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

Distribution-aware compressed full-text indexes

Paolo Ferragina · Jouni Sirén · Rossano Venturini

the date of receipt and acceptance should be inserted later

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 properly designed Directed Acyclic Graph. Interestingly enough, our solution can be used with any compressed index based on the Burrows-Wheeler transform. Our experiments compare this optimal strategy with several other known approaches, showing its effectiveness in practice.

Keywords Full-text Indexing · Compressed Full-text Indexes · Succinct Data Structures · Dynamic Programming

1 Introduction

String processing and searching tasks are at the core of modern web search, IR, data base and data mining applications. Most text operations 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

This work was partially supported by MIUR of Italy under projects PRIN ARS Technomedia 2012 and FIRB Linguistica 2006, the Midas EU Project, Grant Agreement no. 318786, the InGeoCloudS EU Project, Grant Agreement no. 297300, the Finnish Doctoral Programme in Computational Sciences, Academy of Finland, Helsinki Institute for Information Technology, and the Nokia Foundation.

Paolo Ferragina
Dipartimento di Informatica, University of Pisa, L.go B. Pontecorvo 3, 56127 Pisa, Italy.

Jouni Sirén
Department of Computer Science, University of Helsinki, Finland.

Rossano Venturini
Dipartimento di Informatica, University of Pisa, L.go B. Pontecorvo 3, 56127 Pisa, Italy.
E-mail: ferragina@di.unipi.it, jltsiren@cs.helsinki.fi, rossano@di.unipi.it

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 to support faster operations. This dichotomy was addressed starting from the year 2000 [6,11], due to various achievements which showed how to relate Information Theory with String-Matching concepts, in a way that index regularities, which show up when data is compressible, are discovered and exploited to reduce index occupancy without impairing query efficiency (see the surveys [5,16] 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 in efficient time. Due to this latter feature, these data structures are sometime called *self*-indexes.

As experimentally shown in [5,7], 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 represented by Locate queries which ask for the text positions of the pattern occurrences. Consequently they are between two and three orders of magnitude slower than what is achievable with the classic Suffix Array data structure. Also the Extract operation, which returns a decompressed portion of the indexed text, is quite slow compared with other compression methods for sufficiently long substrings. In addition, for Locate and Extract, these indexes need to store some extra information, inducing a trade-off between space and time efficiency: the larger this extra space, the faster is the resulting index. At a 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 the one hand, it is guaranteed that each occurrence of the searched pattern is located in at most $s_{SA}-1$ 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 [2,3,5,14–16], the above sampling strategy remains almost unchanged since the very first proposal. This strategy implicitly assumes all text positions to have uniform probability of being located or extracted. But uniform distributions are very 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 [19] and references therein).

Given these premises, in this paper we address the following question: Is it possible to build a distribution-aware compressed self-index which optimizes the expected query-time by occupying a given space? Given the distribution of the subsequent queries and a bound on the space occupancy, the goal is to find a sampling strategy that induces the fixed 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 properly designed 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 many known compressed indexes without changing their Locate/Extract algorithms.

In the experimental section (Section 4) we compare our optimal sampling strategy against several other different strategies over two large datasets of HTML pages and XML documents. The experiments have been performed by using RLCSA, which is an implementation of the Compressed Suffix Array (CSA). Although restricted to this single index, our experiments will quantify some measures that are independent on the particular implementation of compressed indexes in use, and thus can be adopted for other compressed indexes. Overall we show that our optimal sampling is from 4 to 36 times faster than the uniform sampling. We also compare our optimal strategy against two heuristic approaches, showing that ours is up to a factor 9 faster. One of them is the obvious strategy that "caches" the most probably accessed positions. This heuristic has poor results both in theory and in practice because it does not fully 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 in more detail in Section 4, where we quantify also the impact of the various heuristics by performing a significant set of experiments.

2 Background and Related Work

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 which lie at the crossing point of three distinct fields— data compression, data structures and databases (see e.g. [5,6,11,16]). Most of these indexes can be classified into two families— namely, FM-indexes (shortly, FMI) and Compressed Suffix Arrays (shortly, CSA)— and achieve efficient query times and space close to the one achievable by widely used compressors, like gzip or bzip2. 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 a text $T[1,n]$ drawn from an alphabet of size σ. This bound is much appealing because it can be sublinear in $n \log \sigma$, for highly compressible texts. We recall that $nH_k(T)$ is the classical Information-Theoretic lower bound to the storage complexity of T by means of any k-th order compressor. In addition to being compressed, the index is able to efficiently support the following three operations:

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

2.1 The FM-index family

These compressed indexes were introduced by Ferragina and Manzini in [6], who discovered a way to orchestrate in efficient time and space the relation that exists be-

