Practical Compressed Document Retrieval [⋆]

Gonzalo Navarro[1], Simon J. Puglisi[2], and Daniel Valenzuela[1]

[1] Dept. of Computer Science, University of Chile,
{gnavarro,dvalenzu}@dcc.uchile.cl
[2] School of Computer Science and Information Technology
Royal Melbourne Institute of Technology, simon.puglisi@rmit.edu.au

Abstract. Recent research on document retrieval for general texts has established the virtues of explicitly representing the so-called *document array*, which stores the document each pointer of the suffix array belongs to. While it makes document retrieval faster, this array occupies a significative amount of redundant space and is not easily compressible. In this paper we present the first practical proposal to compress the document array. We show that the resulting structure is significatively smaller than the uncompressed counterpart, and than alternatives to the document array proposed in the literature. We also compare various known algorithms for document listing and top-k retrieval, and find that the most useful combinations of algorithms run over our new compressed document arrays.

1 Introduction

Document retrieval queries aim at finding the documents of a text collection most relevant to a given query, where relevance is usually defined on frequency grounds. Such queries have been classically privative of Natural Language collections and handled with inverted indexes. In the last decade, however, there have been various research efforts towards generalizing them to arbitrary text collections, where the texts can correspond to ADN or protein sequences, text in Oriental languages (some of which cannot be easily split into words), program code, and symbolic sequences in general. See Gagie et al. [7] for a recent survey.

Muthukrishnan [16] established important milestones in this area. He proposed, among other less popular ones, the following fundamental document retrieval queries, which form the basis of more sophisticated retrieval activities:

- *Document listing:* List the $ndoc$ distinct documents where a pattern p appears as a substring.
- *Frequency computation:* Same as above but also compute the number of times p appears within each returned document.
- *Top-k retrieval:* Find the k documents where p appears most often.

[⋆] Partially funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile; by Fondecyt Grant 1-110066, Chile; and by the Australian Research Council.

Assume the text collection is formed by n documents, which are strings over alphabet $[1, \sigma]$, and let us call $T[1, N]$ their concatenation. Classical text indexes [1, 14] can find *all* the *occ* occurrences of pattern p in T in time $O(|p| + occ)$ and then filter the *ndoc* distinct documents. This solves document listing, but *occ* can be much larger than *ndoc*. Muthukrishnan [16] showed how to solve the document listing query in $O(|p| + ndoc)$ time, which is essentially optimal.

A serious concern on this solution is space, however. It requires $O(N \lg N)$ bits, much more than the $N \lg \sigma$ bits required by the text itself. Part of this space is used by a suffix tree [1], which can be easily replaced by one of the many compressed suffix arrays (CSAs) of the literature [17, 4]. Such CSAs represent a suffix array [14] plus the text, all within compressed space $|\mathsf{CSA}| \leq N \lg \sigma$.

The other part of the space owes to a so-called *document array*. Much research has been carried out around the problem of representing it in compact form [22, 20, 11, 8, 3]. The current situation is that one either has to spend $N \lg n$ bits for representing the document array using a *wavelet tree* [10], or one can simulate it using just $o(N)$ bits on top of $|\mathsf{CSA}|$. The second choice is clearly preferable both in theory and in practice for document listing.

However, document listing is just the most elementary activity. In order to rank documents by importance to the query, frequency computation and top-k retrieval are essential, and in this case the situation is very different. The wavelet tree representation directly computes frequencies and supports heuristics for top-k queries. The alternative simulation requires now the space for the global CSA *and* the CSA of each individual document. This turns out to be slower and require much more space in practice than the wavelet-tree based solution [3].

Therefore, the current status is that, if we wish not only to carry out document listing, but also to report frequencies, or to do top-k retrieval, the best alternative in practice is to store a wavelet tree representing the document array, which requires $N \lg n$ bits. This is possibly much more than the (at most) $N \lg \sigma$ bits required by a CSA (which by itself represents T and offers classical indexed text searches). The known techniques to compress wavelet trees [10] do not work for the document array.

In this paper we introduce the first compressed representation of wavelet trees that is useful for the document array. The representation uses grammar compression (precisely, RePair [12]) to exploit repetitions in the array. Such repetitions arise as a consequence of the compressibility of the text collection [9, 7]. Our experiments over various collections show that our technique obtains significant space reductions, up to 40% of the original wavelet tree sizes. In exchange, operation times are higher. Yet we confirm that the time (and space) is still better than the alternative of not using wavelet trees for the problem of document listing with frequencies. We also study the wavelet trees in combination with various techniques for top-k retrieval [11, 3], where our compressed wavelet trees offer a very attractive space/time tradeoff.

