Document Listing on Repetitive Collections

Travis Gagie[1], Kalle Karhu[2], Gonzalo Navarro[3],
Simon J. Puglisi[1], and Jouni Sirén[3]

[1] Helsinki Institute for Information Technology (Aalto),
Department of Computer Science, University of Helsinki
{travis.gagie, simon.j.puglisi}@gmail.com
[2] Department of Computer Science and Engineering, Aalto University,
kalle.karhu@aalto.fi
[3] Department of Computer Science, University of Chile,
{gnavarro, jsiren}@dcc.uchile.cl

Abstract. Many document collections consist largely of repeated material, and several indexes have been designed to take advantage of this. There has been only preliminary work, however, on document retrieval for repetitive collections. In this paper we show how one of those indexes, the run-length compressed suffix array (RLCSA), can be extended to support document listing. In our experiments, our additional structures on top of the RLCSA can reduce the query time for document listing by an order of magnitude while still using total space that is only a fraction of the raw collection size. As a byproduct, we develop a new document listing technique for general collections that is of independent interest.

1 Introduction

Document listing is a fundamental and well-studied problem in information retrieval. It is known how to store a collection of documents in entropy-compressed space such that, given a pattern, we can quickly list the distinct documents in which that pattern occurs [15, 8]. If the collection is repetitive, however — e.g., genomes of individuals of the same or related species, software repositories, or versioned document collections — then its statistical entropy may not capture its true compressibility (the statistical entropy does not decrease if we concatenate the same text several times). Several indexes for exact pattern matching [9, 4, 2] take good advantage of repetitiveness, but to date there has been no work on document retrieval in this setting.

In this paper we show how Mäkinen et al.'s [9] run-length compressed suffix array (RLCSA) can be extended to support fast document listing. We present two different solutions. In Section 3, we show that interleaving the longest common prefix (LCP) arrays of the individual documents, in the order given by the

[1] This work was supported in part by Academy of Finland grants 250345 (CoECGR) and 134287; Fondecyt grant 1-110066, Chile; the Helsinki Doctoral Programme in Computer Science; the Jenny and Antti Wihuri Foundation, Finland; and the Millennium Nucleus for Information and Coordination in Networks (ICM/FIC P10-024F), Chile.

global LCP of the collection, yields long runs of equal values on repetitive collections, which makes this so-called interleaved LCP (ILCP) array highly compressible. Further, we show that a classical document listing technique [11], designed for a completely different array, works almost verbatim over the ILCP, and this yields a new document listing technique of independent interest for generic document collections (not only repetitive). In Section 4 we explore the idea, dubbed PDL, of precomputing the answers of document listing queries for all suffix tree nodes with enough leaves, and exploiting repetitiveness by grammar-compressing the resulting sets of answers. In Section 5 we experimentally show that the ILCP takes very little extra space on top of the RLCSA, and can speed up the RLCSA when the pattern appears many times in the documents; PDL is an order of magnitude faster and still uses only a fraction of the original text size.

2 Related Work

The best current solutions for document listing are based on an idea by Muthukrishnan [11]. Let $T[1..n]$ be the concatenation of the collection of d documents separated by copies of a special character "\$". Muthukrishnan's solution stores the suffix tree [18] of T, which in particular includes the suffix array [10] $\mathsf{SA}[1..n]$. The solution also stores a so-called document array $D[1..n]$ of T, in which each cell $D[i]$ stores the identifier of the document containing $T[\mathsf{SA}[i]]$; an array $C[1..n]$, in which each cell $C[i]$ stores the largest value $h < i$ such that $D[h] = D[i]$, or 0 if there is no such value h; and a data structure supporting range-minimum queries (RMQs) over C, $\text{RMQ}_C(i, j) = \text{argmin}_{i \le k \le j} C[k]$. These data structures take a total of $\mathcal{O}(n \lg n)$ bits. Given a pattern $P[1..m]$, the suffix tree is used to find the interval $\mathsf{SA}[\ell..r]$ that contains the starting positions of the suffixes prefixed by P. It follows that every value $C[i] < \ell$ in $C[\ell..r]$ corresponds to a distinct document in $D[i]$. Thus a recursive algorithm finding all those positions i starts with $k = \text{RMQ}_C(\ell, r)$. If $C[k] \ge \ell$ it stops. Otherwise it reports document $D[k]$ and continues recursively with the ranges $C[\ell, k-1]$ and $C[k+1, r]$ (the condition $C[k] \ge \ell$ always uses the original ℓ value). In total, the algorithm uses $\mathcal{O}(m + \mathsf{ndoc})$ time, where ndoc is the number of documents returned.