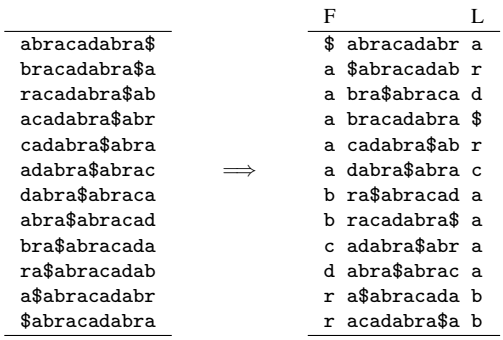

Fig. 1 Example of Burrows-Wheeler transform for the string $T = \texttt{abracadabra\$}$. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the column $L = \texttt{ard\$rcaaaabb}$.

tween the suffix array data structure and the *Burrows-Wheeler Transform* [4] (shortly, BWT). The BWT is a reversible transformation that permutes the symbols of the input string T into a new string $L = \mathsf{BWT}(T)$ which is easier to compress. It can be computed in three steps (see Figure 1):

1. append a special symbol \$ smaller than any other symbol of Σ to the end of T (for the rest of this paper, we assume that $T[n] = \$$);
2. form a *conceptual* matrix $\mathscr{M}(T)$ whose rows are the cyclic rotations of string $T\$$ in lexicographic order;
3. set string $\mathsf{BWT}(T)$ to be the last column L of the sorted matrix $\mathscr{M}(T)$.

Every column of $\mathscr{M}(T)$, including the transformed string L, is a permutation of $T\$$. In particular the first column of $\mathscr{M}(T)$, call it F, is obtained by lexicographically sorting the symbols of $T\$$ (or, equally, the symbols of L). Note that the sorting of the rows of $\mathscr{M}(T)$ is essentially equal to the sorting of the suffixes of T, because of the presence of the special symbol \$. This shows that: (1) symbols preceding the same substring (*context*) in T are grouped together in L, and thus give rise to clusters of nearly identical symbols; (2) there is an obvious relation between $\mathscr{M}(T)$ and SA. Property 1 is the key for devising modern data compressors, Property 2 is crucial for designing compressed indexes and, additionally, suggests a way to compute the BWT through the construction of the suffix array of T: $L[1] = T[n-1]$ and, for any $1 < i \leq n$, set $L[i] = T[SA[i]-1]$, where $T[0] = T[n] = \$$.

Burrows and Wheeler [4] devised two properties which are crucial for the invertibility of the BWT:

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

As a result, 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 [6]: 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 $\mathscr{M}(T)$ is $\$T$, it can be stated that $T[n-1] = L[1]$ and, in general, $T[n-i] = L[LF^i(1)]$, for $i = 1, \ldots, n-1$.

Starting from these basic properties, Ferragina and Manzini [6] proposed a way to combine the compressibility of the BWT with the indexing power of the suffix array. In particular, they have shown that searching operations over 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 $\mathsf{BWT}(T)$, which supports efficient rank queries, a small array $C[c]$ which tells how many symbols smaller than c appear in T (this takes $O(\sigma \log n)$ bits), and the so called *backward search* algorithm, which carefully orchestrates the former two data structures in order to implement efficiently the Count query. 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 $\mathscr{M}(T)$ that are prefixed by P. This is done by maintaining, inductively for $i = p, p-1, \ldots, 1$, the interval $SA[sp_i, ep_i]$ that stores all text suffixes which are prefixed by the pattern suffix $P[i,p]$.

Initially $i = p$, and $SA[sp_p, ep_p]$ corresponds to all suffixes that are prefixed by the last symbol $P[p]$. Hence we can set $sp_p = C[P[p]] + 1$ and $ep_p = C[P[p]+1]$. In any later step, the algorithm has already computed $SA[sp_{i+1}, ep_{i+1}]$. Thus we can derive the next interval $SA[sp_i, ep_i]$ by setting $sp_i = C[P[i]] + \mathsf{rank}_{P[i]}(L, sp_{i+1}-1) + 1$ and $ep_i = C[P[i]] + \mathsf{rank}_{P[i]}(L, ep_{i+1})$. These two computations are actually mapping (via LF) the first and last occurrences (if any) of symbol $P[i]$ in the substring $L[sp_{i+1}, ep_{i+1}]$ to their corresponding occurrences in F. (Indeed, [6] showed that any LF computation boils down to a rank query on L.) As a result, the backward-search algorithm requires to solve $2(p-1)$ rank queries on $L = \mathsf{BWT}(T)$ in order to find out the (possibly empty) range $SA[sp, ep]$ of text suffixes prefixed by 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 string L, $O(p)$ ranks suffice to count the number of occurrences of any pattern P in the indexed text T.

There are various implementations of FMI, whose engineering choices mainly differ in the way the rank-data structure built on $\mathsf{BWT}(T)$ is compressed and scales with the alphabet size of the indexed text. The site Pizza&Chili[1] contains several implementations for 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 [10]). The former approach favors compression, the latter favors query speed.

