Locally Compressed Suffix Arrays

Rodrigo González and Gonzalo Navarro
Dept. of Computer Science, University of Chile.
{rgonzale,gnavarro}@dcc.uchile.cl

Compressed text (self-)indexes have matured up to a point where they can replace a text by a data structure that requires less space and, in addition to giving access to arbitrary text passages, support indexed text searches. At this point those indexes are competitive with traditional text indexes (which are very large) for *counting* the number of occurrences of a pattern in the text. Yet, they are still hundreds to thousands of times slower when it comes to *locating* those occurrences in the text. In this paper we introduce a new, local, compression scheme for suffix arrays which permits locating the occurrences extremely fast, while still being much smaller than classical indexes. The core of our contribution is the identification of the regularities exploited by the compression based on function Ψ, used for long time in compressed text indexing, with those exploited by Re-Pair on the differential suffix array. The latter enjoys the locality properties that the former methods lack. As another consequence of this locality, we show that our index can be implemented in secondary memory, where its access time improve thanks to compression, instead of worsening as is the norm in other self-indexes. Finally, some byproducts of our work, such as a compressed dictionary representation for Re-Pair, can be of independent interest.

Categories and Subject Descriptors: F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*Pattern matching, Computations on discrete structures, Sorting and searching*; H.2.1 [**Database management**]: Physical design—*Access methods*; H.3.2 [**Information storage and retrieval**]: Information storage—*File organization*; H.3.3 [**Information storage and retrieval**]: Information search and retrieval—*Search process*

General Terms: Algorithms

Additional Key Words and Phrases: Compressed text indexing, compressed suffix arrays, Re-Pair

1. INTRODUCTION AND RELATED WORK

Compressed text indexing has become a popular alternative to cope with the problem of giving indexed access to large text collections without using up too much space. Reducing space is important because it gives one the chance of maintaining the whole collection in main memory. The current trend in compressed indexing is *full-text compressed self-indexes* [Navarro and Mäkinen 2007; Ferragina and Manzini 2005; Grossi et al. 2003; Sadakane 2003; Navarro 2004; Ferragina et al. 2007]. Such a self-index (for short) replaces the text by providing fast access to arbitrary text substrings, and in addition gives indexed access to the text by sup-

Partially supported by Yahoo! Research grant "Compressed data structures" (Chile) and Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile. An early partial version of this article appeared in *Proc. 18th Combinatorial Pattern Matching (CPM)*, LNCS 4580, pp. 216–227, 2007.

porting fast search for the occurrences of arbitrary patterns. These indexes take little space, usually from 30% to 150% of the text size (note that this includes the text). This is to be compared with classical indexes such as suffix trees [Weiner 1973] and suffix arrays [Manber and Myers 1993], which require at the very least 10 and 4 times, respectively, the space of the text, plus the text itself. In theoretical terms, to index a text $T = t_1 \dots t_n$ over an alphabet of size σ, the best self-indexes require $nH_k + o(n \log \sigma)$ bits for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$, where $H_k \leq \log \sigma$ is the k-th order empirical entropy of T [Manzini 2001; Navarro and Mäkinen 2007][1]. Just the uncompressed text alone would need $n \log \sigma$ bits, and classical indexes require $O(n \log n)$ bits on top of it.

The search functionality is given via two operations. The first is, given a pattern $P = p_1 \dots p_m$, *count* the number of times P occurs in T. The second is to *locate* the occurrences, that is, list their positions in T. Current self-indexes achieve a counting performance that is comparable in practice with that of classical indexes. In theoretical terms, for the best self-indexes the complexity is $O(m(1 + \frac{\log \sigma}{\log \log n}))$ and even $O(1 + \frac{m}{\log_\sigma n})$, compared to $O(m \log \sigma)$ of suffix trees and $O(m \log n)$ or $O(m + \log n)$ of suffix arrays. Locating, on the other hand, is far behind, hundreds to thousands of times slower than their classical counterparts. While classical indexes pay $O(occ)$ time to locate the occ occurrences, self-indexes pay $O(occ \log^\varepsilon n)$, where ε can in theory be any constant larger than zero but is in practice larger than 1. Worse than that, the memory access patterns of self-indexes are highly non-local, which makes their potential secondary-memory versions rather unpromising. Extraction of arbitrary text portions is also quite slow and non-local compared to having the text directly available as in classical indexes. The only implemented self-index which has more local accesses and faster locate is the LZ-index [Navarro 2004], yet its counting time is not competitive.

In this paper we propose a suffix array compression technique that builds on well-known regularity properties that show up in suffix arrays when the text they index is compressible (more precisely, the runs in the Ψ function of the suffix array, or which is the same, the number of equal-letter runs in the Burrows-Wheeler transform of the text [Navarro and Mäkinen 2007]). This regularity has been exploited in several ways in the past [Mäkinen 2003; Grossi and Vitter 2006; Grossi et al. 2003; Sadakane 2003; Mäkinen and Navarro 2005], but we present a completely novel technique to take advantage of it. We represent the suffix array using differential encoding, which converts the regularities into true repetitions. Those repetitions are then factored out using Re-Pair [Larsson and Moffat 2000], a compression technique that builds a dictionary of phrases and permits fast *local* decompression using only the dictionary (whose size one can limit at will, at the expense of losing some compression). We then introduce novel techniques to further compress the Re-Pair dictionary, which can be of independent interest. We also use specific suffix array properties to obtain a much faster compression method that loses just up to 1% of compression ratio.

Our so-called *locally compressed suffix array (LCSA)* is shown to reduce the suffix array to 20–70% of its original size, depending on the compressibility of various text types. This reduced index can still extract any portion of the suffix array very fast

[1] In this paper log stands for $\log_2$.

by adding a small set of sampled absolute values. By using the deep connection with function Ψ, we prove that the size of the result is $O(H_k \log \frac{1}{H_k}\ n \log n) + o(n)$ bits[2] for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. Note that this reduced suffix array is not yet a self-index as it cannot reproduce the text.

This structure can be used in two ways. One way is to attach it to a self-index able of counting, which in this process identifies as well the segment of the (virtual) suffix array where the occurrences lie. We can then locate the occurrences by decompressing that segment using our structure. The result is a self-index that needs 1–3 times the text size (that is, considerably larger than current self-indexes but also much smaller than classical indexes) and whose counting and locating times are competitive with those of classical indexes, far better for locating than current self-indexes. In theoretical terms, assuming for example the use of an alphabet-friendly FM-index [Ferragina et al. 2007] for counting, our index needs $O(H_k \log \frac{1}{H_k}\ n \log n + n)$ bits of space, counts in time $O(m(1+\frac{\log \sigma}{\log \log n}))$ and locates the *occ* occurrences of P in time $O(occ + \log n)$. In practice, even letting classical self-indexes use the same amount of space to speed up their locating time, they are much slower than our LCSA for reporting more than a few occurrences (2–10).