Sadakane [15] gave a compressed version of Muthukrishnan's solution, which stores only a compressed suffix array CSA of T, a sparse bitvector $B[1..n]$ indicating where in T each document starts, an RMQ data structure for C that returns the position of the leftmost minimum in a range without accessing C, and a bitmap $V[1..d]$ to record which document identifiers we have already returned. Fischer [3] showed that such an RMQ data structure takes only $2n + o(n)$ bits and can answer queries in $\mathcal{O}(1)$ time. These data structures take a total of $|\mathsf{CSA}| + 2n + d \lg(n/d) + \mathcal{O}(d) + o(n)$ bits. Here $d \lg(n/d) + \mathcal{O}(d) + o(n)$ bits are for a sparse bitvector representation (e.g., [13]) of B, which has only d 1s. This representation answers in constant time query $\mathsf{rank}(B, i)$, which gives the number of 1s in $B[1..i]$. Now, given P, we use CSA to find ℓ and r, then emulate Muthukrishnan's algorithm: After each RMQ giving position k we use CSA and B to compute $D[k] = \mathsf{rank}(B, \mathsf{CSA}[k])$, then check the bitmap V to see whether

we have already returned that document. If $V[D[k]] = 1$, we stop that recursive branch, else we return $D[k]$, mark $V[D[k]] \leftarrow 1$, and continue recursing. In total we use $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc} \cdot \mathsf{lookup}(n))$ time, where $\mathsf{search}(m)$ is the time to find ℓ and r and $\mathsf{lookup}(n)$ is the time to access a cell of SA, using CSA.

Hon et al. [8] push the space down further by sampling array C. The array is divided into blocks of length b, and an array $C'[1..n/b]$ stores the minima of the blocks. The recursive RMQs algorithm is run over C', so that each position $C'[k]$ found requires exploring the documents in one block of D, $D[(k-1)b+1..kb]$. By setting, say, $b = \lg^{\epsilon} n$ for a constant $\epsilon > 0$, the space becomes $|\mathsf{CSA}| + d\lg(n/d) + \mathcal{O}(d) + o(n)$ bits and the time raises to $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc} \cdot \mathsf{lookup}(n) \lg^{\epsilon} n)$.

In a repetitive environment, one can use an RLCSA [9] as the CSA. However, those $2n + o(n)$ bits of Sadakane [15], and even the $o(n)$ bits of Hon et al. [8], are likely to dominate the space requirement.

Another trend to simulate Muthukrishnan's algorithm is to represent the document array $D[1..n]$ explicitly using a wavelet tree [7], which uses $n \lg d + o(n)$ bits and can access any $D[i]$, as well as compute $\mathsf{rank}_c(D, i)$ and $\mathsf{select}_c(D, j)$, in time $\mathcal{O}(\lg d)$. The first query counts the number of times c occurs in $D[1..i]$, whereas the second gives the position in D of the jth occurrence of c. The wavelet tree root divides values $\le d/2$ and $> d/2$ in $D[1..n]$, storing only a bitmap $B[1..n]$ where $B[i] = 0$ iff $D[i] \le d/2$. Then, recursively, the left child of the root represents the subsequence of D with values $\le d/2$, and the right child the subsequence with values $> d/2$. The leaves represent runs of a single value in $[1..d]$, and the tree has height $\lg d$.

Mäkinen and Välimäki [17] showed that the wavelet tree of D can also emulate array C, as $C[i] = \mathsf{select}_{D[i]}(\mathsf{rank}_{D[i]}(D, i) - 1)$. Then, Gagie et al. [6] showed that just the CSA and the wavelet tree of D provided document listing in time $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc} \lg(n/\mathsf{ndoc}))$, without using any RMQ structure. Navarro et al. [12] showed that this wavelet tree is grammar-compressible, as D contains repeated substrings at almost the same positions of the runs found in SA.

3 Interleaved LCP Array

The longest-common-prefix array $\mathsf{LCP}_S[1..|S|]$ of a string S is defined such that $\mathsf{LCP}_S[1] = 0$ and, for $2 \le i \le |S|$, $\mathsf{LCP}_S[i]$ is the length of the longest common prefix of the lexicographically $(i-1)$th and ith suffixes of S, that is, between $S[\mathsf{SA}_S[i-1]..|S|]$ and $S[\mathsf{SA}_S[i]..|S|]$, where SA_S is the suffix array of S. We define the interleaved LCP array of T, ILCP, to be the interleaving of the LCP arrays of the individual documents according to the document array.

Definition 1. *Let $T[1,n] = S_1 \cdot S_2 \cdots S_d$ be the concatenation of documents S_j, D the document array of T, and LCP_{S_j} the longest common prefix array of string S_j. Then the* interleaved LCP array of T *is defined, for all $1 \le i \le n$, as*

$$\mathsf{ILCP}[i] \quad = \quad \mathsf{LCP}_{S_{D[i]}}\left[\mathsf{rank}_{D[i]}(D, i)\right].$$

The following property of ILCP makes it suitable for document retrieval.

Lemma 1. *Let $T[1,n] = S_1 \cdot S_2 \cdots S_d$ be the concatenation of documents S_j, SA its suffix array and D its document array. Let $\mathsf{SA}[\ell..r]$ be the interval that contains the starting positions of suffixes prefixed by a pattern $P[1..m]$. Then the values strictly less than m in $\mathsf{ILCP}[\ell..r]$ are in the same positions as the leftmost occurrences in $D[\ell..r]$ of the distinct document identifiers in that range.*