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

2.2 The CSA family

These compressed indexes were introduced by Grossi and Vitter [11], who showed how to compactly represent the suffix array SA in $O(n)$ 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:

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

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 a succinct representation that still permits performing searching operation on it. In their construction they exploit the *piecewise increasing property* of Ψ — namely, if $T[SA[i]] = T[SA[i+1]]$, then $\Psi(i) < \Psi(i+1)$ — to represent the suffix array in $O(n \log \sigma)$ bits. The index must indeed keep the original text available in a non-compressed form to explicitly compare symbols of the text and the pattern during the searches.

This drawback was overcome by two subsequent improvements. The first one, due to Sadakane [17], 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.[2] 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 [10] who devised some further structural properties of Ψ that allow to come close to $nH_k(T)$ bits, still preserving the previous time complexity.

In practice, one of the best implementations of the CSA is the one proposed by Sadakane that actually does not use the hierarchical decomposition mentioned above, 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 [2,3,5,14–16], Locate and Extract strategies remain almost unchanged since the very first proposals. The indexes of both families need some extra information about the underlying suffix array which remarkably impact on their space occupancy. At a high level, the idea consists in storing the relation between text positions and indexes in the suffix array for some sampled positions of the original text. Recall that $\mathsf{Locate}(P)$ must be able to return the value $SA[i]$ for any row i, while $\mathsf{Extract}(l,r)$

[2] Binary vector F is essentially an encoding of array C in FMI.

Algorithm FMI-Locate(i)	**Algorithm** CSA-Locate(i)
$i' \leftarrow i, t \leftarrow 0$;	$i' \leftarrow i, t \leftarrow 0$;
while $SA[i']$ is not explicitly stored **do**	**while** $SA[i']$ is not explicitly stored **do**
$\quad i' \leftarrow LF(i')$;	$\quad i' \leftarrow \Psi(i')$;
$\quad t \leftarrow t+1$;	$\quad t \leftarrow t+1$;
return $SA[i'] + t$;	**return** $SA[i'] - t$;

Fig. 2 Algorithms for locating the position in T of the row with index i in FMI and CSA.

extracts substring $T[l,r]$. Locate is solved by starting from the i-th 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. With a FMI, Extract(l,r) is solved by starting from the sampled position closest to r, and by extracting the substring $T[l,r]$ symbol by symbol by going backward in the text. The same strategy is used in CSA, except that we proceed forward starting from the sampled position closest to l.

The Locate algorithm of FMI and (a practical implementation of) CSA is shown in Fig. 2. This algorithm is used to obtain the position in the text of the suffix that prefixes the i-th row of $\mathscr{M}(T)$. As we said, the basic idea is to logically mark a suitable set of rows of $\mathscr{M}(T)$, 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 then 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 means of Ψ function. To compute the positions of all occurrences of a pattern P, it is thus enough to call Locate(i) for each of the rows identified by the Count(P) operation.

The sampling rate of $\mathscr{M}(T)$'s rows, hereafter denoted by s_{SA}, is a crucial parameter that trades space for query time. Most FMI and CSA implementations [5] sample all the entries $SA[i]$ that are a multiple of s_{SA}. This guarantees that at most $s_{SA}-1$ 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 i-th entry is 1 iff the i-th 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 [12]: 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 discussions it will be more convenient to sample text positions instead of sampling rows of matrix $\mathscr{M}(T)$. Since there is one-to-one correspondence between $\mathscr{M}(T)$'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 $(r-l+s_{SA})$ rank queries: at most $s_{SA}-1$ 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

In this paper 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. We have the user defined parameter s_{SA} that specifies the amount of space that we can use to store information regarding sampled positions. Our aim is to identify an optimal set of sampled positions $\mathscr{P}^*$ 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

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

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

We observe that there could be several different ways to define $c(j,\mathscr{P})$. For example, by setting $c(j,\mathscr{P}) = j-i$, 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 $c(j,\mathscr{P})$ 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 notice that by simply changing the definition of the above cost function $c()$, we can also address the problem of optimally sampling positions for Extract queries. In this case $\Pr(j)$ is the probability of extracting a substring that ends at position j and $c(j,\mathscr{P})$ is the cost of reaching position j starting from the first sampled position in $\mathscr{P}$ 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-aware Optimal Sampling Problem can be reduced to *finding a minimum weight K-link path* [1,18] in a properly defined Directed Acyclic Graph (DAG) $\mathscr{G}_R$. The graph $\mathscr{G}_R$ 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 for the part of expected cost for locating positions between i and $j-1$, assuming that among them only i is a sampled position.

Given the weighted *DAG* $\mathscr{G}_R$ and a parameter K, the problem of finding a minimum weight K-link path 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. Very efficient solutions for this problem do exist in the literature [1, 18], whenever the DAG satisfies the so-called *concave Monge condition*.

Definition 1 A weighted DAG $\mathscr{G}$ 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 $1 < i+1 < j < n$.