Our representation might have independent interest, as it is the first in grammar-compressing a sequence while supporting symbol *rank* and *select* operations on it. Those operations are useful in a wealth of applications.

2 Related Work

Muthukrishnan's solution [16] made use of the so-called *document array*. A *suffix array* $A[1,N]$ points to all the suffixes of $T[1,N]$ in lexicographic order [14]. All the occurrences of any p in T are pointed from an interval $A[sp,ep]$, which can be found in time $O(|p|\lg N)$ (or $O(|p|)$ with the help of the suffix tree [1]). The document array $D[1,N]$ is such that $D[i]$ tells the document suffix $A[i]$ belongs in T. So the document listing problem is solved by first finding $[sp,ep]$ using A, and then listing the distinct values in $D[sp,ep]$. To do this in $O(ndoc)$ time, Muthukrishnan uses a second array $C[1,N]$, which at $C[i]$ stores the last $j<i$ such that $D[j]=D[i]$. C must also answer range minimum queries (RMQs), telling in constant time the position of the minimum in a range of C cells.

In order to reduce space, the suffix tree or array can be replaced by a compressed suffix array (CSA) [17,4], which stores both A and T in compressed space, for example within $|\mathsf{CSA}|=(1+\frac{1}{\epsilon})H_0(T)+o(n\lg\sigma)$ bits [19] or $|\mathsf{CSA}|=nH_k(T)+o(n\lg\sigma)$ bits [5,10], where $H_k(T)\le\lg\sigma$ is the empirical k-th order entropy of T [17]. CSAs find $[sp,ep]$ in time as good as $search(p)=O(|p|\lg N)$ [19] or even $search(p)=O(|p|\lg\sigma)$ [5,10]. They compute any $A[i]$ or $A^{-1}[i]$ value in time t_A, for example $t_A=O(\log^{\epsilon}N)$ [19] or $t_A=O(\log^{1+\epsilon}N)$ [5,10]. They can also reproduce any text substring.

The document array D, however, requires $N\lg n$ bits, which is significant and totally redundant: one can infer $D[i]$ from $A[i]$ and some information on the limits of the documents in T. Array C is even more space-consuming, $N\lg N$ bits, and equally redundant. The RMQ data structure [6] adds $2N+o(N)$ bits. This extra space limits the applicability of the solution to document retrieval.

There have been various approaches to reduce the space of Muthukrishnan's solution. Mäkinen and Välimäki [22] used a *wavelet tree* [10] to represent D. While the wavelet tree takes essentially the same space of the plain representation, $N\lg n+o(N\lg n)$ bits, it can emulate array C, which is thus not represented. The time for document listing becomes $O(search(p)+ndoc\lg n)$. The wavelet tree also allowed them to compute the frequency of p within any document, in time $O(\lg n)$. The RMQ data structure was still necessary.

Gagie et al. [8] showed that the wavelet tree was powerful enough to get rid of the whole Muthukrishnan's algorithm. The wavelet tree alone, through a so-called *range quantile* operation, was able to deliver the distinct elements in $D[sp,ep]$, with their frequencies, in $O(\lg n)$ time per delivered item.

Culpepper et al. [3] explored different heuristics to solve the top-k problem using this very same wavelet tree. They found out that their so-called “greedy” heuristic was able to find the top-k documents much faster than listing them all and choosing the most frequent ones. They also showed that the structure was competitive with inverted indexes on Natural Language text collections.

A parallel development started with Sadakane [20]. He proposed to store a bitmap $B[1,N]$ marking with a 1 the positions in T where the documents started. B was enhanced with *rank* operations: $rank(B,i)$ tells the number of 1s in $B[1,i]$. Hence $D[i]=rank(B,A[i])$ could be computed with very little extra space on top of the CSA: A compressed representation of B requires just

$n \lg \frac{N}{n} + O(n) + o(N)$ bits and supports *rank* in constant time [18]. To emulate Muthukrishnan's algorithm, Sadakane showed that access to C is not really needed, just RMQ queries on C. He designed an RMQ structure using $4N + o(N)$ bits that does not need to access C. The time to list each document is $O(t_A)$. Both in theory and in practice, this solution is competitive in time and uses much less space than those based on wavelet trees, yet it only solves document listing. Hon et al. [11] showed how to reduce the extra space to just $o(N)$ by sparsifying the RMQ structure, yet the time raises to $O(t_A \lg^{1+\epsilon} N)$.