A second and simpler way to use the structure is, together with the plain text, as a replacement of the classical suffix array. In this case we must not only use it for locating the occurrences but also for binary searching. The binary search is done over the samples first and then we decompress the area between two consecutive samples to finish the search. This yields a very practical alternative requiring 0.8–2.4 times the text size (as opposed to 4) plus the text, in exchange for being just 2–28 times slower for locating.

On the other hand, if the text is very large, even a compressed index must reside on disk. Performing well on secondary memory with a compressed index has proved extremely difficult, because of their non-local access pattern. Thanks to its local decompression properties, our reduced suffix array performs very well on secondary memory. It needs the optimal $\lceil \frac{occ}{B} \rceil$ disk accesses for locating the *occ* occurrences, being B the disk block size measured in integers. On average, if the compression ratio (compressed divided by uncompressed suffix array size) is $0 \leq c \leq 1$, we perform $\lceil \frac{c \cdot occ}{B} \rceil$ accesses. That is, our index actually performs *better*, not worse (as it seems to be the norm), thanks to compression. We show how to build this structure almost I/O-optimally in secondary memory for large suffix arrays.

Achieving compressed indexes able of counting in competitive time was the first important breakthrough in this area. We believe our work is a first important step towards compressed indexes with practical locating times. This is up to date the major concern for adopting compressed indexes in practical applications.

This paper is organized as follows. In Section 2 we describe our LCSA. In section 3 we analyze its compression ratio, relating it to the compressibility of the text. In Section 4 we show how to use the LCSA as part of various indexing schemes. In Section 5 we carry out several experimental tunings and comparisons with alternative compressed and classical indexes. In Section 6 we give a secondary memory

[2]This result is meaningful for $H_k = o(1)$. The general result is $O(N(1 + \log \frac{n}{N}) \log n)$ bits, where N is the number of runs in Ψ. Acording to Mäkinen and Navarro [2005], $N \leq \min(n, nH_k + \sigma^k)$ for any k.

construction algorithm for the LCSA when the structures do not fit in main memory. Finally, in Section 7 we give our conclusions and future work directions.

2. A LOCALLY COMPRESSED SUFFIX ARRAY (LCSA)

Given a text $T = t_1 \dots t_n$ over alphabet Σ of size σ, where for technical reasons we assume $t_n = \$$ is smaller than any other character in Σ and appears nowhere else in T, a *suffix array* $A[1,n]$ is a permutation of $[1,n]$ such that $T_{A[i],n} \prec T_{A[i+1],n}$ for all $1 \leq i < n$, being "$\prec$" the lexicographical order. By $T_{j,n}$ we denote the *suffix* of T that starts at position j. Since all the occurrences of a pattern $P = p_1 \dots p_m$ in T are prefixes of some suffix, a couple of binary searches in A suffice to identify the segment in A of all the suffixes that start with P, that is, the segment pointing to all the occurrences of P. Thus the suffix array permits counting the occurrences of P in $O(m \log n)$ time and reporting the *occ* occurrences in $O(occ)$ time. With an additional array of integers, the counting time can be reduced to $O(m + \log n)$ [Manber and Myers 1993].

Suffix arrays turn out to be compressible whenever T is. The k-th order empirical entropy of T, H_k [Manzini 2001], shows up in A in the form of large segments $A[i, i+\ell]$ that appear elsewhere in $A[j, j+\ell]$ with all the values shifted by one position, $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$. Actually, one can partition A into *runs* of maximal segments that appear repeated (shifted by 1) elsewhere, and the number of such runs is at most $nH_k + \sigma^k$ for any k [Mäkinen and Navarro 2005; Navarro and Mäkinen 2007].

This property has been used several times in the past to compress A. Mäkinen's Compact Suffix Array (CSA) [Mäkinen 2003] replaces runs with pointers to their definition (copy) elsewhere in A, so that the run can be recovered by (recursively) expanding the definition and shifting the values. Mäkinen and Navarro [2005] use the connection with FM-indexes (runs in A are related to equal-letter runs in the Burrows-Wheeler transform of T, basic building block of FM-indexes) and run-length compression. Yet, the most successful technique to take advantage of those regularities has been the definition of function $\Psi(i) = A^{-1}[A[i]+1]$ (or $A^{-1}[1]$ if $A[i] = n$). It can be seen that $\Psi(i) = \Psi(i-1)+1$ within runs of A, and therefore a differential encoding of Ψ is highly compressible [Grossi et al. 2003; Sadakane 2003].

2.1 Basic LCSA Idea

We present a completely different method to compress A. We first represent A in differential form A':

DEFINITION 1. *Let $A[1,n]$ be an array of integers. Then we define $A'[1,n]$ as follows: $A'[1] = A[1]$ and $A'[i] = A[i] - A[i-1]$ for all $1 < i \leq n$.*

The next simple lemma shows that runs of A become true repetitions in A'.

LEMMA 1. *Consider a run of A of the form $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$. Then $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$.*

PROOF. $A'[j+s] = A[j+s] - A[j+s-1] = (A[i+s]+1) - (A[i+s-1]+1) = A[i+s] - A[i+s-1] = A'[i+s]$. □

We can now exploit those repetitions using any classical compression method. In particular, we seek a method that allows fast local decompression of A'.

We resort to Re-Pair [Larsson and Moffat 2000], a dictionary-based compression method based on the following algorithm:

(1) Identify the most frequent pair $A'[i]A'[i+1]$ in A', let ab be such pair.
(2) Create a new integer symbol s, larger than all existing symbols in A', and add rule $s \to ab$ to a dictionary R.
(3) Replace every occurrence of ab in A by s.[3]
(4) Iterate until every pair has frequency 1.

The result of the compression algorithm is the dictionary of rules R plus the sequence C of (original and new) symbols into which A' has been compressed. Note that R can be easily stored as a vector of pairs, so that rule $s \to ab$ is represented by $R[s-n] = a : b$.

Any portion of C can be easily decompressed in optimal time and fast in practice. To decompress $C[i]$, we first check if $C[i] \leq n$. If it is, then it is an original symbol of A' and we are done. Otherwise, we obtain both symbols from $R[C[i]-n]$, and expand them recursively (they can in turn be original or created symbols, and so on). We reproduce u cells of A' in $O(u)$ time, and the accesses pattern is local if R is small.

Since R grows by 2 integers (a, b) for every new pair, we could stop creating pairs when the most frequent one appears only twice. R can be further reduced by preempting this process, which trades its size for overall compression ratio.

Apart from R and C, we need a few more structures to recover the values of A:

—An array S such that $S[i] = A[i \cdot l]$, that is, a sampling of absolute values of A at regular intervals l.
—A bitmap $L[1, n]$, marking the positions where each symbol of C (which could represent several symbols of A') starts in A'.
—$o(n)$ further bits to answer *rank* queries on L in constant time [Jacobson 1989; Navarro and Mäkinen 2007], where $rank(L, i)$ is the number of 1's in $L[1, i]$.