The following Lemma shows that our DAG $\mathscr{G}_R$ satisfies the concave Monge condition.

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

Proof For the cost function in $\mathscr{G}_R$, we have that

$$w(i,j+1) - w(i,j) = \sum_{i \leq l \leq j} \Pr(l) \cdot (l-i) - \sum_{i \leq l < j} \Pr(l) \cdot (l-i) = \Pr(j) \cdot (j-i).$$

Since the concave Monge condition can be rewritten as

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

we have that

$$w(i+1,j+1) - w(i+1,j) = \Pr(j) \cdot (j-i-1) \leq w(i,j+1) - w(i,j) = \Pr(j) \cdot (j-i).$$

□

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 Theorem proved in [1].

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

Thus Theorem 1 provides a weakly polynomial algorithm for the problem which suffices for most of the practical interesting cases. The probabilities for locating text positions are typically derived from pattern frequencies in some query log. The locating probability of suffix $T[j,n]$ is set to be the normalized sum of the numbers of occurrences for those patterns that match the suffix. If we skip the normalization, we get integer frequencies $f(j)$ for text positions, and hence integral weights for edges. For completeness, we point out that there are also solutions whose time complexities are independent of the weights. The current best result is the one presented in [18] that is summarized in the following theorem.

Theorem 2 *Given a DAG $\mathscr{G}$ satisfying the concave Monge condition, a minimum weight K-link path in $\mathscr{G}$ can be computed in $O(n \cdot 2^{O(\sqrt{\log K \log\log n})})$ time for any $K = \Omega(\log n)$,*

3.2 Practical computation of Optimal sampling

For our experiments, we identified the optimal sample of positions by resorting to the idea behind the algorithm of Theorem 1 (see [1]). Namely, given the DAG $\mathscr{G}_R$ and an integral value q, let $\mathscr{G}_R(q)$ denote the adjusted DAG with exactly the same sets of vertices and edges as $\mathscr{G}_R$, in which each edge (u,v) has weight $w(u,v)+q$. Observe that the Monge Condition remains satisfied in any graph $\mathscr{G}_R(q)$.

The algorithm uses binary search to find a value q^*, such that the adjusted graph $\mathscr{G}_R(q^*)$ has a minimum weight path from v_1 to v_{n+1} with K edges.[3] Such an integral value q^* always exists and belongs to the interval $[-3U, 3U]$, where U is the maximum absolute value of the weights in $\mathscr{G}_R$ [1]. Moreover, it can be proven that a minimum weight path with K edges in $\mathscr{G}_R(q^*)$ is a minimum-weight K-link path in the original graph. These considerations imply that the value of q^* can be binary searched by considering integral candidates in $[-3U, 3U]$. For each candidate q, we compute the shortest and the longest minimum-weight paths in the graph $\mathscr{G}_R(q)$. If these paths are respectively shorter than and longer than K, then a K-link path can be built by appropriately combining these two paths. The computation of these paths can be done in linear time over DAG satisfying the concave Monge condition (see [9,21] and references therein). Thus, the overall time complexity of the algorithm is $O(n \log U)$.

Our implementation is $O(\log n)$ times slower in the worst case, as we resort to the simpler $O(n \log n)$-time algorithm by Hirschberg and Larmore [13] for the latter task of computing the shortest and longest paths in $\mathscr{G}_R(q)$. In practice, all factors in the time bound $O(n \log U \log n)$ are quite pessimistic, and we have three main reasons for expecting significantly better performance in practice.

First, the optimal sampling never contains text positions with frequency 0. Hence we have to consider only nodes with positive frequencies, replacing the factor n with the number of those positions.

Second, the value q^* is typically much smaller than U, so using doubling search instead of binary search replaces the $\log U$ factor with $\log q^*$. Assuming that the largest (integer) frequency of a text position is F, then the weight of edge $(i, i+\ell)$ of $\mathscr{G}_R(q)$ is at most $F\ell(\ell-1)/2+q$. For $q = 2Fn^2/K^2$, the weight of any K-link path must be at least $2Fn^2/K$, while the weight of the $K/2$-link path visiting all nodes v_i with i divisible by $2n/K$ is at most $2Fn^2/K$. As larger q leads to shorter minimum-weight paths in $\mathscr{G}_R(q)$, it must be that $q^* \leq 2Fn^2/K^2$. On the other hand, if the frequencies are relatively evenly distributed in the text, then $U = w(1, n+1) = \Theta(\bar{F}n^2)$, where $\bar{F}$ is the average frequency of querying a text position.

