Distribution-aware compressed full-text indexes*

Paolo Ferragina[1], Jouni Sirén[2], and Rossano Venturini[3]

[1] Dept. of Computer Science, Univ. of Pisa. ferragina@di.unipi.it
[2] Dept. of Computer Science, Univ. of Helsinki. jltsiren@cs.helsinki.fi
[3] ISTI-CNR, Pisa. rossano.venturini@isti.cnr.it

Abstract. In this paper we address the problem of building a compressed self-index that, given a distribution for the pattern queries and a bound on the space occupancy, minimizes the expected query-time within that index-space bound. We solve this problem by exploiting a reduction to the problem of finding a minimum weight K-link path in a particular Directed Acyclic Graph. Interestingly enough, our solution is independent of the underlying compressed index in use. Our experiments compare this optimal strategy with several other standard approaches, showing its effectiveness in practice.

1 Introduction

String processing and searching tasks are at the core of modern web search, IR, data base and data mining applications. Most of text manipulations required by these applications involve, sooner or later, *searching* those (long) texts for (short) patterns or *accessing* portions of those texts for subsequent processing/mining tasks. Despite the increase in processing speeds of current CPUs and memories/disks, sequential text searching long ago ceased to be a viable approach, and indexed text searching has became mandatory.

Data compression and indexing seem "opposite approaches" because the former aims at removing data redundancies, whereas the latter introduces extra data in the index to support faster operations. This dichotomy was successfully addressed starting from the year 2000 [4, 7], due to various scientific achievements that showed how to relate Information Theory with String-Matching concepts, in a way that index regularities that show up when data is compressible are discovered and exploited to reduce index occupancy without impairing query efficiency (see the surveys [11, 3] and references therein). The net result has been the design of *compressed data structures* for indexing texts (aka *compressed indexes*, or *compressed and searchable data formats*) that take space close to the kth order entropy of the input text, and support the powerful *substring* queries and the *extraction* of arbitrary portions of data. Due to this latter feature, these data structures are sometime called *self*-indexes.

* This work was partially supported by EU-PSP-BPN-250527 (ASSETS), POR-FESR 2007-2013 No 63748 (VISITO Tuscany) projects, MIUR of Italy under project PRIN MadWeb 2008, the Finnish Doctoral Programme in Computational Sciences, Academy of Finland (project 1140727), and the Nokia Foundation.

As experimentally shown in [3, 5], these self-indexes are very space-efficient (close to best known compressors), and most of them are particularly fast in counting the number of occurrences of the input pattern. Their bottleneck is in the Locate queries, which are roughly between two and three order of magnitude slower than what is achievable with the classic Suffix Array data structure. Also the Extract operation is quite slow compared with other compression methods for sufficiently long decompressed portions. In addition, for Locate and Extract, these indexes need to store some extra information that induces a trade-off between space and time efficiency: the larger is this extra space, the faster is the resulting index. At high level, the extra information is obtained by *sampling* entries of the suffix array at regular distance s_{SA}. This parameter governs the space/time trade-off, because on one hand, each occurrence of the searched pattern is located in at most s_{SA} steps; but on the other hand, the space required is $O(\frac{n \log n}{s_{SA}})$ bits, where n is the length of the indexed text.

Even though the last years have seen a proliferation of different compressed full text indexes [11, 3], the above sampling strategy has remained almost unchanged since the very first proposals. This strategy implicitly assumes all text positions to have uniform probability of being located or extracted. But uniform distributions are rare in practice, where we often observe (very) skewed distributions. For example, it is well-known that requests in IR or database systems are drawn accordingly to power law or Zipfian distributions (e.g. see [14] and references therein).

Given these premises we address in this paper the following question: Is it possible to build a distribution-aware compressed self-index that optimizes the expected query-time by occupying a given space? Namely, given the distribution of the subsequent queries and a bound on the space occupancy, the goal is to find a sampling strategy that induces that space bound and minimizes the expected time required for solving Locate/Extract queries drawn accordingly to the input distribution. We solve this problem by exploiting a reduction to the problem of finding a minimum weight K-link path in a particular Directed Acyclic Graph (DAG) (Section 3). Interestingly enough, our solution provides a way to optimally select a set of sampled positions that could be blindly used by mostly known compressed indexes without changing their Locate/Extract algorithms.