For computing frequencies, Sadakane [20] proposed to store a CSA for each document d of the collection. By computing A and A^{-1} a constant number of times over the global CSA and that of document d, it is possible to compute frequencies on document d. An extra, symmetric, RMQ data structure must be stored for this sake. Thus the space is $2|\mathsf{CSA}| + O(N)$ bits, which may compare favorably to the $|\mathsf{CSA}| + N \lg n + o(N \lg n)$ bits needed by wavelet trees. The time for computing a frequency is $O(t_A)$, which again may compare favorably with wavelet tree's $O(\lg n)$. In practice, however, Culpepper et al. [3] found that many small CSAs posed a much higher space overhead than that of the global CSA, so the structure was much larger than the wavelet tree. The speed was also slower than that offered by wavelet trees. We confirm in this paper that the solution is still slower than our slower-and-smaller compressed wavelet trees.

Hon et al. [11] showed that the second RMQ data structure introduced by Sadakane is unnecessary if one accepts an $O(\lg N)$ slowdown factor. In the light of the results of Culpepper et al. [3], this is unlikely to change matters in practice, because a reduction in $2N$ bits is insignificant but the slowdown is not. The key contribution of Hon et al., however, is an algorithm for top-k retrieval with time guarantees (which the heuristics of Culpepper et al. do not offer). Hon et al. build a sparse suffix tree on the collection, so that top-k queries over an interval of multiples of $g = k \cdot b$, for a parameter b, are precomputed. Thus to solve for any interval $[sp, ep]$, only $O(kb)$ elements at the extremes must be accessed, their frequency counted, and possibly inserted into the precomputed result. The extra space is $O((N/b) \lg N \lg n)$ bits on top of a solution for computing frequencies. By choosing $b = \Theta(\lg^{2+\epsilon} N)$, this space is $o(n)$ and the time for top-k queries is $O(k\, t_A \lg^{3+\epsilon} N)$. Any of the discussed solutions for computing frequencies can be used, and thus wavelet trees are of interest as a building block of this solution. There is no comparison in the literature, however, between this technique and the heuristics of Culpepper et al. [3], which as explained work on wavelet trees.

Thus, the best performance in practice is given by the wavelet tree of D, but its space is still high. The only clue at compressing it was given by Gagie et al. [7], who noted that D contains almost the same repetitions of the *differential* suffix array [9], and thus a grammar-based compression would reduce its size when the text is compressible. The practical impact of this theoretical result had not been verified, however. Moreover, the situation is more complicated because we do not need to represent D, but the wavelet tree of D, in order to support the various document retrieval tasks. The main point of this paper is to implement a grammar-compressed wavelet tree for D and evaluate its practical performance.

3 Bitmaps and Wavelet Trees

Given a bitmap $B[1, N]$, we define, for $b \in \{0, 1\}$, operation $rank_b(B, i)$ as the number of occurrences of bit b in $B[1, i]$, and $select_b(B, j)$ as the position in B of the jth occurrence of bit b.

Both operations can be solved in constant time by spending $o(N)$ bits on top of B [15], or by representing B in compressed form [18]: Let m be the number of 1s in B, then the total space is $m \lg \frac{N}{m} + O(m) + o(N)$.

A *wavelet tree* [10] represents a sequence $S[1, N]$ over an alphabet $[1, \rho]$. At the root it stores a bitmap $B[1, N]$ so that $B[i] = 0$ iff $S[i] \leq \rho/2$. The left child of the root represents the subsequence of S formed by the symbols $\leq \rho/2$; the other symbols form a subsequence at the right child. Those children are processed recursively over their alphabet ranges, until reaching the leaves. The wavelet tree has $O(\rho)$ nodes and height $\lceil \lg \rho \rceil$. Its bitmaps add up $N \lceil \lg \rho \rceil$ bits.

By using *rank* and *select* on the bitmaps, the wavelet tree gives access to any $S[i]$ in time $O(\lg \rho)$, thus it constitutes an alternative representation of S within about the same size, $N \lg \rho + o(N \lg \rho)$ bits. Other wavelet tree traversals compute also symbol *rank* and *select* on S, where the argument b can be any $c \in [1, \rho]$, also in time $O(\lg \rho)$. As explained, the wavelet tree is also useful for other types of queries of interest (in particular) to document retrieval [8, 3].

If the compressed bitmap representation is used [18], then the space of the wavelet tree becomes the zero-order entropy of S, $NH_0(S) + o(N \log \sigma)$ [10]. In our case, however, the zero-order entropy of the document array is likely to be $\lg n$ bits per symbol, unless the document sizes are very different. There is no relation to the compressibility of the text itself.