With this structure, the algorithm to retrieve $A[i, j]$ is as follows:

(1) Check if there is a multiple of l in $[i, j]$, extending i to the left or j to the right to include such a multiple if necessary.
(2) Use the mechanism to decompress one symbol in C (described above) to obtain $A'[i, j]$, by expanding $C[rank(L, i), rank(L, j)]$. We expand from right to left, so the first symbol may be not totally expanded.
(3) Use any absolute sample of A included in $S[\lfloor i/l \rfloor, \lfloor j/l \rfloor]$ to obtain, using the differences in $A'[i, j]$, the values $A[i, j]$.
(4) Return the values in the original requested interval $[i, j]$.

[3]If $a = b$ it might be impossible to replace all occurrences, e.g. aa in aaa. But, in such case, one can at least replace each other occurrence in a row.

The overall time complexity of this decompression is the output size plus what we have expanded the interval to include a multiple of l (i.e., $O(l)$) and to ensure an integral number of symbols in C. The latter can be controlled by limiting the length of the uncompressed version of the symbols we create.

2.2 Compression using Ψ

A weak point of using Re-Pair is its compression speed and space usage. Re-Pair can be implemented in $O(n)$ time, but this needs too much space [Larsson and Moffat 2000]. Instead of using the original Re-Pair, we opt for a technique that needs less space and usually runs in $O(n \log n)$ time. This technique is not too interesting (and we do not describe it) except as a control value to test our more important contribution: We introduce a fast approximate technique specialized to compressing suffix arrays A. We show that Ψ (which is easily built in $O(n)$ time from A) can be used to obtain a much faster compression algorithm, which in practice compresses almost as much as the original Re-Pair.

Recall that $\Psi(i)$ tells where in A is the value $A[i]+1$. The idea is that, if $A[i, i+\ell]$ is a run such that $A[j+s] = A[i+s]+1$ for $0 \leq s \leq \ell$ (and thus $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$), then $\Psi(i+s) = j+s$ for $0 \leq s \leq \ell$. Thus, by following permutation Ψ we have a good chance of finding repeated pairs in A'. The basic idea is to choose the pairs while following permutation Ψ, cycling several times over A', until no further replacements can be done. This does not guarantee to choose the same pairs of the original Re-Pair, but we expect them to be sufficiently good.

Data Structures. To compress using Ψ we need only two arrays and one bitmap.

—An array $D[1, n]$, which initially stores the suffix array of text T in differential form, $D[i] = A'[i]\ \forall i$. At the end, we compact the valid values of D to obtain C.

—An array $P[1, n]$, which initially stores the values of function Ψ of text T, $P[i] = \Psi(i)\ \forall i$.

—The bitmap $L[1, n]$, where $L[i] = 1$ indicates that $D[i]$ is a valid value. In the beginning $L[i] = 1\ \forall i$. At the end, L can be preprocessed for *rank* queries and is ready for querying.

When we replace a pair with a new symbol, array D becomes sparse. A way to find the next valid symbol in constant time is as follows: If a valid symbol $D[i]$ is followed by an invalid symbol $D[i+1]$ (that is, $L[i, i+1] = 10$), then $D[i+1]$ can be used to store the distance $i' - i$ to the next valid symbol $D[i']$ (we use i' with this meaning, for any i, in the next algorithm description). This permits obtaining any pair of the sparse D in constant time.

In practice, it turns out to be faster to calculate $i' = \mathit{selectnext}(L, i)$, which returns the position of the first 1 in $L[i+1, n]$, by scanning the bitmap word-wise.

Algorithm. We make a number of *passes* over D. Each pass starts at $i = 1$ (where value $A'[1] = A[1] = n$ will not be replaced by Re-Pair as it is unique). For each i visited along the pass, we see if $D[i]D[i'] = D[P[i]]D[P[i]']$. If this does not hold, we move on to $i \leftarrow P[i]$ and iterate. If, instead, equality holds, we start a chain of replacements: We add a new pair $s \rightarrow D[i]D[i']$ to R, make the replacements at i and $P[i]$ (invalidating $i+1$ and $P[i]+1$), and move on to $i \leftarrow P[i]$, continuing the

replacements until the pair changes. In this process, when a position $P[j]$ becomes invalid, we set $P[j] \leftarrow P[P[j]]$, so that the position is skipped in the next pass. When the pair finally changes, that is, $D[i]D[i'] \neq D[P[i]]D[P[i]']$, we restart the process with $i \leftarrow P[i]$, looking again for a new pair to create. We keep running passes over D (using P) as long as we replace at least $\alpha n'$ pairs in a pass, where $0 < \alpha < 1$ is a constant and n' is the number of valid elements in D in the previous pass. Figure 1 shows a more detailed pseudocode.

```
Algorithm Compress(D, P, α)
    s ← n, R ← ∅
    for i ← 1...n do L[i] ← 1
    n' ← n, rep ← 0
    do n' ← n' − rep
        rep ← 0
        j ← 1, j' ← selectnext(L, j)
        do
            do i ← j, i' ← j'
                while L[P[i]] = 0 do P[i] ← P[P[i]]
                j ← P[i], j' ← selectnext(L, j)
            while j ≠ 1 and D[i]D[i'] ≠ D[j]D[j']
            if j ≠ 1 then
                ab ← D[i]D[i'], s ← s + 1, R ← R ∪ {s → ab}
                D[i] ← s, L[i'] ← 0, rep ← rep + 1
                do D[j] ← s, L[j'] ← 0, rep ← rep + 1
                    i ← j, i' ← j'
                    while L[P[i]] = 0 do P[i] ← P[P[i]]
                    j ← P[i], j' ← selectnext(L, j)
                while j ≠ 1 and ab = D[j]D[j']
        while j ≠ 1
    while (rep > αn')
    j ← 1
    for i ← 1...n do
        if L[i] = 1 then D[j] ← D[i], j ← j + 1
    return (C[1, n'] = D, R, L)
```

Fig. 1. Algorithm to compress $D = A'$ using $P = \Psi$ in $O(n)$ time.

Cost. Let n_i be the number of elements in the i-th pass, then $n_{i+1} \leq (1-\alpha)n_i$. Since $n_0 = n$, it holds $n_i \leq (1-\alpha)^i n$. The i-th pass costs $O(n_i)$ time. Let k be the number of passes doing more than $\alpha n'$ replacements. So the total cost is at most

$$\sum_{i=0}^{k}(1-\alpha)^i n + (1-\alpha)^k n \;\leq\; n\sum_{i\geq 0}(1-\alpha)^i + (1-\alpha)^k n \;\leq\; \left(1+\frac{1}{\alpha}\right)n \;=\; O(n).$$

Thus our algorithm achieves linear time while requiring only the space for D (overwritten on A' and finally leaving there C), for P (overwritten on Ψ), and for L (which is also needed in the final structure). It is also simple and fast in practice.

