys*, 39(1), 2007.

[18] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*, 2007.

[19] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, 1999.