In the experimental section (Section 4) we compare our optimal sampling strategy against several other strategies over two large datasets of HTML pages and XML documents. The experiments were performed by using RLCSA, which is an implementation of Compressed Suffix Array (Csa). Although restricted to this single index, our experiments quantify some measures that are independent on the particular implementation of a compressed index, and thus can be adopted to extrapolate conclusions for other indexes as well. Overall we show that our optimal sampling is from 4 to 10 times faster than regular sampling. We also compare our optimal strategy against two heuristic approaches, showing that ours is up to a factor 2.5 faster. One of them is the immediate strategy that "caches" the most probably accessed positions. This heuristic can be poor both in theory and in practice due to the fact that it does not consider interdependencies among sampled positions induced by Locate and Extract algorithms. Roughly speaking, in many circumstances it is more convenient to sample a position whose access probability is not among the top, provided that it is followed by positions having sufficiently high access probabilities. Discovering all these cases is a peculiarity of our optimal solution. These considerations are explained more formally in Section 4, where we quantify also the impact of the various heuristics by performing a significant set of experiments.

2 Background

The large space occupancy of (classical) full-text indexes, like Suffix Tree and Suffix Array, has driven researchers to design the so-called *compressed* full-text indexes. These indexes deploy algorithmic techniques and mathematical tools that lie at the crossing point of three distinct fields: data compression, data structures and databases (see e.g. [4, 7, 11, 3]). Most of these indexes can be classified into two families — namely, FM-indexes (Fmi) and Compressed Suffix Arrays (Csa) — and achieve efficient query times and space close to the one achievable by the best known compressors. In theory, these indexes require $O(nH_k(T)) + o(n \log \sigma)$ bits of space, where $H_k(T)$ is the kth order empirical entropy of text T of length n, and σ is the alphabet size. This bound is appealing because it can be sublinear in $n \log \sigma$ for highly compressible texts. We recall that $nH_k(T)$ is the classic Information-Theoretic lower bound to the storage complexity of T by means of any k-th order compressor (see e.g. [10] for more details). In addition to being compressed, the index is able to efficiently support the following three operations:

- $\mathsf{Count}(P[1, p])$ returns the number of occurrences of pattern P in the text;
- $\mathsf{Locate}(P[1, p])$ returns the starting positions of all occurrences of pattern P in the text;
- $\mathsf{Extract}(l, r)$ extracts the substring $T[l, r]$.

2.1 The FM-index family

These compressed indexes were introduced by Ferragina and Manzini in [4], who devised a way to utilize the relation between the suffix array data structure and the *Burrows-Wheeler Transform* (shortly, Bwt [2]) in efficient time and space. The Bwt is a reversible transformation that permutes the symbols of the input string T into a new string $L = \mathsf{Bwt}(T)$ that is easier to compress. This permutation is the last column of a conceptual matrix $\mathcal{M}(T)$ whose rows are the cyclic rotations of string $T\$$ in lexicographic order.

It is well-known that the original text T can be obtained backwards from L by resorting to a function LF that maps row indexes to row indexes, and is defined as follows [4]: if the Bwt maps $T[j-1]$ to $L[i']$ and $T[j]$ to $L[i]$, then $LF(i) = i'$ (so LF implements a sort of *backward* step over T). Now, since the first row of $\mathcal{M}(T)$ is $\$T$, it can be stated that $T[n] = L[0]$ and, in general, $T[n-i] = L[LF^i(0)]$, for $i = 1, \ldots, n-1$.