2.3 Stronger Compression based on Ψ

The only advantage of using the original Re-Pair is that it yields better compression and enforces the property that each new rule in the dictionary removes no more

pairs than the previous rule. The latter comes from the fact that the pairs in Re-Pair are replaced in decreasing order of frequency. This prevents less frequent pairs to break longer chains of replacements. We now modify the algorithm that uses Ψ to obtain compression ratios as close to Re-Pair's as desired, at the expense of $O(n \log n)$ complexity (multiplied by a constant that increases as the compression ratio improves). The key idea is to replace longer chains first.

Algorithm. The algorithm is as follows:

—We make one pass searching for the longest chain of equal pairs obtained by following Ψ, let f be its length.

—We apply the previous algorithm, yet we only replace the chains of length at least $t_0 = \delta \cdot f$, where $0 < \delta < 1$ is a constant.

—Again we apply the previous algorithm using $t_1 = \delta \cdot t_0$ then $t_2 = \delta \cdot t_1$ and so on, until $t_i \leq \gamma$. At this point we decrement t_i one by one until we reach $t_i = 1$. Here γ is another parameter.

Cost. We already know that the total cost of all passes that replace more than $(1-\alpha)n'$ elements adds up to $O(n)$. The number of passes where we replace less than $(1-\alpha)n'$ pairs, on the other hand, is at most $\log_\delta f + \gamma$. This is, $\log_\delta f$ for the part where $t_{(\cdot)}$ decreases by a δ fraction, plus γ for the part where $t_{(\cdot)}$ decreases one by one. Thus the total cost is at most:

$$\frac{1}{\alpha}\, n + (\log_\delta f + \gamma)\, n.$$

Now, if we choose a constant s, $\alpha = 1/(s \cdot \log n)$, and $\gamma = \log n$, the total time is $O(n \log n)$. Choosing other values of s, δ and γ we obtain better complexities, but worsen the compression quality. Within $O(n \log n)$ complexity, we can improve the compression ratio by tuning δ and s.

2.4 Compressing the Dictionary

We now develop a technique to reduce the dictionary of rules R without affecting C. This can be of independent interest for Re-Pair in general. We note that the dictionary compression methods in the original Re-Pair article [Larsson and Moffat 2000] achieve much more compression. The advantage of our scheme is that we can decompress parts of the text without uncompressing the dictionary. This permits handling larger dictionaries in main memory.

A first observation is that, if we have a rule $s \to ab$ and s is only mentioned in another rule $s' \to sc$, then we could perfectly remove rule $s \to ab$ and rewrite $s' \to abc$. This gives a net gain of one integer, but now we have rules of varying length. This is easy to manage, but we prefer to go further. We develop a technique that permits *eliminating every rule definition that is used within R, once or more, and gain one integer for each such rule eliminated*. The key idea is to write down explicitly the binary tree formed by expanding the definitions (by doing a preorder traversal and writing 1 for internal nodes and 0 for leaves), so that not only the largest symbol (tree root) can be referenced later, but also any subtree.

For example, assume the rules $R = \{s \to ab,\ t \to sc,\ u \to ts\}$, and $C = tub$. We could first represent the rules by the bitmap $R_B = \texttt{100100100}$ (where s corresponds

to position 1, t to 4, and u to 7) and the sequence $R_S = ab1c41$ (we are using letters for the original symbols of A', and bitmap positions as the identifiers of created symbols). We express C as $47b$. To expand, say, 4, we go to position 4 in R_B and compute $rank_0(R_B, 4) = 2$ (number of zeros up to position 4, $rank_0(i) = i - rank(i)$). Thus the corresponding symbols in R_S start at position $2 + 1 = 3$. We extract one new symbol from R_S for each new zero we traverse in R_B, and stop when the number of zeros traversed exceeds the number of ones (this means we have completed the subtree traversal). This way we obtain the definition $1c$ for symbol 4.

More generally, R can be seen as $R = \{s_1 \to a_1b_1, s_2 \to a_2b_2, \ldots, s_k \to a_kb_k\}$, where indeed $s_k = n + k$ (as $n = A'[1] = A[1]$ is the maximum value in A'). Thus, we write down R_B, R_S and the new C as follows (note that positions in R_B are written in R_S shifted by n to distinguish them from the original symbols):

—$R_B = (100)^k$.

—$R_S = a_1b_1a_2b_2 \ldots a_kb_k = r_1r_2r_3 \ldots r_{2k}$, except that if $r_i > n$ we set it to $r_i = n + 1 + 3(r_i - n - 1)$, so that they point to the 1's in R_B.

—$C = c_1c_2 \ldots c_{n'}$ undergoes the same transformation: if $c_i > n$, we set it to $c_i = n + 1 + 3(c_i - n - 1)$.

Let us now reduce the dictionary, in our example, by expanding the definition of s within t (even if s is used elsewhere). The new bitmap is $R_B = \texttt{11000100}$ (where $t = 1$, $s = 2$, and $u = 6$), the sequence is $R_S = abc12$, and $C = 16b$. We can now remove the definition of t by expanding it within u. This produces the new bitmap $R_B = \texttt{1110000}$ (where $u = 1$, $t = 2$, $s = 3$), the sequence $R_S = abc3$ and $C = 21b$. Further reduction is not possible because u's definition is only used from C.[4] At the cost of storing at most $2|R|$ bits (for R_B), we can reduce R by one integer for each definition that is used at least once within R.

The reduction can be easily implemented in linear time, avoiding the successive renamings of our example, as follows: We first check for each rule if it is used within R, marking this in a bitmap U. Then we traverse R and only write down (the bits of R_B and the sequence R_S for) the entries that are not used within R. We recursively expand those entries, appending the resulting tree structure to R_B and leaf identifiers to R_S. Whenever we find a created symbol that does not yet have an identifier, we give it as identifier the current position in R_B and recursively expand it. We store these new identifiers in an array NV. Otherwise the expansion finishes and we write down a leaf (a `"0"`) in R_B and the identifier in R_S. Then we rewrite C using the renamed identifiers. Figure 2 shows detailed pseudocode.

We can further compress the dictionary, if we take into account that a rule only uses previous rules or original symbols. That is, the i-th rule can only point to elements with representation of length $\lceil \log_2 i \rceil$ bits. With a simple arithmetic computation we can directly access any rule.

Another way to further compress the dictionary, yet with a time penalty, is as follows: Instead of using the position i of a rule inside bitmap R_B, use $j =$

[4]It is tempting to replace u in C, as it appears only once, but our example is artificial: A symbol that is not mentioned in R must appear at least twice in C.