Proof. Let $\mathsf{SA}_{S_j}[\ell_j..r_j]$ be the interval of all the suffixes of S_j starting with $P[1..m]$. Then it must hold that $\mathsf{LCP}_{S_j}[\ell_j] < m$, as otherwise $S_j[\mathsf{SA}[\ell_j - 1]..\mathsf{SA}[\ell_j - 1] + m - 1] = S_j[\mathsf{SA}[\ell_j]..\mathsf{SA}[\ell_j] + m - 1] = P$ as well, contradicting the definition of ℓ_j. For the same reason, it holds that $\mathsf{LCP}_{S_j}[\ell_j + k] \geq m$ for all $1 \leq k \leq r_j - \ell_j$. Now, let S_j start at position $p_j + 1$ in T, where $p_j = |S_1 \cdots S_{j-1}|$. Because each S_j is terminated by the special symbol "\$", the lexicographic ordering between the suffixes $S_j[k..]$ in SA_{S_j} is the same as of the corresponding suffixes $T[p_j + k..]$ in SA. That is, it holds that $\langle \mathsf{SA}[i], D[i] = j, 1 \leq i \leq n \rangle = \langle p_j + \mathsf{SA}_{S_j}[i], 1 \leq i \leq |S_j| \rangle$. Or, put another way, $\mathsf{SA}[i] = p_j + \mathsf{SA}_{S_j}[\mathsf{rank}_j(D,i)]$ whenever $D[i] = j$. Now, let f_j be the leftmost occurrence of j in $D[\ell..r]$. This means that $\mathsf{SA}[f_j]$ is the lexicographically first suffix of S_j that starts with P. By definition of ℓ_j, it holds that $\ell_j = \mathsf{rank}_j(D, f_j)$. Thus, by definition of ILCP, it holds that $\mathsf{ILCP}[f_j] = \mathsf{LCP}_{S_j}[\mathsf{rank}_j(D, f_j)] = \mathsf{LCP}_{S_j}[\ell_j] < m$, whereas all the other $\mathsf{ILCP}[k]$ values, for $\ell \leq k \leq r$, where $D[k] = j$, must be $\geq m$. □

Therefore, for the purposes of document listing, we can replace the C array by ILCP in Muthukrishnan's algorithm: instead of recursing until listing all the positions k such that $C[k] < \ell$, we recurse until listing all the positions k such that $\mathsf{ILCP}[k] < m$.

3.1 Document Listing in General Collections

Under Szpankowski's very general A2 probabilistic model [16] (which includes Bernoulli and Markov chains of fixed memory), the maximum LCP value in a string S is almost surely (a very strong kind of convergence[2], which we abbreviate a.s.) $\mathcal{O}(\lg |S|)$ [16]. This means that storing ILCP explicitly requires a.s. at most $n \lg\lg(n/d) + \mathcal{O}(n)$ bits, usually far less than the $n \lg d$ bits required by C.

The fact that we are interested in the values 0 to $m-1$ in ILCP gives a new relevant index for document listing in general collections. Grossi et al. [7] proved that, if we give the wavelet tree of a sequence S any shape (i.e., not necessarily balanced) and represent the wavelet tree bitmaps using a compressed representation (e.g., [13]), then the total space is the zero-order entropy of the represented sequence, $H_0(S)$, plus $o(nh)$ bits, where h is the wavelet tree's height. The $o(nh)$ bits can become $\mathcal{O}(nh/\lg n)$ if we use the bitmap representation of Pătraşcu [14] instead. Now consider a representation where the leftmost leaf is at depth 1, the next 2 leaves are at depth 3, the next 4 leaves are at depth 5, and in general the 2^{d-1}th to $(2^d - 1)$th leftmost leaves are at depth $2d - 1$. Then the ith

[2] A sequence X_n tends to a value β almost surely if, for every $\epsilon > 0$, the probability that $|X_N/\beta - 1| > \epsilon$ for some $N > n$ tends to zero as n tends to infinity, $\lim_{n\to\infty} \sup_{N>n} \Pr(|X_N/\beta - 1| > \epsilon) = 0$.

leftmost leaf is at depth $\mathcal{O}(\lg i)$. If we build this wavelet tree on sequence ILCP, the total space is $H_0(\mathsf{ILCP}) + \mathcal{O}(n \lg d / \lg n)$, which is a.s. $n \lg\lg(n/d) + \mathcal{O}(n)$. What is interesting about this shape is that, using the traversal of Gagie et al. [6] to reach the leaves with values 0 to $m-1$, we need only reach m leaves at depth $\mathcal{O}(\lg m)$ (i.e., the leftmost m in the wavelet tree), and thus we need to traverse only $\mathcal{O}(m)$ wavelet tree nodes. Array D can be stored in plain form, but permuted so that it is aligned to the wavelet tree leaves, which allows determining each distinct document identifier in $\mathcal{O}(1)$ time.

Theorem 1. *Let $T[1..n] = S_1 \cdot S_2 \cdots S_d$ be the concatenation of d documents S_j and let l be the maximum length of a repeated string in any S_j. Let* CSA *be a compressed suffix array on T that searches for any pattern $P[1..m]$ in time $\mathsf{search}(m) \geq m$. Then we can store T in $|\mathsf{CSA}| + n(\lg d + \lg l + \mathcal{O}(1))$ bits such that the* ndoc *documents where $P[1..m]$ occurs can be listed in time $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc})$. If T is generated under Szpankowski's A2 model [16], then the space is $|\mathsf{CSA}| + n(\lg d + \lg\lg(n/d) + \mathcal{O}(1))$ bits.*

In particular, if we use the CSA of Belazzougui and Navarro [1], we recover the optimal time of Muthukrishnan's solution, using (in most cases) less space.