Ferragina and Manzini [4] proposed a way to combine the compressibility of the Bwt with the indexing power of the suffix array. In particular, showed that searching operations on T can be reduced to counting queries of *single* symbols in L, now called rank operations. For any symbol $c \in \Sigma$ and position i in L, the query $\mathsf{rank}_c(L, i)$ returns how many times the symbol c appears in $L[1, i]$. An FM-index then consists of three key tools: a compressed representation of $\mathsf{Bwt}(T)$ that supports efficient rank queries, a small array $C[c]$ that tells how many symbols smaller than c appear in T (this takes $O(\sigma \log n)$ bits), and the so called *backward search* algorithm that implements the Count query by using the two structures. More precisely, Fmi searches the pattern $P[1, p]$ backwards in p steps, which eventually identify the interval of text suffixes that are prefixed by P or, equivalently, the interval of rows of $\mathcal{M}(T)$ that are prefixed by P. This is done by maintaining, inductively for $i = p, p-1, \dots, 1$, the interval $SA[sp_i, ep_i]$ that stores all text suffixes that are prefixed by the pattern suffix $P[i, p]$. The final interval $SA[sp_1, ep_1]$, if any, corresponds to all the suffixes that are prefixed by the pattern $P[1, p]$. Thus, $\mathsf{Count}(P)$ can be solved by returning the value $occ = ep_1 - sp_1 + 1$. Since each of the above steps requires the computation of two rank queries over the strings L, $O(p)$ ranks suffice to count the number of occurrences of any pattern P.

In practice, there are various implementations of Fmi, with their main differences in the way the rank-data structure built on $\mathsf{Bwt}(T)$ is compressed. The site Pizza&Chili[4] has several implementations of Fmi that mainly boil down to the following trick: $\mathsf{Bwt}(T)$ is split into blocks (of equal or variable length) and values of rank_c are precomputed for all block beginnings and all symbols $c \in \Sigma$. A query $\mathsf{rank}_c(L, i)$ is solved by summing up the answer available for the beginning of the block that contains $L[i]$, plus the rest of the occurrences of c in that block — they are obtained either by sequentially decompressing the block or by using a proper compressed data structure built on it (e.g. the Wavelet Tree of [6]). The former approach favors compression, the latter favors query speed.

2.2 The CSA family

These compressed indexes were introduced by Grossi and Vitter [7], who showed how to compactly represent the suffix array SA in $O(n \log \sigma)$ bits and still be able to access any of its entries efficiently. Their solution is based on a function Ψ, which is the inverse of the function LF introduced for Bwt:

$$\Psi(i) = \begin{cases} i' \textit{ such that } SA[i'] = SA[i] + 1 & (\textit{if } SA[i] < n) \\ i' \textit{ such that } SA[i'] = 1 & (\textit{if } SA[i] = n) \end{cases}$$

In other words, $\Psi(i)$ refers to the position in the suffix array of the text suffix that follows $SA[i]$ in T, namely, the text suffix which is one symbol shorter. Grossi and Vitter show how to hierarchically decompose the suffix array SA in order to obtain its succinct representation that still permits to perform searching operation on it. In their construction they exploit the *piecewise increasing*

[4] `http://pizzachili.dcc.uchile.cl/` or `http://pizzachili.di.unipi.it/`.

property of Ψ — namely that $\Psi(i) < \Psi(i+1)$ if $T[SA[i]] = T[SA[i+1]]$ — to represent the suffix array within $O(n \log \sigma)$ bits. The index must keep the original text in a non-compressed form to explicitly compare symbols of the text and the pattern during the searches.

This drawback has been overcome by two subsequent improvements. The first one, due to Sadakane [12], showed that the original text T can be replaced with a binary vector F such that $F[i] = 1$ iff the first symbol of the suffixes $SA[i-1]$ and $SA[i]$ differs. Since the suffixes in SA are lexicographically sorted, one can determine the first symbol of any suffix in constant time by just executing a $rank_1$ query on F. This fact, combined with the retrieval of Ψ's values in constant time, allows to compare any suffix with the searched pattern $P[1,p]$ in $O(p)$ time. Sadakane also provided an improved representation for Ψ achieving $nH_0(T)$ bits. Theoretically, the best variant of Csa is due to Grossi, Gupta and Vitter [6] who used some further structural properties of Ψ to get close to $nH_k(T)$ bits, still preserving the previous time complexity.

In practice, one of the best implementation of the Csa is the one proposed by Sadakane. It does not use the hierarchical decomposition, but orchestrates a compact representation of the function Ψ together with the backward search of the Fmi family.