Algorithm Compress_Dictionary($R = \{s_1 \to a_1 b_1, \ldots, s_k \to a_k b_k\}$, $C = c_1 \ldots c_{n'}$)
  for $i \leftarrow 1 \ldots k$ **do** $U[i] \leftarrow 0$
  for $i \leftarrow 1 \ldots k$ **do**
    if $a_i > n$ **then** $U[a_i - n] \leftarrow 1$
    if $b_i > n$ **then** $U[b_i - n] \leftarrow 1$
  for $i \leftarrow 1 \ldots k$ **do** $NV[i] \leftarrow 0$
  $j \leftarrow 1$, $R_B \leftarrow \langle\rangle$, $R_S \leftarrow \langle\rangle$
  $LR_B \leftarrow 0$ // length in bits of bitmap R_B
  for $j \leftarrow 1 \ldots k$ **do**
    if $U[j] = 0$ **then** Expand_Rule(j)
  for $i \leftarrow 1 \ldots n'$ **do**
    if $c_i > n$ **then** $c_i \leftarrow NV[c_i - n] + n$
return (R_B, R_S, C)

Algorithm Expand_Rule(j)
  $R_B \leftarrow R_B : 1$, $LR_B \leftarrow LR_B + 1$
  $NV[j] \leftarrow LR_B$
  if $a_j \leq n$ **then**
    $R_S \leftarrow R_S : a_j$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
  else if $NV[a_j - n] > 0$ **then**
    $R_S \leftarrow R_S : NV[a_j - n] + n$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
  else Expand_Rule($a_j - n$)
  if $b_j \leq n$ **then**
    $R_S \leftarrow R_S : b_j$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
  else if $NV[b_j - n] > 0$ **then**
    $R_S \leftarrow R_S : NV[b_j - n] + n$, $R_B \leftarrow R_B : 0$, $LR_B \leftarrow LR_B + 1$
  else Expand_Rule($b_j - n$)

Fig. 2. Algorithm to compress the dictionary R and to update C in $O(n)$ time.R_B, R_S, NV, and LR_B act as global variables. "$\langle\rangle$" is the empty sequence and ":" the concatenation operator.

$rank_1(R_B, i)$. Given that j, we find the position in R_B where the rule starts with $i = select_1(R_B, j)$. We gain at least 1 bit per rule in the dictionary and in the text.

3. ANALYSIS OF COMPRESSION RATIO

We analyze the compression ratio of our data structure, first for the exact and then for our approximate method based on Ψ.

Let N be the number of runs in Ψ. As shown in [Mäkinen and Navarro 2005; Navarro and Mäkinen 2007], $N \leq \min(n, nH_k + \sigma^k)$ for any $k \geq 0$. Except for the first cell of each run, we have that $A'[i] = A'[\Psi(i)]$ within the run. Thus, we cut off the first cell of each run, to obtain up to $2N$ runs in A'. Every pair $A'[i]A'[i+1]$ contained in such runs must be equal to $A'[\Psi(i)]A'[\Psi(i)+1]$, thus the only pairs of cells $A'[i]A'[i+1]$ that are not equal to the "next" pair are those where i is the last cell of its run. This shows that there are at most $2N$ different pairs in A', as a traversal following Ψ permutation will change pairs only $2N$ times.

Exact Re-Pair. Since there are at most $2N$ different pairs, the most frequent pair appears at least $\frac{n}{2N}$ times. Because of overlaps, it could be that only each other occurrence can be replaced, thus the total number of replacements in the first iteration is at least βn, for $\beta = \frac{1}{4N}$.

After we choose and replace the most frequent pair, we end up with at most $n(1-\beta)$ integers in A'. The number of runs has not varied, because a replacement

cannot split a run. Thus, the same argument shows that the second time we end up with at most $n(1-\beta)^2$ symbols, and so on.

After M iterations, the length of C is at most $n(1-\beta)^M$ and the length of R is $2M$. Assume for a while $N/n \leq 1/4$, thus $\beta n \geq 1$. The total size is optimized for $M^* = \frac{\ln n + \ln\ln\frac{1}{1-\beta} - \ln 2}{\ln\frac{1}{1-\beta}}$, where it is $\frac{2(\ln n + \ln\ln\frac{1}{1-\beta} - \ln 2 + 1)}{\ln\frac{1}{1-\beta}}$. (Re-Pair shortens the total file size in each new iteration, so the final result cannot be worse than that after M^* iterations.) Since $\ln\frac{1}{1-\beta} = \ln\frac{4N}{4N-1} = \frac{1}{4N}(1+O(\frac{1}{N}))$, we have $M^* = \ln\frac{n}{4N} + O(1)$ and the total size is $8N\ln\frac{n}{4N} + O(N)$ integers. Since $N \leq H_k n + \sigma^k$, if we stick to $k \leq \alpha\log_\sigma n$ for any constant $0 < \alpha < 1$, it holds $\sigma^k = O(n^\alpha)$ and $N/n \leq H_k + O(n^{\alpha-1})$. If $H_k + O(n^{\alpha-1}) < 1/e$ the space formula is increasing as a function of N, thus the total space is $O((nH_k + O(n^\alpha))\log\frac{1}{H_k+O(n^{\alpha-1})} + nH_k + O(n^\alpha)) = O(nH_k\log\frac{1}{H_k+O(n^{\alpha-1})} + n^\alpha\log n)$ integers. As $h\log\frac{1}{h+x} = h\log\frac{1}{h} + h\log\frac{1}{1+x/h} = h\log\frac{1}{h} + O(x)$, the space is $O(nH_k\log\frac{1}{H_k} + n^\alpha\log n)$ integers. This is $O(H_k\log\frac{1}{H_k}\, n\log n) + o(n)$ bits, as even after the M^* replacements the numbers need $\Theta(\log n)$ bits. The result is interesting only for $N = o(n)$ and $H_k = o(1)$, in which case all of our conditions on N and H_k hold. Yet, note that the results hold anyway for larger H_k and N.

THEOREM 1. *After running exact Re-Pair, our structure represents A' using R and C in $O(N(1+\log\frac{n}{N}))$ integers, where N is the number of runs in Ψ. If $H_k = o(1)$, this is $O(H_k\log\frac{1}{H_k}\, n\log n) + o(n)$ bits, for any $k \leq \alpha\log_\sigma n$ and any constant $0 < \alpha < 1$. Otherwise the space is $O(n\log n)$ bits.*

As a comparison, Mäkinen's CSA [Mäkinen 2003] needs $O(H_k n\log n)$ bits of space [Navarro and Mäkinen 2007], which is always better as a function of H_k. Yet, both tend to the same space as H_k goes to zero. Other self-indexes are usually smaller.

Approximate Re-Pair. We now show that the simplified replacement methods of Sections 2.2 and 2.3 reach the same asymptotic space complexity. Assume as before that N and H_k are sufficiently small.

Just as for the exact method, the traversal using Ψ will create up to $2N$ pairs per pass. Assume for simplicity that, as we find each new pair in the traversal using Ψ, we always replace the pair, even if this involves creating it in R for just one occurrence in C (this is never better than the real algorithm). Thus we try to make all the $|A'|$ replacements, but we may fail because replacements overlap. That is, assume we have *abcd* and first replace $s \to bc$. In the new sequence *asd* we cannot make a replacement $s' \to ab$ nor $s' \to cd$. Indeed, in the best case we can carry out $\lfloor |A'|/2 \rfloor$ replacements, whereas in the worst case this is only $\lfloor |A'|/3 \rfloor$ (when we first choose all multiples of 3 as initial pair positions).