Corollary 1. *Under the conditions of Theorem 1, we can obtain $nH_k(T) + o(nH_k(T)) + n(\lg d + \lg l + \mathcal{O}(1))$ bits and $\mathcal{O}(m + \mathsf{ndoc})$ time, where $H_k(T)$ is the k-th order empirical entropy of T, for any $k \leq \alpha \lg_\sigma n$, σ the alphabet size of T, and $0 < \alpha < 1$ any constant.*

3.2 Document Listing in Repetitive Collections

Array ILCP has yet another property, which also makes it attractive for repetitive collections.

Lemma 2. *Let S be a string generated under Szpankowski's A2 model. Let T be formed by concatenating d copies of S, each terminated with the special symbol "\$", and then carrying out s edits (symbol insertions, deletions, or substitutions) at arbitrary positions in T (excluding the '\$'s). Then, a.s., the* ILCP *array of T is formed by $\rho \leq r + \mathcal{O}(s \lg(r+s))$ runs of equal values, where $r = |S|$.*

Proof. Before applying the edit operations, we have $T = S_1 \cdots S_d$ and $S_j = S\$$ for all j. At this point, ILCP is formed by at most $r+1$ runs of equal values, since the d equal suffixes $S_j[\mathsf{SA}_{S_j}[i]..r+1]$ must be contiguous in the suffix array SA of T, in the area $\mathsf{SA}[(i-1)d+1..id]$. Since the values $l = \mathsf{LCP}_{S_j}[i]$ are also equal, and ILCP values are the LCP_{S_j} values listed in the order of SA, it follows that $\mathsf{ILCP}[(i-1)d+1..id] = l$ forms a run, and thus there are $r+1 = n/d$ runs in ILCP. Now, if we carry out s edit operations on T, any S_j will be of length at most $r+s+1$. Consider an arbitrary edit operation at $T[k]$. It changes all the suffixes $T[k-h..n]$ for all $0 \leq h < k$. However, since a.s. the string depth of a leaf in the suffix tree of S is $\mathcal{O}(\lg(r+s))$ [16], the suffix will possibly be moved in SA only for $h = \mathcal{O}(\lg(r+s))$. Thus, a.s., only $\mathcal{O}(\lg(r+s))$ suffixes are moved in SA, and possibly the corresponding runs in ILCP are broken. Hence $\rho \leq r + \mathcal{O}(s \lg(r+s))$ a.s. □

This proof generalizes Mäkinen et al.'s [9] arguments, which hold for uniformly distributed strings S. There is also experimental evidence [9] that, in real-life text collections, a small change to a string usually causes only a small change to its LCP array. Next we design a document listing data structure whose size is bound in terms of ρ.

Let $\mathsf{LILCP}[1..\rho]$ be the array containing the partial sums of the lengths of the ρ runs in ILCP, and let $\mathsf{VILCP}[1..\rho]$ be the array containing the values in those runs. We can store LILCP as a bitvector $L[1..n]$ with ρ 1s, so that $\mathsf{LILCP}[i] = \mathsf{select}(L, i)$. Bitmap L can be stored using a structure by Okanohara and Sadakane [13] that requires $\rho \lg(n/\rho) + \mathcal{O}(\rho)$ bits and answers select queries in $\mathcal{O}(1)$ time[3]. For rank it requires $\mathcal{O}(\lg(n/\rho))$ time, but we can reduce it to $\mathcal{O}(\lg \lg n)$ by building a y-fast trie [19] on every $(\lg n)$th value of LILCP and completing the query with a binary search using select, adding $\mathcal{O}(\rho)$ bits.

With this representation, it holds that $\mathsf{ILCP}[i] = \mathsf{VILCP}[\mathsf{rank}(L, i)]$. We can map from any position i to its run $i' = \mathsf{rank}(L, i)$ in time $\mathcal{O}(\lg \lg n)$, and from any run i' to its starting position in ILCP, $i = \mathsf{select}(L, i')$, in constant time.

This is sufficient to emulate Sadakane's algorithm [15] on a repetitive collection. We will use RLCSA as the CSA. The sparse bitvector $B[1..n]$ marking the document beginnings in T will be represented just like L, so that it requires $d \lg(n/d) + \mathcal{O}(d)$ bits and lets us compute any value $D[i] = \mathsf{rank}(B, \mathsf{SA}[i])$ in time $\mathcal{O}(\lg \lg n + \mathsf{lookup}(n))$. Finally, we build an RMQ data structure on VILCP, requiring $2\rho + o(\rho)$ bits and without needing access to VILCP [3].

Assume we have already used RLCSA to find ℓ and r in $\mathcal{O}(\mathsf{search}(m))$ time. Now we compute $\ell' = \mathsf{rank}(L, \ell)$ and $r' = \mathsf{rank}(L, r)$, which are the endpoints of the interval $\mathsf{VILCP}[\ell'..r']$ containing the values in the runs in $\mathsf{ILCP}[\ell..r]$. Now we run the recursive RMQs algorithm on $\mathsf{VILCP}[\ell'..r']$. Each time we find a minimum at $\mathsf{VILCP}[i']$, we remap it to the run $\mathsf{ILCP}[i..j]$, where $i = \max(\ell, \mathsf{select}(L, i))$ and $j = \min(r, \mathsf{select}(L, i+1) - 1)$. For each $i \leq k \leq j$, we compute $D[k]$ using B and RLCSA as explained, mark it in $V[D[k]] \leftarrow 1$, and report it. Since we do not have access to the values in ILCP nor in VILCP, the condition to stop the recursion at some value i' is that $V[D[i]] = 1$ is already marked. We show next that this is correct as long as RMQ returns the leftmost minimum in the range and that we recurse first to the left and then to the right of each minimum $\mathsf{VILCP}[i']$ found.