2.3 Locate and Extract queries

Even though in the last years we have seen a proliferation of different compressed full text indexes [11, 3], Locate and Extract strategies remain almost unchanged since the very first proposals. At a high level, the idea consists in storing the relation between text positions and indexes in the suffix array of some sampled positions of the original text. Recall that Locate(P) requires to return the position $pos(i) = SA[i]$ of any suffix i, while Extract(l, r) extracts the substring $T[l, r]$. Locate is solved by starting from the ith suffix and by going backward or forward in the text by means of LF or Ψ functions. The procedure stops whenever a sampled position is found. Extract(l, r) is solved with a Fmi by starting from the sampled position closest to r and extracting the substring $T[l, r]$ bacwards symbol by symbol. The same strategy is used in Csa, except that we proceed forward starting from the sampled position closest to l.

```
Algorithm Fmi-Locate($i$)

$i' \leftarrow i, t \leftarrow 0$;
while $SA[i']$ is not explicitly stored do
    $i' \leftarrow LF(i')$;
    $t \leftarrow t + 1$;
return $SA[i'] + t$;
```

```
Algorithm Csa-Locate($i$)

$i' \leftarrow i, t \leftarrow 0$;
while $SA[i']$ is not explicitly stored do
    $i' \leftarrow \Psi(i')$;
    $t \leftarrow t + 1$;
return $SA[i'] - t$;
```

Fig. 1. Algorithms for locating the row with index i in Fmi and Csa.

The Locate algorithm of Fmi and (a practical implementation of) Csa is shown in Fig. 1. This algorithm is used to obtain the position in the text of the suffix that prefixes the ith row of $\mathcal{M}(T)$. As we said, the basic idea is to logically mark a suitable set of rows of $\mathcal{M}(T)$, and keep for each of them their position in T (that is, we store the corresponding SA values). Then, Locate(i) scans the text T backward using the LF-mapping, until a sampled row i' is found, and reports $SA[i'] + t$, where t is the number of backward steps used to find such i'. Csa works by going forward in the text by using Ψ function. To compute the positions of all occurrences of a pattern P, it is thus enough to call Locate(i) for all rows identified by the Count(P) operation.

The sampling rate of $\mathcal{M}(T)$'s rows, hereafter denoted by s_{SA}, is a crucial parameter that trades space for query time. Most Fmi and Csa implementations [3] sample all the $SA[i]$ that are a multiple of s_{SA}. This guarantees that at most s_{SA} steps of LF (or Ψ) suffice for locating the text position of any occurrence. The extra space required to store these positions is $O(\frac{n \log n}{s_{SA}})$ bits. In addition to these positions, we need to store a data structure that is able to, given a row, tell us if the row is sampled and, in that case, return its position in the text. An immediate solution resorts to a bitmap $B[1, n]$ whose ith entry is 1 iff the ith row is sampled. Then, all the sampled $SA[i]$s are stored contiguously in suffix array order, so that if $B[i] = 1$ then one finds the corresponding $SA[i]$ at position $rank_1(B, i)$ in that contiguous storage. In this case the extra space becomes $\frac{n \log n}{s_{SA}} + n + o(n)$ bits. There exist other more space efficient, but probably less practical, solutions. For example, one could resort to Minimal Perfect Hash functions [8]: we create a perfect hash function for the set of marked rows having their positions as satellite data. In this case the extra space is $O(\frac{n \log n}{s_{SA}})$ bits.

For our discussion it is more convenient to sample text positions instead of sampling rows of matrix $\mathcal{M}(T)$. Since there is one-to-one correspondence between $\mathcal{M}(T)$'s rows and text's positions, the problem of sampling positions is exactly the same as the problem of sampling rows.

The algorithm for Extract(l, r) resorts to a similar approach. Each query takes no more than $(r - l + s_{SA} + 1)$ rank queries: at most s_{SA} rank queries are required to reach r starting from the closest sampled position, and $r - l + 1$ queries are required to extract the substring $T[l, r]$ symbol by symbol.

The net result is that the space and time complexities of Fmi and Csa depend on the value s_{SA} and on the performance guaranteed by the data structure used to compute rank queries on the Bwt-string. The extra space required by the best (theoretical) data structures added to support Locate and Extract is bounded by $O((n \log n)/s_{SA})$ bits, which is $o(n)$ whenever s_{SA} is large enough.