This shows that, in the first pass over Ψ, we add up to $4N$ integers to R and remove at least $n/3$ integers from A'. For the next pass, the key point is that the number of runs is still limited by $2N$: If we had that $A'[i] = A'[\Psi(i)]$, the fact stays valid after we replace both $A'[i]$ and $A'[\Psi(i)]$ by a new symbol (cells $A'[i+1]$ and $A'[\Psi(i)+1]$ are invalid for the next pass). Therefore we can analyze the process using recurrence $S(n) = 4N + S(2n/3)$. If we repeat the process i times and then call C the remaining cells of A', we get $S(n) = 4Ni + (2/3)^i n$, which is optimized

for $i^* = \log_{3/2}(n/4N) - 1$ iterations, where we get $S(n) = O(N \log \frac{n}{N})$ integers. Even after adding $O(Ni^*)$ new symbols these integers need $\Theta(\log n)$ bits.

Stronger approximate Re-Pair. This analysis is similar to that of exact Re-Pair. The relevant invariant, which is easy to check from the description of Section 2.3, is as follows: *The approximate algorithm always replaces a pair that appears at least $\delta \cdot f$ times, being f the frequency of the currently most frequent pair.* In this sense, the algorithm acts as a δ-approximation.

In exact Re-Pair, we first replace the most frequent pair, which appears at least $\frac{n}{2N}$ times. In this case, we first replace a pair that appears at least $\frac{\delta n}{2N}$ times. This gives us a total number of replacements in the first iteration of at least $\beta' n$, where $\beta' = \delta\beta = \frac{\delta}{4N}$. The same occurs at each stage of the algorithm. Applying the same arguments of the analysis of exact Re-Pair with this new β', and the fact that $0 < \delta < 1$ is a constant, we obtain the same result as in Theorem 1.

THEOREM 2. *The process of replacing pairs following permutation Ψ, in either of its variants, achieves a data structure that fits in $O(N(1+\log \frac{n}{N}))$ integers, where N is the number of runs in Ψ. If $H_k = o(1)$, this is $O(H_k \log \frac{1}{H_k}\ n \log n) + o(n)$ bits, for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. Otherwise, the space is $O(n \log n)$ bits.*

4. TOWARDS A TEXT INDEX

As explained in the Introduction, the LCSA is not by itself a text index. We explore now some alternatives to upgrade it to a full-text index.

4.1 A Smaller Classical Index

A simple and practical alternative is to use our LCSA just like the classical suffix array, that is, not only for locating but also for searching, keeping the text in uncompressed form as well. This is not really a compressed index, but a practical alternative to a classical index.

The binary search of the interval that corresponds to P will start over the absolute samples of our LCSA. Only when we have identified the interval between consecutive samples of A where the binary search must continue, we decompress the whole interval and finish the binary search. If the two binary searches finish in different intervals, we will also need to decompress the intervals in between for locating all the occurrences. For displaying, the text is at hand.

The cost of this search is $O(m \log n)$ plus the time needed to decompress the portion of A between two absolute samples. We can easily force the compressor to make sure that no symbol in C spans the limit between two such intervals, so that the complexity of this decompression can be controlled with the sampling rate l. For example, $l = O(\log n)$ guarantees a total search time of $O(m \log n + occ)$, just as the suffix array version that requires 4 times the text size (plus text).

THEOREM 3. *There exists a full-text index for text T of length n over an alphabet of size σ and k-th order entropy H_k, which requires $O(H_k \log \frac{1}{H_k}\ n \log n + n \log^{1-\epsilon} n)$ bits of space in addition to T, for any constant $0 \leq \epsilon \leq 1$, any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$. It can count the occurrences of a pattern of length m in time $O(m \log n)$ and locate its occ occurrences in time $O(occ + \log^\epsilon n)$.*

The theorem is obtained by considering that C and R use $O(H_k \log \frac{1}{H_k}\ n \log n + n^\alpha \log^2 n)$ bits, to which we must add $O((n/l)\log n)$ bits for the absolute samples in A', and the extra cost to limit the formation of symbols that represent very long sequences. If we limit such symbol lengths to l as well, we have an overhead of $O((n/l)\log n)$ bits, as this can be regarded as inserting spurious symbols every l positions in A' to prevent the formation of longer symbols. By choosing $l = \log^\varepsilon n$ we have $O(H_k \log \frac{1}{H_k}\ n \log n + n \log^{1-\varepsilon} n)$ bits of space. The time is $O(m \log n + \log^\varepsilon n)$ for counting, and $O(occ + \log^\varepsilon n)$ for locating the occurrences.

4.2 A Compressed Self-Index

Another choice to achieve a full-text index is to plug our LCSA to any of the many self-indexes able of giving the suffix array range of the occurrences of P within little space [Ferragina and Manzini 2005; Ferragina et al. 2007; Sadakane 2003; Grossi et al. 2003]. Given the area $[i, j]$ where the occurrences lie in A, locating the occurrences boils down to decompressing $A[i, j]$ from our LCSA structure.

To fix ideas, consider the alphabet-friendly FM-index [Ferragina et al. 2007]. It takes $nH_k + o(n \log \sigma)$ bits of space for any $k \leq \alpha \log_\sigma n$ and constant $0 < \alpha < 1$, and can count in time $O(m(1 + \frac{\log \sigma}{\log \log n}))$. Our additional structure dominates the space complexity, requiring $O(H_k \log \frac{1}{H_k}\ n \log n + n \log^{1-\varepsilon} n)$ bits.

Extracting substrings can be done with the same FM-index, but the time to display ℓ text characters is, using $n \log^{1-\varepsilon} n$ additional bits of space, $O((\ell + \log^\varepsilon n)(1 + \frac{\log \sigma}{\log \log n}))$. By using instead the structure proposed by González and Navarro [2006] we have other $nH_k + o(n \log \sigma)$ bits of space for $k = o(\log_\sigma n)$ (this space is asymptotically negligible) and can extract the characters in optimal time $O(1 + \frac{\ell}{\log_\sigma n})$.

THEOREM 4. *There exists a self-index for text T of length n over an alphabet of size σ and k-th order entropy H_k, which requires $O(H_k \log \frac{1}{H_k}\ n \log n + n \log^{1-\varepsilon} n) + o(n \log \sigma)$ bits of space, for any $0 \leq \varepsilon \leq 1$. It can count the occurrences of a pattern of length m in time $O(m(1 + \frac{\log \sigma}{\log \log n}))$ and locate its occ occurrences in time $O(occ + \log^\varepsilon n)$. For $k = o(\log_\sigma n)$ it can display any text substring of length ℓ in time $O(1 + \frac{\ell}{\log_\sigma n})$. For larger $k \leq \alpha \log_\sigma n$, for any constant $0 < \alpha < 1$, this time becomes $O((\ell + \log^\varepsilon n)(1 + \frac{\log \sigma}{\log \log n}))$.*

4.3 A Secondary Memory Index

In this section we design an LCSA structure that works in secondary memory, retrieving the *occ* occurrences with $\lceil \frac{occ}{B} \rceil$ accesses to disk. This is worst-case optimal.