Lemma 3. *Using the procedure described, we correctly find all the positions* $\ell \leq k \leq r$ *such that* $\mathsf{ILCP}[k] < m$.

Proof. Let $j = D[k]$ be the leftmost occurrence of document j in $D[\ell..r]$. By Lemma 1, among all the positions where $D[k'] = j$ in $D[\ell..r]$, k is the only one where $\mathsf{ILCP}[k] < m$. Since we find a minimum ILCP value in the range, and then explore the left subrange before the right subrange, it is not possible to find first another occurrence $D[k'] = j$, since it has a larger ILCP value and is to the right of k. Therefore, when $V[D[k]] = 0$, that is, the first time we find a $D[k] = j$, it must hold $\mathsf{ILCP}[k] < m$, and the same is true for all the other ILCP values in the run. Hence it is correct to list all those documents and mark them in V.

[3] Using a constant-time $\mathsf{rank}/\mathsf{select}$ data structure for their internal array H.

Conversely, whenever we find a $V[D[k']] = 1$, the document has already been reported, thus this is not its leftmost occurrence and then $\mathsf{ILCP}[k'] \geq m$ holds, as well as for the whole run. Hence it is correct to avoid reporting the whole run and to stop the recursion in the range, as the minimum value is already $\geq m$. $\square$

We have thus obtained our first result for repetitive collections:

Theorem 2. *Let $T = S_1 \cdot S_2 \cdots S_d$ be the concatenation of d documents S_j, and* RLCSA *be a suffix array on T, searching for any pattern $P[1..m]$ in time* $\mathsf{search}(m)$ *and accessing* $\mathsf{SA}[i]$ *in time* $\mathsf{lookup}(n)$. *Let ρ be the number of runs in the* ILCP *array of T. We can store T in $|\mathsf{RLCSA}| + \rho \lg(n/\rho) + \mathcal{O}(\rho) + d \lg(n/d) + \mathcal{O}(d)$ bits such that document listing takes $\mathcal{O}(\mathsf{search}(m) + \mathsf{ndoc} \cdot (\lg \lg n + \mathsf{lookup}(n)))$ time.*

3.3 Document Counting

Finally, array ILCP allows us to efficiently count the number of distinct documents where P appears, without listing them all. Sadakane [15] showed how to compute it in constant time adding just $2n + o(n)$ bits of space. With ILCP we can obtain a variant that is suitable for repetitive collections.

We represent VILCP using a skewed wavelet tree as in Section 3.1. We can visit the first m leaves in time $\mathcal{O}(m)$. Moreover, the traversal algorithm [6] tells us how many times each value $0 \leq l < m$ occurs in $\mathsf{VILCP}[\ell'..r']$. More precisely, we arrive at each leaf l with an interval $[\ell'_l, r'_l]$ such that $\mathsf{VILCP}[\ell'..r']$ contains from the ℓ'_lth to the r'_lth occurrences of value l in $\mathsf{VILCP}[\ell'..r']$. We store a reordering of the run lengths so that the runs corresponding to each value l are collected left to right in ILCP and stored aligned to the wavelet tree leaf l. Those are concatenated into another bitmap $L'[1..n]$ with ρ 1s, similar to L, which allows us, using $\mathsf{select}(L', \cdot)$, to count the total length spanned by the ℓ'_lth to r'_lth runs in leaf l. By adding the areas spanned over the m leaves, we count the total number of documents where P occurs. Note that we need to correct the lengths of runs ℓ' and r', as they may overlap the original interval $\mathsf{ILCP}[\ell..r]$.

Theorem 3. *Let $T = S_1 \cdot S_2 \cdots S_d$ be the concatenation of d documents S_j, and* RLCSA *a compressed suffix array on T that searches for any pattern $P[1..m]$ in time $\mathsf{search}(m) \geq m$. Let ρ be the number of runs in the* ILCP *array of T and l be the maximum length of a repeated substring inside any S_j. Then we can store T in $|\mathsf{RLCSA}| + \rho(\lg l + 2\lg(n/\rho) + \mathcal{O}(1))$ bits such that the number of documents where a pattern $P[1..m]$ occurs can be computed in time $\mathcal{O}(\mathsf{search}(m))$.*

4 Precomputed Document Listing

When the document collection is repetitive, the document array is also repetitive. Let $\mathsf{SA}[i..j]$ be a run in the suffix array, so that there is another area $\mathsf{SA}[i'..j']$, where $\mathsf{SA}[i + k] = \mathsf{SA}[i' + k] - 1$ for all $k \leq j - i$. Then $D[i + k] = D[i' + k]$ for all $k \leq j - i$, except for at most d cells in the entire array D [5]. Navarro

et al. [12] used this repetitiveness in grammar-based compression of the wavelet tree of D. We can also use it to compress the precomputed answers to document listing queries covering long intervals of suffixes.