4 Grammar Compression of Bitmaps

We describe a grammar-based compression of bitmaps $B[1, N]$ that supports *rank* and *select* operations on the compressed representation. We focus on Re-Pair [12] compressor. It successively finds the most frequent pair of symbols in the text, yz, and replaces it by a new symbol x (which can be involved in further pairings), adding a new grammar rule $x \rightarrow yz$. When all the pairs are unique, RePair terminates and delivers the remaining sequence, $\mathcal{C}$, and the set of rules, $\mathcal{R}$. We use a variant that generates a balanced grammar [21], of height $O(\lg N)$.

For providing random access we use sampling. Let $\ell(c) = 1$ for terminals c, and for nonterminals let $\ell(x)$ be the length of the string of terminals x expands to (that is, $\ell(x) = \ell(y) + \ell(z)$ if $x \rightarrow yz \in \mathcal{R}$). Now let $L(i) = 1 + \sum_{j=1}^{i-1} \ell(\mathcal{C}[j])$ the starting position in B of the symbol $\mathcal{C}[i]$ when expanded.

We sample B at regular intervals s. For each position $B[i \cdot s]$ we store $P[i] = (p, o, r)$, where p is the position in $\mathcal{C}$ of the symbol whose expansion will contain $B[i \cdot s]$, that is, $p = \max\{j,\ L(j) \leq i \cdot s\}$. The second component is the offset within that symbol, $o = i \cdot s - L(p)$, and the third is the rank up to that symbol, $r = rank_1(B, L(p) - 1)$. Finally, we store, for the nonterminals x, the length $\ell(x)$ and the number of 1s, $r(x)$, of the string of terminals they expand to.

To answer $rank_1(B, i)$, we compute $j = \lfloor i/s \rfloor$ and $P[j] = (p, o, r)$. We then start from $\mathcal{C}[p]$ with position $l = L(p) = i - o$ and rank r. From position p we advance in $\mathcal{C}$ until $l > i$. Each symbol of $\mathcal{C}$ can be processed in constant time while l and r are updated, since we know $\ell(x)$ and $r(x)$ for any symbol x. Finally we arrive at a position $p' \geq p$ so that $l = L(p') \leq i < L(p' + 1) = l + \ell(\mathcal{C}[p'])$. At this point we complete our computation by recursively expanding $\mathcal{C}[p'] = x$. Let $x \rightarrow yz \in \mathcal{R}$, then if $l + \ell(y) \leq i$ we expand y; otherwise we increase l by $\ell(y)$, r by $r(y)$, and expand z. As the grammar is balanced the total time is $O(s + \lg N)$.

For *select* we obtain the same complexity by first binary searching P to find the right interval and then traversing sequentially the block, until exceeding the desired number of 0s or 1s, and finally expanding the last symbol of $\mathcal{C}$.

Let $R = |\mathcal{R}|$ be the number of rules in the grammar and $C = |\mathcal{C}|$ the length of the final array. Then a RePair compressor would require $O((R + C) \lg R)$ bits. Our representation requires $O(R \lg N + C \lg R + (N/s) \lg N)$, and the time for the operations is $O(s + \lg N)$. The minimum interesting value for s is $\lg N$, where we achieve space $O((R + C) \lg N + N)$ bits and $O(\lg N)$ time for the operations. We can reduce the $O(N)$ extra space to $o(N)$ by increasing s, which impacts query times and makes them superlogarithmic.

The scheme can be extended to sequences $S[1, N]$ over a small alphabet $[1, \rho]$. The only difference is that the nonterminals x must store $r_c(x)$ for each $c \in [1, \rho]$. Similarly we must store ρ rank values at each sampled position. This raises the overall space to $O(R\rho \lg N + C \lg R + (N\rho/s) \lg N)$. The time stays the same.

In practice. There are several ways to represent $\mathcal{R}$ in compressed form. We choose one [9] that allows for random access to the rules. It represents $\mathcal{R}$ in the form of a directed acyclic graph (DAG) as a sequence S_R and a bitmap S_B. A node is identified as a position in S_B, where a 1 denotes and internal node and a 0 a leaf. The two children of $S_B[i] = 1$ are written next to i, thus we obtain all the subtree by traversing $S_B[i \ldots]$ until we have seen more 0s than 1s. The 0s in S_B are associated to positions in S_R (that is, $S_B[i] = 0$ is associated to $S_R[rank_0(S_B, i)]$). Those leaf symbols are either terminals or nonterminals. Nonterminals are actually positions in S_B that must be recursively expanded. This DAG representation takes, in good cases, as little as 50% of the space required by a plain array representation of $\mathcal{R}$ [9].