3 Optimal distribution-aware Locate and Extract

The problem we address in this paper is defined as follows. We assume that, for any position j of the input text T, we know the probability $\Pr(j)$ that the position j will be located (i.e., the probability that we search a pattern P which is a prefix of the jth suffix of T). We have the user defined parameter s_{SA}

that specifies the amount of the space that we can use to store information regarding sampled positions. Our aim is that of identifying a optimal set of sampled positions $\mathcal{P}^*$ of size $K = n/s_A$ that allows us to minimize the expected time required to solve Locate queries. The expected time is given by

$$E[\mathcal{P}^*] = \sum_{j=1}^{n} \Pr(j) \cdot c(j, \mathcal{P}^*)$$

where $c(j, \mathcal{P}^*)$ is the cost (e.g., time or number of backward steps) required to reach the first sampled position in $\mathcal{P}^*$, say i, that precedes j in T. We call this problem the *distribution-aware optimal sampling problem.*

We observe that there are several different ways to define $c(j, \mathcal{P})$. For example, by setting $c(j, \mathcal{P}) = j - i$, we are simply counting the number of backward steps required to reach position i from j. This implies that we are implicitly assuming that all the backward steps have the same cost (in terms of CPU usage). Or one could refine the measure by setting $c(j, \mathcal{P})$ to be the sum of the *real cost* of the backward steps required to reach position i from j. To simplify the discussion we will use the first cost type.

We can address the problem of optimally sampling positions for Extract queries by changing the cost function $c()$. In this case, $\Pr(j)$ is the probability of extracting a substring that starts at position i and $c(j, \mathcal{P})$ is the cost of reaching position j starting from the first sampled position in $\mathcal{P}$ that follows j.

The discussion above implicitly assumes that we are dealing with a Fmi. As a Csa scans the text forward in Locate, and starts from the closest sampled position before the substring in Extract, the cost functions are used in the opposite way.

3.1 On finding a minimum weight K-link path over a DAG

The Distribution Optimal Sampling Problem can be reduced to the problem of *finding a minimum weight K-link path* [1, 13] in a particular Directed Acyclic Graph (DAG) $\mathcal{G}_R$. Given a weighted *DAG* $\mathcal{G}_R$ and a parameter K, the problem of finding a minimum weight K-link asks to identify a path from v_1 to v_{n+1} consisting of exactly K edges, whose cost is the minimum among all such paths.

In our solution the graph $\mathcal{G}_R$ has a vertex for each text position denoted $v_1, v_2, \ldots, v_n$ plus a dummy vertex v_{n+1} that marks the end of the text. For any pair of positions i and j such that $1 \leq i < j \leq n + 1$, we have an edge (v_i, v_j), whose cost $w(i, j)$ is equal to $\sum_{l=i}^{j-1} \Pr(l) \cdot (l - i)$. Intuitively, $w(i, j)$ accounts the part of expected cost for locating positions between i and $j - 1$, assuming that i is the only sampled position among them.

Efficient solutions for the problem of computing a minimum weight K-link path have been provided in literature [1, 13], if the DAG satisfies the so-called *concave Monge condition.*

Definition 1. *A weighted DAG $\mathcal{G}$ satisfies the concave Monge condition if*

$$w(i, j) + w(i + 1, j + 1) \leq w(i, j + 1) + w(i + 1, j)$$

holds for all $1 < i + 1 < j < n$.

Lemma 1. *The DAG $\mathcal{G}_R$ satisfies the concave Monge condition.*

The best known solutions for the computation of a minimum weight K-link path on a DAG satisfying the concave Monge condition are summarized in the following Theorems (Proved in [1] and [13]).

Theorem 1. *Given a DAG $\mathcal{G}$ satisfying the concave Monge condition and whose weights are integers, a minimum weight K-link path in $\mathcal{G}$, for any K, can be computed in $O(n \log U)$ time, where U is the maximum absolute value of the weights.*

Theorem 1 provides a weakly polynomial algorithm for the problem, which suffices for most of the interesting cases in practice. In fact, the probabilities of locating positions are typically frequencies derived by observing queries in a query-log of total length, say, m. Thus, we can label the edges of $\mathcal{G}_R$ with integral weights by appropriately multiplying each of these frequencies by m. In this way, the factor $\log U$ in the time complexity of Theorem 1 is $O(\log n + \log m)$. For completeness, we notice that there exists also a solution whose time complexity is independent of the weights.