The third reason depends on the behavior of Hirschberg and Larmore's algorithm. Let $W(v_i)$ be the weight of a minimum-weight path from node v_1 to node v_i. The

[3] Recall that in our problem K is equal to n/s_{SA}.

algorithm of Hirschberg and Larmore processes nodes form left to right. At each step, it maintains a set of active nodes. A previously processed node v_i is active, if it is strictly better than any other processed node for reaching some node v_j that has not yet been processed. More formally, node v_i is active, if there exists a non-processed node v_j, such that $W(v_i) + w(v_i, v_j) + q < W(v_{i'}) + w(v_{i'}, v_j) + q$, for any node $v_{i'}$ with $i' < i$. When processing a new node, the algorithm uses several tests to determine which nodes remain active, and whether the current node is added to the active nodes. The $\log n$ factor comes from using binary search to determine which is the last active node that should remain active. Again, using doubling search can reduce the $\log n$ factor in many practical cases.

While the algorithm for finding optimal samples is sequential in its nature, a small change to the problem makes the algorithm easy to parallelize. Assume that we want to find an optimal set of samples, with the restriction that the set should always contain one out of s' text positions with a positive frequency. Then the minimum-weight K-link path from v_1 to v_{n+1} must contain the vertices corresponding to these text positions. Let v_i and v_j, where $i < j$, be two of these vertices. When looking for the minimum-weight paths in graph $\mathscr{G}_R(q)$, we can find the part of the path from v_i to v_j independently of the rest of the graph (i.e., all the nodes before v_i and after v_j). This allows us to use up to n'/s' parallel threads in computing the minimum-weight paths, where n' is the number of text positions with a positive frequency. Additionally, as long as the number of non-optimal samples n'/s' is negligible compared to the total number of samples n/s_{SA}, the resulting set of samples should be almost as good as an optimal set. In the next section we compare the time required by both sequential and parallel optimal sampling showing that the construction using the latter is significantly faster.

The algorithms discussed above assume constant time access to any required cost $w(\cdot,\cdot)$. However, since there are $\Theta(n^2)$ edges in $\mathscr{G}_R$, we cannot compute and store all its weights beforehand. Instead, we have to be able to compute on-the-fly any required cost in constant time. This is obtained by precomputing two arrays of length $n+1$. The first array stores edge weights $w(i, n+1)$ for all nodes i, while the second stores the sums of frequencies $F[i] = \sum_{j=i}^{n} f(j)$ for all i. By using these arrays, we compute edge weights as $w(i, j) = w(i, n+1) - w(j, n+1) - (j-i)F[j]$. While the weights of long edges can exceed $2^{64} - 1$, storing the lowest 64 bits is enough in practice. As the weight of a minimum weight path in $\mathscr{G}_R(q^*)$ is $O(Fn^2/K)$, the necessary computations can be done in 64 bits, for realistic values of F, n, and K.

4 Experiments

For the experimental evaluation we implemented the following four different sampling strategies.

- Uniform sampling is the classical strategy that samples every s_{SA}th position
- Optimal sampling selects an optimal set of n/s_{SA} positions as described in the previous section.
- Greedy sampling selects n/s_{SA} text positions with the largest access probability.

– HalfGreedy sampling uses first the Uniform sampling with rate $2s_{SA}$, and then greedily selects $n/(2s_{SA})$ of the remaining positions.

Before presenting experimental results on real datasets, it is worth comparing the behavior of these strategies with their worst-case distributions with respect to our Optimal strategy. We present these considerations just for Locate, since Extract gets similar performance.

The worst distribution for Uniform 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 precedes one of the positions that have been sampled by Uniform. Thus, the expected time to solve Locate is $O(s_{SA})$. Clearly the 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 $\Omega(n - n/s_{SA})$.[4] On this distribution Optimal performs much better by sampling every other position with a 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 Uniform 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` [15].[5]

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. Only one core was used for the queries and for finding the optimal samples, while the suffix array construction algorithms and parallel sampling used all 8 cores. The system was running Ubuntu 12.04 with Linux kernel 3.2.0.

We use two large datasets in the experiments. `Html Pages` is a 1.24-gigabyte set of web pages obtained by downloading the first five Yahoo! search results for all query terms with at least 100 occurrences in an MSN query log. `Dblp` contains the DBLP Computer Science Bibliography[6] in XML format, for a total size of 813 megabytes. Both datasets were downloaded in March 2011 and are available at `http://www.cs.helsinki.fi/group/suds/rlcsa/`.

The set of patterns to be searched for `Html Pages` was constructed by selecting all the terms but stop words in a query log from an anonymous web search engine with a total of about 439 million of queries. The final dataset consists of 29,175,101

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

[5] Available at `http://www.cs.helsinki.fi/group/suds/rlcsa/`. January 2013 version includes the code used in the experiments.