In order to reduce the $O(R \lg n)$ space required to store $\ell(x)$ and $r(x)$ for nonterminals x, we store the data only for some of them and obtain the others via expanding the nonterminals. Given a parameter δ, we guarantee that no nonterminal in $\mathcal{C}$ requires expanding at depth more than δ to determine its length and number of 1s. That is, we expand each $\mathcal{C}[i]$ until depth δ or until reaching an already sampled nonterminal. Those nonterminals at depth δ are then sampled. We set up a bitmap B_δ where each sampled nonterminal has a 1, and store $\ell(x)$ and $r(x)$ of marked nonterminal x at an array $E[rank_1(B_\delta, x)]$.

We use a RePair implementation by ourselves (available at `www.dcc.uchile.cl/gnavarro/software`). It has a variant that, although does not guarantee balancedness, has always produced a grammar of very small height in our experiments. The variant that ensures balancedness harms compression in practice.

5 Grammar Compression of Wavelet Trees

Given now a sequence $S[1, N]$ over alphabet $[1, n]$, we build the wavelet tree of S and represent its bitmaps using the compressed format of Section 4. Consider a RePair representation $(\mathcal{R}, \mathcal{C})$ of S, where the sizes of the components is R and C as before. Now take the top-level bitmap B of the wavelet tree. Bitmap B can be regarded as the result of mapping the alphabet of S onto two symbols, 0 and 1. Thus, a grammar $(\mathcal{R}', \mathcal{C}')$ where the terminals are mapped accordingly, generates B. Since the number of rules in $\mathcal{R}'$ is still R and that of $\mathcal{C}'$ is C, the representation of B requires $O(R \lg N + C \lg R + (N/s) \lg N)$ bits (this is of course pessimistic; many more repetitions could arise due to the mapping).

The bitmaps stored at the left and right children of the root correspond to a partition of S into two subsequences S_1 and S_2. Given the grammar that represents S, we can obtain the one that represents S_1 and S_2 by removing all the terminals in the right sides that do not belong to the proper subalphabet, and removing rules with right hands of length 0 or 1. Thus at worst the left and right side bitmaps can also be represented within $O(R \lg N + C \lg R)$ bits each, plus $O((N/s) \lg N)$ for the whole level. Added over the n wavelet tree nodes, the overall space is no more than n times that of the RePair compression of S. The time for the operations raises to $O((s + \lg N) \lg n)$.

Although this does not look alphabet-friendy, and actually the upper bounds are no better than applying the method of Section 4 on a large alphabet ($\rho = n$), the analysis is a (very) pessimistic upper bound. Still one can expect that the repetitions exploited by RePair get cut by half as we descend one level of the wavelet tree, so that after descending some levels no repetition structure can be identified and RePair compression becomes ineffective.

In practice. As n is large, we use a wavelet tree design that concatenates all the bitmaps of the same wavelet tree level [2]. We use one set of rules $\mathcal{R}$ per level.

As the repetitions that could be present in S get shorter when we move deeper in the wavelet tree, we evaluate at each level whether our RePair-based compression is actually better than an entropy-compressed representation [18] or even a plain one, and choose the one with smallest space. Moreover, as *rank* and *select* operations are significantly slower on our RePair-compressed representation, we use a parameter $0 < \alpha \leq 1$ so that we prefer RePair compression only when its size is α times that of the alternatives, or less.

6 Experimental Results

In this section we compare various practical alternatives to document listing and top-k document retrieval. We have chosen four collections of different nature, such as English, Chinese, biological, and symbolic sequences. We show the bpc of its global CSA divided by $\lg \sigma$ to give an idea of its compressibility ratio.

ClueChin: A 2.3 MB sample of ClueWeb09 (`http://boston.lti.cs.cmu.edu/Data/clueweb09`), formed by 23 Web pages in Chinese. Ratio: 5.34/7.99=0.68.

ClueWiki: A 141 MB sample of ClueWeb09, formed by 3,334 Web pages from the English Wikipedia. Ratio: 4.74/6.98=0.68.

KGS: A 75 MB collection of 18,838 sgf-formatted Go game records from year 2009 (`http://www.u-go.net/gamerecords`). Ratio: 4.48/6.93=0.65.

Proteins: A 60 MB collection formed by 143,244 sequences of Human and Mouse proteins (`http://www.ebi.ac.uk/swissprot`). Ratio: 6.02/6.57=0.92.

Our tests were run on a Intel Core2 Duo machine, 3Ghz, with 8GB RAM and 6MB cache. Our code was compiled using g++ with full optimization.

As the CSA search for the interval $[sp, ep]$ corresponding to a pattern p is common to all the approaches, we do not consider the time for this search (which never exceeds 0.02 milliseconds per query) nor the space for that global CSA (shown for each collection in the previous itemization), but only the extra space/time to support document retrieval once $[sp, ep]$ has been determined. We give the space usage in bits per text character (bpc), and measure user times.