Theorem 2. *Given a DAG $\mathcal{G}$ satisfying the concave Monge condition, a minimum weight K-link path in $\mathcal{G}$ can be computed in $O(nK^{\epsilon})$ time for $K = \Omega(\log n)$ and any fixed ϵ.*

4 Experiments

We implemented our Optimal sampling strategy by resorting to the algorithm of Theorem 1. The algorithm uses binary search to find an adjustment Q, such that $\mathcal{G}_R$ has a minimum weight path from v_1 to v_{n+1} with n/s_{SA} edges, when Q is added to all edge weights. That path is then a minimum-weight n/s_{SA}-link path in $\mathcal{G}_R$. For each candidate of Q, we search for the shortest and the longest minimum-weight paths. If n/s_{SA} falls between the extremes, then a n/s_{SA}-link path can be built by combining the shortest and the longest paths. As we use a simple $O(n \log n)$-time algorithm [9] for finding the minimum-weight paths, the overall time bound is $O(n \log n \log U)$. In practice, the bound is quite pessimistic.

In addition to our Optimal sampling strategy, we implemented three other strategies. Regular sampling is the classical strategy that samples one out of every s_{SA} positions. Greedy sampling selects n/s_{SA} text positions with the largest access probabilities. HalfGreedy first uses regular sampling with rate $2s_{SA}$, and then greedily selects $n/(2s_{SA})$ of the remaining positions.

Before presenting experimental results about these approaches, it is worth to compare the behavior of these strategies for their worst-case distribution with respect to our Optimal strategy. We present these considerations just for Locate, since similar bounds hold for Extract too.

The worst distribution for Regular is clearly the one in which there are n/s_{SA} positions with probability s_{SA}/n, while the others have chance 0 of being located.

Each of these positions follows one of the positions that have been sampled by Regular. Thus, the expected time to solve a locate is $O(s_{SA})$. Clearly, Optimal strategy achieves expected time equal to $O(1)$ by simply sampling all the positions having a positive probability.

Greedy is much worse. Consider the following distribution: each of the first n/s_{SA} positions of the text has probability $\frac{s_{SA}}{n-1}$, while the last n/s_{SA} positions have probability $\frac{s_{SA}}{n+1}$. Greedy wrongly selects the first n/s_{SA} positions, leaving a large part of the text unsampled. Thus its expected time is at least $\Theta(n - n/s_{SA})$.[5] On this distribution Optimal performs much better by sampling every other position with positive probability. In this way, it achieves an expected time of $O(1)$. As far as HalfGreedy is concerned, we observe that its worst expected time is $2s_{SA}$, and this is obtained by using a distribution which is a mixture of the ones used for Regular and Greedy.

The distributions above are specifically designed to highlight the drawbacks of the other strategies. In the remaining part of the section we experimentally compare these strategies on real datasets and with real query-distributions. As we will see, even in this practical setting, Optimal provides a less impressive but yet significant improvement. The different sampling strategies have been plugged in the compressed index RLCSA[6].

The implementation was written in C++ and compiled on g++ version 4.3.3. Experiments were done on a system with 32 gigabytes of memory and two quad-core Intel Xeon E5540 processors running at 2.53 GHz (we used only one core). The system was running Ubuntu 10.04 with Linux kernel 2.6.32. As the optimal sampling requires about $28n$ bytes of memory for a text of length n, we had to use another system with more memory for constructing some of the indexes.

We use two large datasets in the experiments. `Html Pages` is a 1.24-gigabyte set of web pages obtained by downloading the first 5 Yahoo! search results for all query terms with at least 100 occurrences in a MSN query log. `dblp` contains the DBLP Computer Science Bibliography[7] in XML format, for a total size of 813 megabytes. Both datasets were downloaded in March 2011.