Let v be a suffix tree node. We write SA_v to denote the interval of the suffix array covered by node v, and D_v to denote the set of distinct document identifiers occurring in the same interval of the document array. Given block size b and a constant $\beta \geq 1$, we build a sparse suffix tree that allows us to answer document listing queries efficiently. For any suffix tree node v, it holds that

1. $|\mathsf{SA}_v| < b$, and thus documents can be listed in time $\mathcal{O}(b \cdot \mathsf{lookup}(n))$ by using CSA and bitvector B; or
2. we can compute the set D_v as a union of some sets $D_{u_1}, \ldots, D_{u_k}$ of total size at most $\beta \cdot |D_v|$, where nodes $u_1, \ldots, u_k$ are in the sparse suffix tree.

We start by selecting suffix tree nodes $v_1, \ldots, v_L$, so that no selected node is an ancestor of another, and the intervals SA_{v_i} of the selected nodes cover the entire suffix array. Given node v and its parent w, we select v if $|\mathsf{SA}_v| \leq b$ and $|\mathsf{SA}_w| > b$, and store D_v with the node. These nodes become the leaves of the sparse suffix tree, and we assume that they are numbered from left to right. Next we proceed upward in the suffix tree. Let v be an internal node, $u_1, \ldots, u_k$ its children, and w its parent. If the total size of sets $D_{u_1}, \ldots, D_{u_k}$ is at most $\beta \cdot |D_v|$, we remove node v from the tree, and add nodes $u_1, \ldots, u_k$ to the children of node w. Otherwise we keep node v in the sparse suffix tree, and store D_v there.

Let $v_1, \ldots, v_L$ be the leaf nodes and $v_{L+1}, \ldots, v_{L+I}$ the internal nodes of the sparse suffix tree. We use grammar-based compression to replace frequent subsets in sets $D_{v_1}, \ldots, D_{v_{L+I}}$ with grammar rules expanding to those subsets. Given a set Z and a grammar rule $X \rightarrow Y$, where $Y \subseteq \{1, \ldots, d\}$, we replace Z with $(Z \cup \{X\}) \setminus Y$, if $Y \subseteq Z$. As long as $|Y| \geq 2$ for all grammar rules $X \rightarrow Y$, each set D_{v_i} can be decompressed in $\mathcal{O}(|D_{v_i}|)$ time.

When all rules have been applied, we store the reduced sets $D_{v_1}, \ldots, D_{v_{L+I}}$ as an array A of document and rule identifiers. The array takes $|A| \lg(d + n_R)$ bits of space, where n_R is the total number of rules. We mark the first cell in the encoding of each set with a 1 in a bitvector $B_A[1..|A|]$, so that set D_{v_i} can be retrieved by decompressing $A[\mathsf{select}(B_A, i), \mathsf{select}(B_A, i+1) - 1]$. The bitvector takes $|A|(1 + o(1))$ bits of space and answers select queries in $\mathcal{O}(1)$ time [13]. The grammar rules are stored similarly, in an array G taking $|G| \lg d$ bits and a bitvector $B_G[1..|G|]$ of $|G|(1 + o(1))$ bits separating the array into rules (note that right hand sides of rules are formed only by terminals).

In addition to the sets and the grammar, we also have to store the sparse suffix tree. Bitvector $B_L[1..n]$ marks the first cell of interval SA_{v_i} for all leaf nodes v_i, allowing us to convert interval $\mathsf{SA}[\ell, r]$ into a range of nodes $[ln, rn] = [\mathsf{rank}(B_L, \ell), \mathsf{rank}(B_L, r+1) - 1]$. By using the same bitvector as for LILCP in Section 3.2, we can store B_L in $L \lg(n/L) + \mathcal{O}(L)$ bits and answer rank queries in $\mathcal{O}(\lg \lg n)$ time and select queries in constant time. Another bitvector $B_F[1..L{+}I]$ of $(L + I)(1 + o(1))$ bits marks the nodes that are the first children of their respective parents, supporting rank queries in constant time [13]. Array F of $I \lg I$ bits stores pointers to parent nodes, so that if node v_i is a first child, its

```
function listDocuments(ℓ, r)
    (res, ln) ← (∅, rank(B_L, ℓ))
    if select(B_L, ln) < ℓ:
        r' ← min(select(B_L, ln + 1) − 1, r)
        (res, ln) ← (list(ℓ, r'), ln + 1)
        if r' = r: return res
    rn ← rank(B_L, r + 1) − 1
    if select(B_L, rn + 1) ≤ r:
        ℓ' ← select(B_L, rn + 1)
        res ← res ∪ list(ℓ', r)
    return res ∪ decompress(ln, rn)

function decompress(ℓ, r)
    (res, i) ← (∅, ℓ)
    while i ≤ r:
        next ← i + 1
        while B_F[i] = 1:
            (i', next') ← parent(i)
            if next' > r + 1: break
            (i, next) ← (i', next')
        res ← res ∪ set(i)
        i ← next
    return res

function parent(i)
    par ← F[rank(B_F, i)]
    return (par + L, N[par])

function set(i)
    res ← ∅
    ℓ ← select(B_A, i)
    r ← select(B_A, i + 1) − 1
    for j ← ℓ to r:
        if A[j] ≤ d: res ← res ∪ {A[j]}
        else: res ← res ∪ rule(A[j] − d)
    return res

function rule(i)
    ℓ ← select(B_G, i)
    r ← select(B_G, i + 1) − 1
    return G[ℓ...r]

function list(ℓ, r)
    res ← ∅
    for i ← ℓ to r:
        res ← res ∪ {rank(B, SA[i])}
    return res
```