Sadakane's representation [20] builds a separate CSA for each document. For this sake we use a very competitive variant [13, 2] available at *PizzaChili* (`http://pizzachili.dcc.uchile.cl/indexes/SSA`). It uses a plain and fast representation for bitmap B (from `http://libcds.recoded.cl`), and an efficient implementation for the two RMQs (from `http://www.uni-ulm.de/in/theo/research/sdsl`). For the space we charge only $2N$ bits for each RMQ structure and zero for B, to account for possible future space reductions.

Our grammar compressed wavelet trees offer a space/time tradeoff depending on the α value, which can be the same for all levels, or decreasing for the deeper levels (where one visits more nodes and thus being slower has a higher impact). Another space/time tradeoff is obtained with the sampling parameter s. We only show one alternative with $\alpha = 1$ and one best-performing alternative with $\alpha < 1$.

As explained, alternative solutions [20, 11] for the basic document listing problem are hardly improvable. They require very little extra space and are likely to perform similarly to wavelet trees in time. Our experiments focus on document listing with frequencies, and in top-k retrieval.

6.1 Document Listing with Frequencies

Previous work [3] has demonstrated that the quantile approach [8] is clearly preferable, in space and time, over previous ones based on wavelet trees [22]. Therefore we carry out the quantile algorithm over a plain wavelet tree representation (*WT-Plain*), over one where the bitmaps are statistically compressed [18] (*WT-RRR*), and over our RePair-compressed ones. As explained, we show a variant with $\alpha = 1$ (*WT-RP*, which at each level chooses the lowest space between RePair, plain, or statistically compressed bitmap), and the best performing policy we tried for choosing $\alpha < 1$ values (*WT-RP alpha*).

We also compare Sadakane's approach [20] (*SADA*) on collection *ClueChin*. The construction times over the other collections, with many more documents, was extremely high. This tiny collection will be sufficient to expose the practicality problems of this approach.

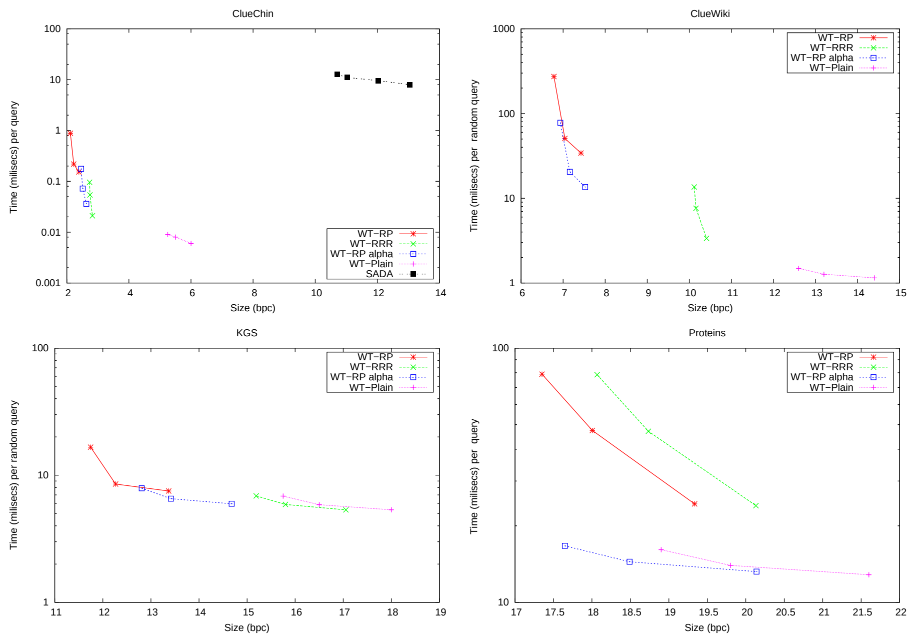

Fig. 1. Experiments for document listing with term frequencies.

We chose 10,000 random intervals of the form $[sp, ep]$, considering values for $ep - sp$ from $1,000$ to $10,000$, and listed the distinct documents in the interval with their frequencies. The relative positions of the curves were the same for every $ep-sp$ value, so for lack of space we show just the plots for $ep-sp = 10,000$.

Fig. 1 shows the results. Even on *ClueChin*, with just 23 documents, the space overhead of indexing them separately makes Sadakane's approach impractical (even with the generous assumptions on extra bitmaps and RMQs). It is also slower. For this reason we do not compare Hon et al.'s variant [11], that achieves $|\mathsf{CSA}| + o(n)$ extra space, because it will be much slower and the reduction on space (by $4N$ bits), would be insufficient to make it competitive.