The set of patterns to be searched for `Html Pages` was constructed by selecting all terms from the MSN query log and by removing stop words. Each pattern was associated the number of its occurrences in the query log. For `dblp`, we built a synthetic set of patterns obtained by selecting all author names and all non-stop word terms appearing in paper titles. Each term has associated a number of occurrences that is taken from the previous set of patterns. From the two sets of patterns, we computed the access frequency of each position of the text as follows. For position i, we set its frequency to be the sum of the number of occurrences of those patterns that are prefixes of suffix $T[i, n]$. The frequencies of all positions (suffixes) are plotted in Figure 2 after they have been sorted decreasingly.

[5] Notice that at least $n - 2n/s_{SA}$ steps are required to locate each of the last n/s_{SA} positions.

[6] Available at `http://www.cs.helsinki.fi/group/suds/rlcsa/`.

[7] `http://dblp.uni-trier.de/db/`

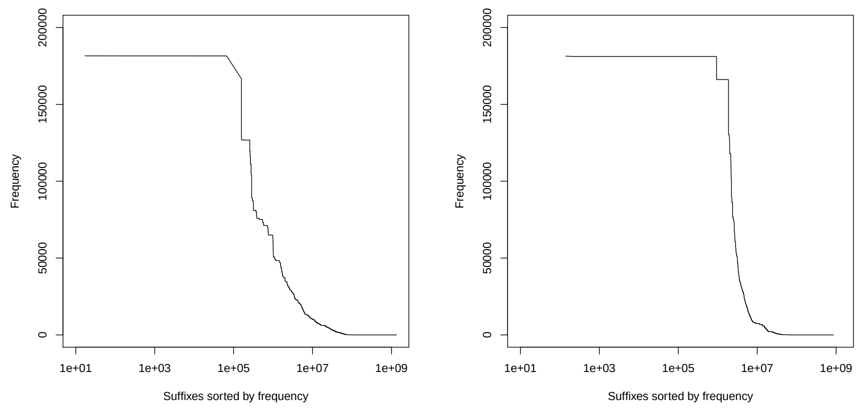

Fig. 2. Distributions of position access frequencies for `Html Pages` (left) and `dblp` (right).

	`Html Pages`				`dblp`			
s_{SA}	Regular	HalfGreedy	Greedy	Optimal	Regular	HalfGreedy	Greedy	Optimal
16	7.5	0.7	0.2	0.1	7.5	0.15	0.005	0.004
32	15.5	4.7	3.0	0.9	15.5	1.2	0.6	0.3
64	31.5	13.9	42.5	4.2	31.5	7.2	4.4	1.9
128	63.5	43.0	104.2	14.7	63.5	26.0	31.4	8.9

Table 1. Average number of LF or Ψ steps required to locate pattern occurrences depending on value of s_{SA} and sampling strategy in use.

For our experiments, we built RLCSA with $s_{SA} = \{16, 32, 64, 128\}$ for both datasets. We searched for $10,000$ patterns randomly selected accordingly to the previously constructed query distributions for a total of about 187.3 million located positions for `Html Pages` and about 276.6 million positions for `dblp`. We also extracted snippets of length 16, 32, and 64 from $1,000,000$ randomly selected positions according to position frequencies. In addition to measuring the number of located positions and extracted characters per second (Figure 3), we also determined the average number of LF/Ψ steps required to find a sampled position (see Table 1).

All distribution-aware strategies performed similarly in Locate with low values of s_{SA}, being almost 8 times faster than Regular. This behavior is due to the fact that, for small values of s_{SA}, the distribution-aware strategies are able to sample most of the positions with positive frequencies. With larger s_{SA}, Optimal retained its lead, while Greedy and HalfGreedy became worse. The highest gain of Optimal w.r.t. to Regular is obtained for $s_{SA} = 32$ (factors 8.4 and 10.1 for `Html Pages` and `dblp` respectively) while the lowest is obtained for $s_{SA} = 128$ (factors 5.6 and 4.4 respectively). In Extract, Optimal is roughly twice faster than the other strategies. The gain is limited due to the fact that, in any case, Ex-