Fig. 1. Pseudocode for document listing using precomputed answers. Function listDocuments(ℓ, r) lists the documents from interval $\mathsf{SA}[\ell, r]$; decompress(ℓ, r) decompresses the sets stored in nodes $v_\ell, \ldots, v_r$; parent(i) returns the parent node and the leaf node following it for a first child v_i; set(i) decompresses the set stored in v_i; rule(i) expands the ith grammar rule; and list(ℓ, r) lists the documents from interval $\mathsf{SA}[\ell, r]$ by using CSA and bitvector B.

parent node is v_j, where $j = L + F[\mathsf{rank}(B_F, i)]$. Finally, array N of $I \lg L$ bits stores a pointer to the leaf node following each internal node.

Figure 1 contains pseudocode for document listing using the precomputed answers. Function list(ℓ, r) takes $\mathcal{O}((r + 1 - \ell)(\lg \lg n + \mathsf{lookup}(n)))$ time, set(i) takes $\mathcal{O}(|D_{v_i}|)$ time, and parent(i) takes $\mathcal{O}(1)$ time. Function decompress(ℓ, r) requires $\mathcal{O}(|res|)$ time to decompress the sets. Traversing the tree takes additional $\mathcal{O}(h)$ time per decompressed set, where h is the height of the sparse suffix tree. As each set contains at least one document, and we may have to list each document up to β times, this sums to $\mathcal{O}(\beta h \cdot |res|)$ time in the worst case. Hence the total time for listDocuments(ℓ, r) is $\mathcal{O}(\mathsf{ndoc} \cdot \beta h + \lg \lg n)$, if the answer has been precomputed, and $\mathcal{O}(b \cdot (\lg \lg n + \mathsf{lookup}(n)))$ otherwise.

5 Experiments

We implemented the document listing approaches described in preceding sections, and measured their performance on two datasets. All experiments were

	High		Medium		Low	
	Mean	SD	Mean	SD	Mean	SD
FIWIKI, ndoc	1810.8	1369.8	602.7	654.9	327.0	556.7
FIWIKI, ratio	32.04	378.62	4.26	22.72	1.75	2.46
INFLUENZA, ndoc	111021.3	29379.4	69666.5	19056.8	46304.3	17082.8
INFLUENZA, ratio	1.55	0.26	1.23	0.08	1.11	0.06

Table 1. Means and standard deviations (SD) of ndoc and the ratio $\frac{occ}{ndoc}$ for the pattern sets.

run on an Intel i7 860 2.8 GHz (8192 KB cache), with 16 GB RAM, running Ubuntu 12.04 and compiling with gcc-4.6.3 `-O3`.

Test data. We used two repetitive text collections. FIWIKI is a 400 MB prefix of Finnish Wikipedia version history. Each version of each Wikipedia article is considered a separate document, giving 20,433 documents. INFLUENZA is composed of genomes of the influenza virus, totaling 321.2 MB, and 227,356 documents.

Test patterns. Let occ be the number of times a pattern occurs in the whole collection, and recall ndoc is the number of documents containing the pattern. Document listing queries for patterns with similar occ and ndoc are easily handled by just enumerating all the positions of pattern occurrences (with the RLCSA) and mapping them to document identifiers. This approach however becomes less feasible as the separation between occ and ndoc grows, and at some point specialized document listing approaches become necessary. With this in mind, for each collection we constructed three sets of patterns as follows. First, we listed all patterns of length k present, and then ordered the patterns in descending order by value $occ - ndoc$, picking specific intervals of this list for testing.

For FIWIKI, the pattern length is 8, and each pattern set contains 20,000 patterns, starting at ranks 1,001, 40,001 and 100,001 of the full list of patterns. For INFLUENZA, the pattern length is 6, the set size 1000, and starting ranks are 1, 1,001 and 2,001. We call these three sets in both collections the *high*, *medium* and *low* pattern sets, respectively. Table 1 gives pattern statistics.

Results. Figure 2 shows the space-time tradeoff achieved by our document listing methods. The interleaved LCP array approach (Section 3) is called ilcp, and values following underscores represent the RLCSA sample rate. The precomputed document listing approach (Section 4) is called pdl, and values following underscores represent block size and the β value.

As a baseline we measured the time for a brute force (brute) approach, which simply enumerates pattern occurrences with the RLCSA, collecting distinct documents. This approach adds no space to the index. Like ilcp, brute's tradeoff comes from the sample period of the RLCSA.