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

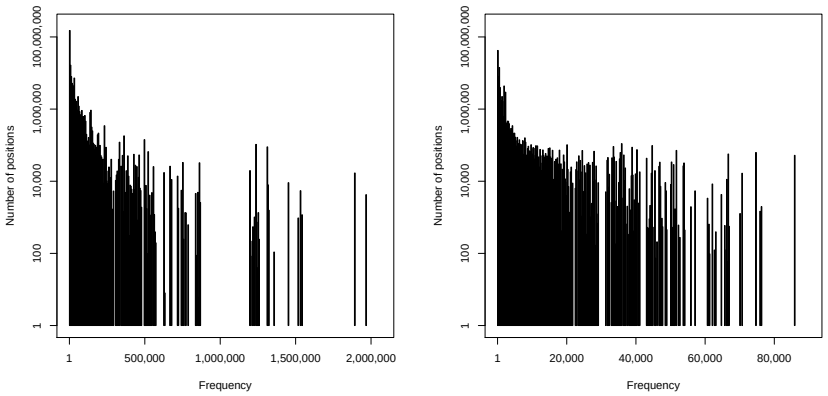

Fig. 3 Distributions of position access frequencies for `Html Pages` (left) and `Dblp` (right). Axis x reports the access frequency while axis y tells us the number of positions that have that frequency. Positions with access frequency equal to 0 are not reported.

distinct terms, and the frequency of a pattern was set to be the number of its occurrences in the query log. The set of patterns for `Dblp` was obtained by crawling entries from `www.mendeley.com` corresponding to computer science papers in September 2011. From a total of $519,545$ papers, we extracted a set of $877,592$ terms, consisting of all author names and all non-stop word terms appearing in paper titles. For each pattern, we computed the sum of the number of users having a paper in their library for those papers, where the term appears in the title or as an author. This sum was then used as the frequency of the pattern.

From a set of patterns, we computed the access frequency of each position of the text as follows. For position j, we set its frequency $f(j)$ to be the sum of pattern frequencies of those patterns that are prefixes of suffix $T[j,n]$. The frequencies of all positions (suffixes) are plotted in Figure 3: axis x reports the access frequency while axis y tells us the number of positions that have that frequency. Positions with access frequency equal to 0 are not reported.

For our experiments, we built `RLCSA` with $s_{SA} = \{8,16,32,64,128\}$ for both datasets. For computing the optimal sampling we implemented two different algorithms: Sequential and Parallel. The former finds an optimal set of samples by using the sequential algorithm described in the previous section. Parallel, instead, implements the (potentially suboptimal) parallel algorithm discussed at the end of the previous section by (pre)sampling one out of n'/τ^2 text positions with a positive frequency, where τ is the number of threads used. In our experiments these algorithms obtained two samplings having the same performances at query time, while the latter algorithm is sensibly faster in construction. Table 1 shows the time and space requirements of index construction and sampling selection. The suffix array construction algorithm used in `RLCSA` is a prefix-doubling algorithm that supports large alphabets [20]. As a comparison, we have included the results for suffix array construction with `libdivsufsort` 2.0.1 by Yuta Mori[7] that is currently considered

[7] `https://code.google.com/p/libdivsufsort/`

Html Pages

	RLCSA		Optimal			
	Construction		Sequential		Parallel	
s_{SA}	Time	Space	Time	Space	Time	Space
8	8.91	18.23	6.38	7.75	1.21	7.75
16	8.27	18.25	16.60	7.41	2.77	7.41
32	7.67	18.25	35.60	7.41	5.37	7.41
64	7.97	18.21	56.55	7.41	8.78	7.41
128	7.93	18.27	62.18	7.41	9.30	7.41
SA Construction			2.74	6.21		

Dblp

	RLCSA		Optimal			
	Construction		Sequential		Parallel	
s_{SA}	Time	Space	Time	Space	Time	Space
8	3.28	11.05	1.42	4.24	0.36	4.24
16	2.96	10.99	3.54	4.24	0.67	4.24
32	2.91	11.10	7.77	4.24	1.26	4.24
64	2.76	11.26	13.80	4.24	2.29	4.24
128	2.72	11.10	22.18	4.24	2.97	4.24
SA Construction			1.91	3.97		

Table 1 The table reports time and memory usage required to construct the RLCSA index and to compute the Optimal sampling with Sequential and Parallel algorithms by varying the value of s_{SA}. Times are in minutes and memory usage in gigabytes. As a comparison, we include time and memory usage of the plain suffix array construction with libdivsufsort 2.0.1.

	Html Pages			Dblp		
s_{SA}	Standard	Uniform	Optimal	Standard	Uniform	Optimal
8	1948	1977	2293	948	967	1090
16	1309	1401	1678	552	611	787
32	988	1123	1370	354	440	597
64	827	989	1216	255	357	502
128	747	924	1139	205	317	454

Table 2 Index sizes in megabytes for RLCSA with various sample rates s_{SA} and sampling strategies. Greedy and HalfGreedy strategies require the same space as Optimal, with the exception that HalfGreedy requires 1168 megabytes on Dblp with $s_{SA} = 8$. Standard is the same as Uniform, except that the bit vector used to mark sampled rows is gap encoded, making it slower and smaller than the succinct bit vectors used with other strategies. RLCSA without any samples takes 667 megabytes for Html Pages and 156 megabytes for Dblp.

to be the fastest implementation of a suffix array construction algorithm. Both construction algorithms and Parallel sampling algorithm were parallelized with OpenMP, while Sequential sampling was done with a single thread. Observe that with low values of s_{SA}, Parallel sampling is faster than index construction, while larger values led to similar time usage. Space requirements of sampling is $32n'$ bytes, where n' is the number of text positions with a positive frequency. In practice, this became 5–6 times larger than the text size which is an overhead similar to suffix array construction with libdivsufsort.