Assume table R is small enough to fit in main memory. This always can be achieved at the price of losing some compression, by simply preempting the process when the size of the dictionary exceeds the alloted memory. During construction, by using an extra bit per rule, we can predict exactly the final size the dictionary R will have after applying the compression method of Section 2.4, without yet carrying out that compression: When we create a rule, its extra bit is set to 0 and the space usage is increased by three bits (R_B) plus two integers (R_S). Then, every time we use a rule a inside another one, we check the extra bit of a. If the bit is still 0, we set it to 1 and decrease the space usage of R by one bit (R_B) and one integer (R_S). Otherwise, we do nothing.

To retrieve an interval of the suffix array, we read its corresponding area of C from disk, and uncompress each cell in memory by using R without any further disk access. We need a sample of the suffix array inside each read block to undo the differential form of A'; this sample can be stored within the same disk block. To determine the disk block where the desired area of C lies, we also need an in-memory binary search over an array storing the absolute position of the first C cell of each disk block. All these amount to a couple of extra integer values per disk block, which is negligible (a B-tree-like structure could be used in the unlikely case the absolute positions array does not fit in main memory).

On average, if we achieved compression ratio $c \leq 1$, we will need to read $c \cdot occ$ cells from C, at a cost of $\lceil \frac{c \cdot occ}{B} \rceil$. Therefore, we achieve for the first time a locating complexity that is *better thanks to compression, not worse*. Note that Mäkinen's CSA [Mäkinen 2003] would not perform well at all under this scenario, as the decompression process is highly non-local.

González and Navarro [2007] describe an index that uses $O(\frac{n}{Bt} \cdot \sigma \log n + \frac{n}{B} \log n)$ bits in main memory and $nH_k(T) + O(\sigma^{k+1} \log n) + O(\frac{n}{B} \cdot \sigma \log(t \cdot B \log n))$ bits in secondary memory, where B is the number of bits in a disk block and t a parameter. It can identify the suffix array area containing the occurrences of a pattern of length m (and thus count its occurrences) using at most $2(m-1)$ accesses to disk. This structure can be enhanced to an index able of locating by plugging a secondary memory version of our LCSA.

To extract text passages of length ℓ in optimal time one can use compressed sequence mechanisms like that of González and Navarro [2006], which easily adapts to disk and offers local decompression [González and Navarro 2007].

Similarly, other secondary memory data structures relying on suffix arrays can benefit from replacing it by our secondary-memory LCSA. Some examples are the Compact Pat Tree (CPT) [Clark and Munro 1996], which represents a suffix tree in secondary memory in compact form, storing in its leaves the suffix array values; the String B-tree [Ferragina and Grossi 1999], based on a combination between B-trees and Patricia tries, which can be used as a suffix array; the backward-searched Compressed Suffix Array [Mäkinen et al. 2004], which uses Ψ on secondary memory for counting but has no good solution for locating; and the disk-based Suffix Array [Baeza-Yates et al. 1996], a suffix array on disk with some memory-resident structures that improve the the search, which needs at the end to complete the search on a chunk of the suffix array read from disk.

5. EXPERIMENTAL RESULTS

We present three series of experiments in this section. The first one regards compression performance, the second the use of our technique as a classical index using reduced space, and the third the use as a plug-in for boosting the locating performance of a compressed self-index.

We use text collections obtained from the *PizzaChili* site[5]. This site offers a collection of texts of various types and sizes. We use all the types available (XML, English, Source Code, Proteins, DNA and Pitches) and use sizes of 50MB and

[5] `http://pizzachili.dcc.uchile.cl/`

100MB for our experiments. The experiments were run on a Pentium IV, 2.0 GHz with 2GB RAM using Linux with kernel 2.4.31 and g++ with -O3 optimization.

Compression performance. In Section 2.2 we mentioned that the compression time of exact Re-Pair would be an issue, and gave two approximate methods based on Ψ which should be faster.

In this section we call RP our implementation of the exact Re-Pair compression algorithm[6], RPΨ_0 the Ψ-based approximation that runs in $O(n)$ time (Section 2.2), and RPΨ the Ψ-based approximation that runs in $O(n \log n)$ time (Section 2.3). We also include the methods RPSP, RPΨ_0SP, and RPΨSP. These forbid a rule to cross a 256-cell boundary. In all cases, we take absolute A' samples each 64 positions.

In practice, RPΨ_0 and RPΨ_0SP are very fast in comparison with the rest of the methods, yet compress less. Considering this, we could relax the stopping condition with the aim of better balancing these factors. In Figure 3 we show space reduction achieved from one pass to the next, using RPΨ_0SP. For example, in the case of `xml` (the one needing the most passes), we remove 45% of the file in the first pass, and 40% of that in the second, but after 8 passes the compression gain is less than 0.01%. Considering this, we force at least 8 passes after reaching less than $\alpha n'$ replacements per pass. We use $\alpha = 1/4$, which gives good results.

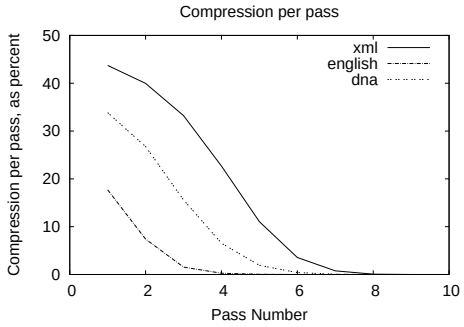

Fig. 3. Compression achived per pass using RPΨ_0SP. We use files of size 100MB.

To tune the parameters of the approximate variant RPΨSP, we test different values on two small files (`english` and `xml`, truncated to 50MB). We show, among several we carried out, the following experiments, as they best reflect the choice of parameters. Table I shows that the compression gain for increasing s loses importance for $s > 8$. Table II, on the other hand, shows that increasing δ does not give any gain on `english`, yet it slightly improves compression ratio on `xml`. A fair choice of parameters for RPΨSP, which we use for the rest of the experiments, is $s = 8$, $\delta = 3/4$ and $\gamma = \log n$.

Table III shows that the compression ratio varies widely. On `xml` data we achieve 23.5% compression (the reduced suffix array is smaller than the text!), whereas compression is extremely poor on `dna`. In many text types of interest we slash the

[6]Those we could find freely available did not work properly.

s	Compr. ratio `xml`	Compr. ratio `english`	Compr. time (sec) `xml`	Compr. time (sec) `english`
1	26.03%	55.47%	766	1,374
2	25.95%	55.35%	905	1,475
4	25.91%	55.32%	1,080	1,621
8	25.89%	55.30%	1,241	1,837
16	25.88%	55.29%	1,344	2,028
32	25.87%	55.28%	1,519	2,425
64	25.87%	55.28%	1,645	2,801
128	25.87%	55.28%	1,717	3,213

Table I. Compression ratio obtained using different values of s for the approximation RPΨSP. In this case we use $\delta = 1/2$. The percentage is computed by comparing with the $4n$ bytes required by a standard suffix array implementation.