Our first observation is that the new approaches achieve small space overhead, particularly on the FIWIKI set. Specifically, the RLCSA with sample period 128 takes 29 MB and 27 MB for the FIWIKI and INFLUENZA collections, respectively (about 7% and 8% of the uncompressed collection sizes). Including such

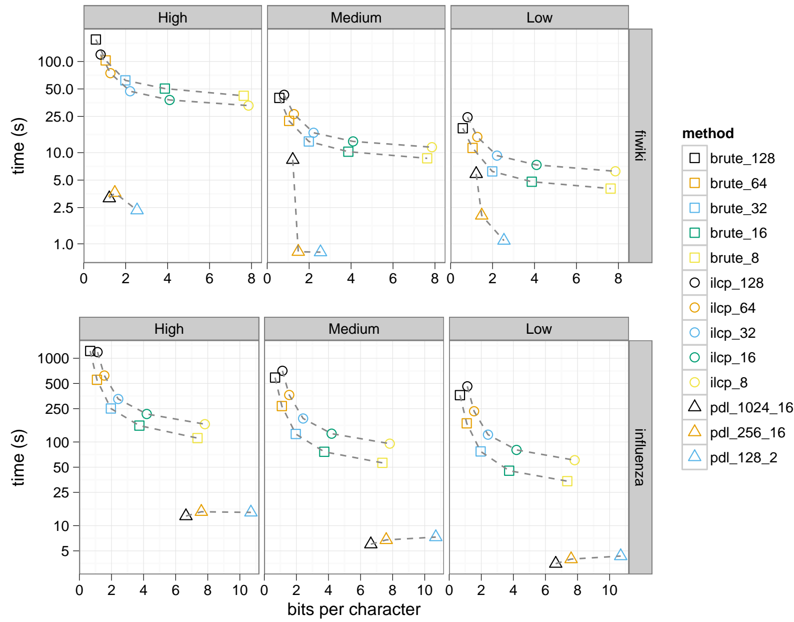

Fig. 2. Document listing times and memory required by different document listing approaches. Bits per character are shown on the x-axis, and time taken to list the documents on the y-axis (note the logarithmic scale). The time taken to find suffix array intervals corresponding to each pattern is not included in times shown here.

RLCSA, ilcp took 40 MB and 45 MB (about 10% and 14%). With block size $b = 1024$ and $\beta = 16$, pdl took 61 MB and 267 MB (about 15% and 83%).

With respect to query time, pdl significantly outperforms ilcp and brute on both data sets and is around an order of magnitude faster than the others when memory is equated. On the other hand ilcp is beaten by brute, except when the separation between occ and ndoc becomes large (the *high* FIWIKI pattern set).

Our most important experimental result is that, on the FIWIKI collection, pdl speeds up document listing by around an order of magnitude over brute while still using total space that is only a fraction of the uncompressed collection size. We were unable to compare to more sophisticated document listing techniques [12] designed for non-highly-repetitive collections because we could not construct them on our data sets. We leave an extensive comparison for the full paper.

6 Conclusions

We have described two approaches to document listing in highly repetitive collections — using an interleaved LCP array (ilcp) and precomputed document listing (pdl) — and shown that, on some representative collections, pdl signifi-

cantly reduces the query time of a brute-force solution, while still using only a fraction of the space of the uncompressed collection.

Aside from further experimental analysis, there are many directions for future work. Probably the most interesting one is to apply the ilcp approach over faster document listing indices, such as the wavelet tree of Theorem 3, which would yield an interesting space/time tradeoff.

Acknowledgements. We thank Giovanni Manzini for suggesting this line of research, Veli Mäkinen and Jorma Tarhio for helpful discussions, Cecilia Hernández for her grammar compressor, and Meg Gagie for righting our own grammar.

References

1. D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. ESA*, LNCS 6942, pages 748–759, 2011.
2. F. Claude and G. Navarro. Improved grammar-based compressed indexes. In *Proc. SPIRE*, LNCS 7608, pages 180–192, 2012.
3. J. Fischer. Optimal succinctness for range minimum queries. In *Proc. LATIN*, pages 158–169, 2010.
4. T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. LATA*, LNCS 7183, pages 240–251, 2012.
5. T. Gagie, G. Navarro, and S. J. Puglisi. Colored range queries and document retrieval. In *Proc. SPIRE*, pages 67–81, 2010.
6. T. Gagie, G. Navarro, and S.J. Puglisi. New algorithms on wavelet trees and applications to information retrieval. *Theor. Comp. Sci.*, 426-427:25–41, 2012.
7. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 636–645, 2003.
8. W.-K. Hon, R. Shah, and J. Vitter. Space-efficient framework for top-k string retrieval problems. In *Proc. FOCS*, pages 713–722, 2009.
9. V. Mäkinen, G. Navarro, J. Sirén, and N. Valimäki. Storage and retrieval of highly repetitive sequence collections. *J. Computational Biology*, 17(3):281–308, 2010.
10. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.
11. S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. SODA*, pages 657–666, 2002.
12. G. Navarro, S. J. Puglisi, and D. Valenzuela. Practical compressed document retrieval. In *Proc. SEA*, LNCS 6630, pages 193–205, 2011.
13. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. ALENEX*, 2007.
14. M. Pătraşcu. Succincter. In *Proc. FOCS*, pages 305–313, 2008.
15. K. Sadakane. Succinct data structures for flexible text retrieval systems. *J. Disc. Alg.*, 5(1):12–22, 2007.
16. W. Szpankowski. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Comput.*, 22(6):1176–1198, 1993.
17. N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. CPM*, LNCS 4580, pages 205–215, 2007.
18. P. Weiner. Linear pattern matching algorithm. In *Proc. SAT*, pages 1–11, 1973.
19. D. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Pr. Lett.*, 17(2):81–84, 1983.