Html Pages

s_{SA}	Uniform	Optimal	Greedy	HalfGreedy
8	3.50	0.000021	0.000022	0.044
16	7.50	0.089	0.40	0.89
32	15.50	1.14	7.76	4.45
64	31.50	5.16	27.79	16.10
128	63.49	16.61	116.29	48.97

Dblp

s_{SA}	Uniform	Optimal	Greedy	HalfGreedy
8	3.50	0	0	0.044
16	7.50	0.052	0.060	0.89
32	15.51	1.02	4.27	4.45
64	31.51	4.67	12.80	16.10
128	63.51	14.62	41.06	48.97

Table 3 Average number of LF or Ψ steps required to locate pattern occurrences depending on value of s_{SA} and sampling strategy in use.

The size of the index with various sample rates s_{SA} and sampling strategies can be seen in Table 2. Non-uniform samples require more space than uniform ones for two main reasons: 1) we have to use an extra bit vector to mark the sampled text positions for Extract; 2) sampled positions are no longer multiples of s_{SA} and, thus, they are stored by using $\log n$ bits instead of $\log(n/s_{SA})$ bits.

We searched for $10,000$ patterns randomly selected accordingly to the previously constructed query distributions, for a total of about 122 million located positions for Html Pages, and about 84 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 4), we also determined the average number of LF/Ψ steps required to find a sampled position (see Table 3). The results reported in this table are interesting, because this measure is independent with respect to the implementation details of the underlying compressed index. As the performance of Sequential and Parallel is essentially the same[8], both are denoted as Optimal in the query experiments.

All distribution-aware strategies performed similarly in Locate with the lowest value of s_{SA} (i.e., $s_{SA} = 8$), being up to 22 times faster than Uniform. 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. However, just by moving to $s_{SA} = 16$, we observe that Optimal starts to become significantly better than Greedy and HalfGreedy. The highest gain for Optimal with respect to Uniform is obtained for $s_{SA} = 16$ (factors 34.3 and 35.7 for Html Pages and Dblp, respectively) while the lowest is obtained for $s_{SA} = 128$ (factors 4.0 and 4.3). The highest gain for Optimal with respect to HalfGreedy is obtained for $s_{SA} = 16$ (factors 4.6 and 3.7) while the highest gain with respect to Greedy is obtained for $s_{SA} = 32$ (factors 8.6 and 4.1). In both cases, the lowest gain is with $s_{SA} = 8$ where their performances are very close.

[8] The average number of steps required to find a sample with Sequential and Parallel was always within a factor of $1 + 2 \cdot 10^{-6}$ in our experiments.

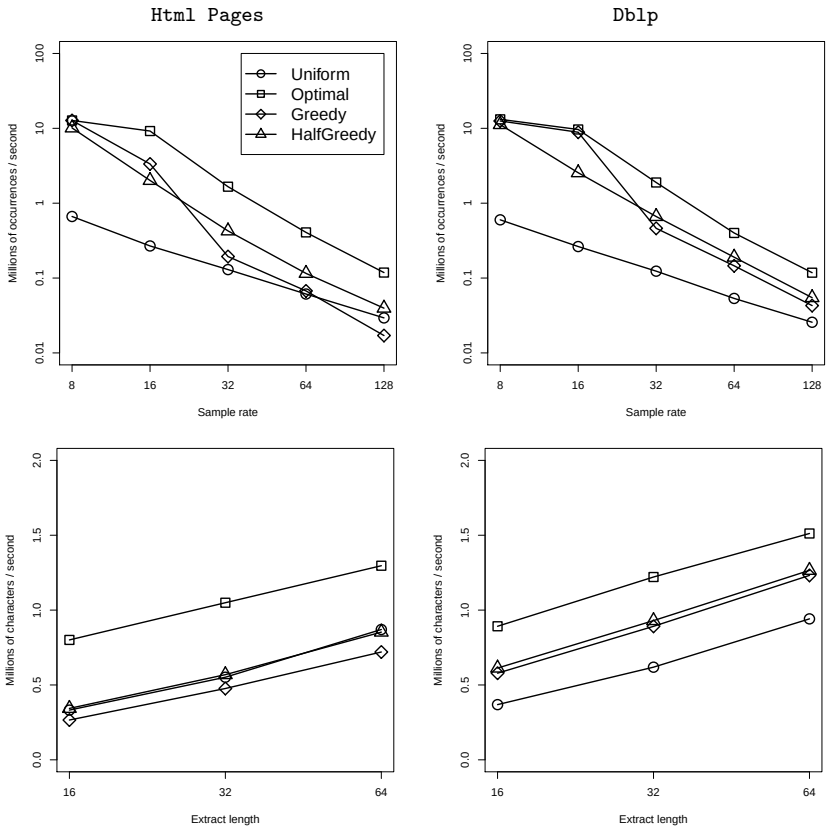

Fig. 4 Experimental results for `Html Pages` (left) and `Dblp` (right). Locate performance (top) and Extract performance with $s_{SA} = 128$ (bottom).