The results are different depending on the type of collections, but in general our compressed representation is able to reduce the space of the plain wavelet tree by a wide margin. The compressed size is 40% to 75% of the original size. The exception is *Proteins*, where the text is mostly incompressible and this translates into the incompressibility of the document array.

While the *WT-RP* is significantly slower than *WT-Plain* (up to 20 times slower in the most extreme case), the *WT-Alpha* versions provide useful tradeoffs. They achieve compression ratios of 50% to 80% and significantly reduce time gaps, to 7 times slower in the most extreme case. The answer time over the interval $[sp, ep]$ of length 10,000 is around 10-20 milliseconds (msecs). We note that our slowest version is still 10 times faster than *SADA*. It is also much faster than listing all the documents individually (e.g., 500 times faster on *ClueChin*).

6.2 Top-k Retrieval

Culpepper et al. [3] present and test two heuristics for top-k retrieval, which run on wavelet trees. The one called *greedy* is always superior, so we test that one in this paper, over our different wavelet tree representations.

We also compare Hon et al.'s structure [11]. Short of implementing it, we (quite) optimistically simulate its performance by charging zero time and zero space to some parts of the data structure and search process. We combine it only with the most promising wavelet tree in each plot.

Recall that the method divides the suffix array A into blocks of fixed length g. After finding the suffix array interval $A[sp, ep]$ corresponding to a pattern, the part of the search corresponding to the blocks fully contained in $[sp, ep]$ is solved with a sampled suffix tree (which we have not implemented and will assume costs zero space and time). The other two subintervals from sp to the start of the next block, and from the end of the last block to ep, are solved by brute force, that is, extracting all the distinct documents and computing their frequency in the whole interval $A[sp, ep]$. This part will be actually executed in different ways. Finally, the candidates obtained by brute force and those given by the suffix tree are ranked and merged (which we will not do and will assume costs zero).

Various alternatives to extract the values of D and compute their frequency are considered. *WT-RP-HON* and *WT-RP-alpha HON* use the corresponding wavelet tree variants for this task. In case the interval $[sp, ep]$ contains no blocks, they switch to Culpepper et al.'s method. On *ClueChin* we tried other variants related to Sadakane's solution [20]. *SADA-HON* uses Sadakane's structure just as in the document listing experiment. *HON* does not use the two RMQ structures, but instead maps the start of the interval $[sp, ep]$ to the local CSA using A and A^{-1}, and then binary searches the end of the local interval by mapping each probe back to the global CSA [11]. Finally, *Search-HON* simply searches for p in the local CSA in order to determine its frequency.

Fig. 2 shows the results. We selected 1,000 substrings at random positions, of length 3 and 6, and retrieved the top-k documents for each, for $k = 1$ and 10. Longer patterns produce shorter $[sp, ep]$ intervals. The relative space and time performance comparisons are similar to those of document listing with frequencies. Most times are around a few tens of msecs per query.

With respect to Hon et al.'s method, *Search-HON* and *HON* are very similar in time, much slower and $4N$ bits smaller than *SADA-HON*. Yet, none of those variants of the original formulation [11] is competitive in practice. What is much more interesting is their combination with a wavelet tree. While it is slightly inferior to Culpepper et al.'s *greedy* heuristic on collections *ClueChin* and *ClueWiki*, on the other two Hon et al.'s method speeds up the heuristic by a factor of up to 1.5–6.5 for $k = 1$ and 2.2–2.5 for $k = 10$. While this is a space- and time-optimistic simulation of Hon et al.'s method, it should be quite tight.

Final remarks. We have shown that the wavelet tree is the best data structure to compute frequencies and support top-k algorithms, and reduced its size by up to 40% while answering within tens of msecs. Also, theoretical top-k proposals