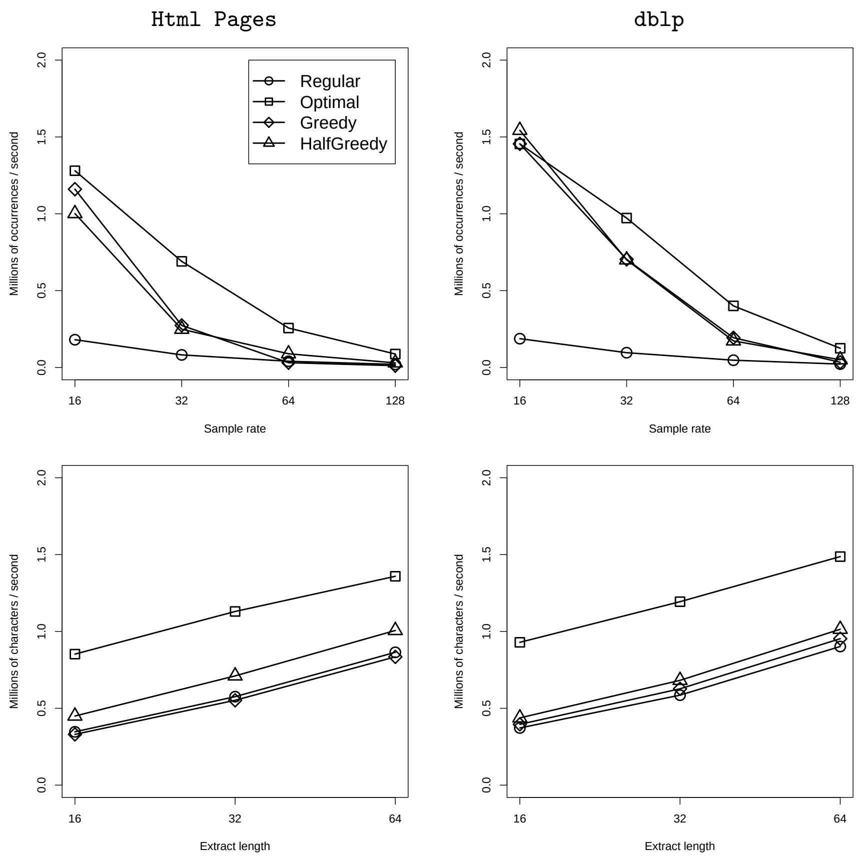

Fig. 3. Experimental results for `Html Pages` (left) and `dblp` (right). Locate performance (top) and Extract performance with $s_{SA} = 128$ (bottom).

tract requires c steps to extract c symbols after finding a sampled position. We observed an odd result for $s_{SA} = 16$ on `dblp` where HalfGreedy is slightly better than Optimal. This is due to the last step of Locate algorithm that checks if the current row index is sampled or not. Whenever the average number of LF/Ψ steps is close to 0, the cost of this step becomes dominant. This step is usually performed by resorting to rank/select queries over a bit vector. In the current implementation of RLCSA, the time cost of this operation may slightly vary depending on the underlying bit vector. In the final version of the paper, we will investigate the possibility of designing more suitable solutions for this step as well as compare other implementations of compressed indexes.

5 Future work

In this paper we addressed the problem of designing distribution-aware compressed full-text indexes. We showed that an optimal selection of positions can be computed efficiently in time and space when the distribution of subsequent queries is known beforehand. The advantage at query time is between 4–10 times better than the classical approach to Locate. In case of Extract the advantage is reduced to 2.

An interesting open problem asks for designing distribution-aware compressed indexes that are able to self-adapt themselves to the unknown distribution of queries. We believe that the field of compressed data structures could benefit a lot by following this line of research.

References

1. A. Aggarwal, B. Schieber, and T. Tokuyama. Finding a minimum-weight k-link path graphs with the concae monge property and applications. *Discrete & Computational Geometry*, 12:263–280, 1994.
2. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
3. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.
4. P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.
5. P. Ferragina and G. Manzini. On compressing the textual web. In *WSDM*, pages 391–400, 2010.
6. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
7. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. of the 32nd ACM Symposium on Theory of Computing*, pages 397–406, 2000.
8. T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *STACS*, pages 317–326, 2001.
9. D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.
10. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
11. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.
12. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
13. B. Schieber. Computing a minimum weight k-link path in graphs with the concave monge property. *J. Algorithms*, 29(2):204–222, 1998.
14. F. Silvestri. Mining query logs: Turning search usage data into knowledge. *Foundations and Trends in Information Retrieval*, 4(1-2):1–174, 2010.