It is worth noticing that Optimal with $s_{SA} = 16$ is able to locate 9 to 10 million occurrences per second. This can be compared with 500 million occurrences per second located with Suffix Array. However, Suffix Array occupies about 4 times more space than Optimal. In Extract, Optimal is roughly twice as fast as the other strategies. The gain is limited due to the fact that, in any case, Extract requires c steps to extract c symbols, after finding the substring to be extracted. Thus, the penalty from a worse sampling strategy tends to be amortized by the number of extracted symbols (as the value of c becomes large).

A comment is in order at this point because the results presented in this paper are significantly better than the ones reported in its conference version [8]. This is mainly due to the new solution we adopted for marking the sampled rows. At each step of this algorithm, it is required to check whether the current row index is sampled or not. Whenever the average number of LF/Ψ steps is close to 0, the cost of this check becomes dominant. This check is usually performed by resorting to rank/select queries over a bit vector. We replaced the old implementation with a new one which is more stable in its performance, even if it may increase the size. The bit vector

is stored as is, requiring n bits of space. This allows us to determine, whether the current row is sampled, with a single lookup. For the rank query used to determine which sample we have found, we divide the bit vector into 256-bit blocks, and store the number of 1-bits before each block in $\log n$ bits. Solving rank then requires the retrieval of the stored value for the correct block, and counting the number of 1-bits in the block up to the queried position via the 64-bit popcount function provided in GCC. The function compiles either into a single instruction or a small subroutine, depending on architecture.

5 Conclusions

In this paper we addressed the problem of designing distribution-aware compressed full-text indexes when the distribution of subsequent queries is known beforehand. The advantage at query time is between 4–36 times better than the classical approach to Locate. In the case of Extract the advantage is reduced to 2. We showed that an optimal selection of positions can be computed efficiently in time $(O(n \log U))$ and space $(O(n \log n)$ bits). In practice, the time and space requirements are similar to or better than in suffix array construction, which is the usual bottleneck in constructing compressed full-text indexes. An interesting open problem asks for designing distribution-aware compressed indexes that are able to adapt themselves to an unknown distribution of queries.

References

1. A. Aggarwal, B. Schieber, and T. Tokuyama. Finding a minimum-weight k-link path graphs with the concave monge property and applications. *Discrete & Computational Geometry*, 12:263–280, 1994.
2. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proceedings of the 21st International Symposium on Algorithms and Computation, Part II (ISAAC 2010)*, volume 6507 of *LNCS*, pages 315–326. Springer, 2010.
3. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA 2011)*, volume 6942 of *LNCS*, pages 748–759. Springer, 2011.
4. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
5. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics*, 13, 2008.
6. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
7. P. Ferragina and G. Manzini. On compressing the textual web. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM)*, pages 391–400, 2010.
8. P. Ferragina, J. Sirén, and R. Venturini. Distribution-aware compressed full-text indexes. In *Proc 19th Annual European Symposium on Algorithms (ESA)*, pages 760–771, 2011.
9. R. Giancarlo. Dynamic programming: special cases. In Alberto Apostolico and Zvi Galil, editors, *Pattern Matching Algorithms*, pages 201–236. Oxford Univ. Press, 2nd edition, 1997.
10. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
11. R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32nd ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.
12. T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 17th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 317–326, 2001.

13. D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.
14. J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proceedings of the 18th Symposium on String Processing and Information Retrieval (SPIRE 2011)*, volume 7024 of *LNCS*, pages 174–184. Springer, 2011.
15. V. Mäkinen, G. Navarro, Sirén J., and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
16. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.
17. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
18. B. Schieber. Computing a minimum weight k-link path in graphs with the concave monge property. *Journal of Algorithms*, 29(2):204–222, 1998.
19. F. Silvestri. Mining query logs: Turning search usage data into knowledge. *Foundations and Trends in Information Retrieval*, 4(1-2):1–174, 2010.
20. J. Sirén. *Compressed Full-Text Indexes for Highly Repetitive Collections*. PhD thesis, University of Helsinki, 2012.
21. R. E. Wilber. The concave least-weight subsequence problem revisited. *Journal of Algorithms*, 9(3):418–425, 1988.