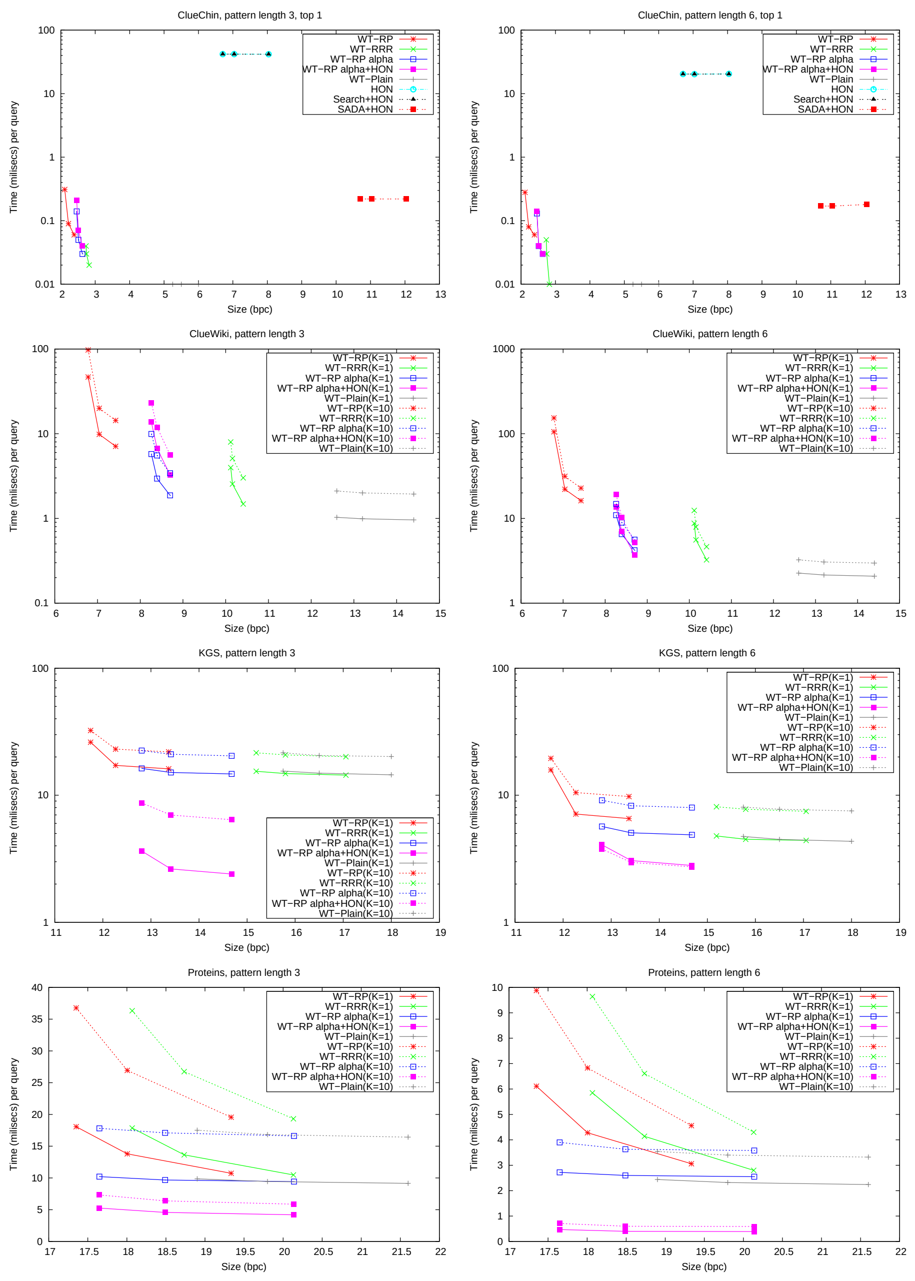

Fig. 2. Experiments for top-k queries.

[11] are shown to be worth implementing. Yet, even our smaller indexes (except on the tiny *ClueChin*) are even bigger than the CSAs of the collections (7-17 vs 4.5-6.0 bpc), thus there is much room for improvement in document retrieval. We are still far from the asymptotic space optimality achieved for pattern matching.

References

1. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
2. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *SPIRE*, pages 176–187, 2008.
3. S. Culpepper, G. Navarro, S. Puglisi, and A. Turpin. Top-k ranked document search in general text databases. In *ESA*, pages 194–205 (part II), 2010.
4. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM JEA*, 13:article 12, 2009.
5. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.
6. J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *ESCAPE*, pages 459–470, 2007.
7. T. Gagie, G. Navarro, and S. Puglisi. Colored range queries and document retrieval. In *SPIRE*, pages 67–81, 2010.
8. T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *SPIRE*, pages 1–6, 2009.
9. R. González and G. Navarro. Compressed text indexes with fast locate. In *CPM*, pages 216–227, 2007.
10. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.
11. W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *FOCS*, pages 713–722, 2009.
12. N.J. Larsson and J. A. Moffat. Offline Dictionary-Based Compression. *Proc. of the IEEE*, 88:1722–1732, 2000.
13. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *SPIRE*, pages 214–226, 2007.
14. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.
15. I. Munro. Tables. In *FSTTCS*, pages 37–42, 1996.
16. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.
17. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1):article 2, 2007.
18. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.
19. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Alg.*, 48(2):294–313, 2003.
20. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Discr. Alg.*, 5(1):12–22, 2007.
21. H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discr. Alg.*, 3:416–430, 2005.
22. N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *CPM*, pages 205–215, 2007.