δ	Compr. ratio `xml`	Compr. ratio `english`	Compr. time (sec) `xml`	Compr. time (sec) `english`
1/2	25.89%	55.30%	1,241	1,837
3/4	25.81%	55.29%	1,573	2,159
7/8	25.74%	55.29%	2,091	2,786
15/16	25.68%	55.29%	2,835	4,184
31/32	25.60%	55.29%	4,185	7,150

Table II. Compression obtained using different values of δ using approximation RPΨSP. In this case we use $s = 8$.

suffix array to around half of its size. Below the name of each collection we wrote the percentage H_3/H_0, which gives an idea of the compressibility of the collection independent of its alphabet size (e.g. it is very easy to compress `dna` to 25% because there are mainly 4 symbols but one chooses to spend a byte for each symbol in the uncompressed text, otherwise `dna` is almost incompressible). The measure turns out to be an excellent predictor of the compression, except for `proteins` where we are closer to H_5/H_0.

We exclude `dna` to state the following numbers, because of its poor compression ratio. The approximation RPΨ$_0$ runs up to 180 times faster and just loses 3.3%–17.8% in compression ratio compared to RP. The approximation RPΨ runs up to 25 times faster and just loses up to 3.5% in compression ratio. RP runs at 25 to 1000 sec/MB, RPΨ$_0$ runs at 5 to 10 sec/MB and RPΨ runs at 31 to 56 sec/MB. Suffix array construction is the same in all the methods and takes around 100 seconds overall in all cases. Thus, most of the indexing time shown in Table III is spent by the compression methods.

Other statistics are also available in Table III. In column 6 we measure the average length of a cell of C if we choose uniformly in A (longer cells are in addition more likely to be chosen for decompression). Those numbers explain the times obtained for the next series of experiments. Note that they are related to compressibility, but not as much as one could expect. Rather, the numbers obey to a more detailed structure of the suffix array: they are higher when the compression is not uniform across the array. In every case, we can limit the maximum length of a C cell. The SP variants show how this impacts compression ratio and decompression speed. We can see that their compression ratio is almost the same, worsening at most by 6.37% (RP), 12.68% (RPΨ$_0$), or 9.17% (RPΨ) on `dna` (and much less on others).

Coll. size (MB),H_3/H_0	Method	Index size (MB)	Compr. ratio	Compr. time (s)	Expected decompr.	Dict. compr.	Compr. 2% in RAM
`xml`,100, 26.28%	RP	93.56	23.39%	29,800	6,936.54	57.22%	34.08%
	RPSP	99.52	24.88%	25,472	134.91	58.29%	35.84%
	RPΨ_0	103.06	25.76%	625	7,570.49	57.46%	91.42%
	RPΨ_0SP	116.13	29.03%	651	83.79	58.68%	91.59%
	RPΨ	94.26	23.56%	3,547	6,948.85	57.18%	36.15%
	RPΨSP	102.90	25.73%	3,598	95.83	58.23%	40.63%
`dna`,100, 97.02%	RP	336.53	84.13%	8,511	5.01	79.49%	92.45%
	RPSP	337.11	84.28%	8,402	4.25	79.72%	92.56%
	RPΨ_0	342.52	85.63%	931	4.73	78.20%	99.17%
	RPΨ_0SP	343.11	85.78%	899	4.04	78.44%	99.18%
	RPΨ	336.37	84.09%	4,279	5.03	79.19%	93.19%
	RPΨSP	336.96	84.24%	4,260	4.28	79.41%	93.30%
`english`,100, 53.05%	RP	227.59	56.90%	87,285	238.31	59.27%	88.15%
	RPSP	230.04	57.51%	86,273	30.37	59.71%	88.33%
	RPΨ_0	249.03	62.26%	974	202.79	59.70%	98.56%
	RPΨ_0SP	252.08	63.02%	944	26.83	60.19%	98.56%
	RPΨ	227.74	56.94%	4,621	215.12	59.17%	88.92%
	RPΨSP	230.26	57.56%	4,600	29.99	59.60%	89.12%
`pitches`,50, 61.37%	RP	116.58	58.29%	11,454	33.96	69.51%	67.41%
	RPSP	117.61	58.81%	11,067	17.00	70.08%	67.78%
	RPΨ_0	126.56	63.28%	279	26.38	66.86%	97.32%
	RPΨ_0SP	127.83	63.91%	535	14.21	67.23%	97.34%
	RPΨ	117.98	58.99%	1,618	28.89	68.67%	70.17%
	RPΨSP	119.18	59.59%	1,807	15.66	69.00%	71.03%
`proteins`,100, 97.21%	RP	284.61	71.15%	2,642	58.98	79.72%	75.63%
	RPSP	285.94	71.48%	2,732	13.87	80.09%	76.00%
	RPΨ_0	294.08	73.52%	1,045	52.52	75.16%	92.13%
	RPΨ_0SP	296.24	74.06%	1,032	10.79	75.03%	92.30%
	RPΨ	285.94	71.49%	5,719	58.46	78.98%	76.77%
	RPΨSP	287.73	71.93%	5,764	11.92	78.80%	77.31%
`sources`,100, 40.74%	RP	154.88	38.72%	107,371	2,041.88	57.63%	65.16%
	RPSP	159.34	39.83%	103,292	60.48	58.33%	65.85%
	RPΨ_0	181.38	45.34%	684	1,778.79	58.09%	96.67%
	RPΨ_0SP	187.52	46.88%	677	50.97	58.80%	96.70%
	RPΨ	156.18	39.04%	4,380	1,928.86	57.48%	68.00%
	RPΨSP	161.21	40.30%	4,778	56.89	58.13%	69.15%

Table III. Index size and build time using Re-Pair and its Ψ-based approximations. We also include versions with rules up to length 256 (SP extension). Compression ratio compares with the $4n$ bytes needed by a plain suffix array representation.

In column 7 we show the compression ratio achieved on the dictionary part using the technique of Section 2.4, charging it the bitmap introduced as well. It can be seen that the technique is rather effective, approaching in some cases the limit of 50% to its possible effectiveness. (We remark that the compression ratios of previous columns do account for the dictionary space and all the necessary structures to operate.)

The last column shows how much compression we would achieve if the structures that must reside on RAM were limited to 2% of the original suffix array size (recall Section 4.3). We still obtain attractive compression performance on texts like `xml`, `sources` and `pitches` (recall that on secondary memory the compression ratio translates almost directly to decompression performance). As expected, RPΨ_0 does a much poorer job here, as it does not choose the best pairs early, but RPΨ achieves almost the same performance as RP.

A classical reduced index. We test RPΨSP (from now on LCSA) as a replacement of the suffix array, that is, adding it the text and using it for binary searching, as explained in Section 4.1. We compare it with a plain suffix array (SA) and

Mäkinen's CSA (MakCSA [Mäkinen 2003]), as the latter operates similarly.

Fig. 4 shows the result. MakCSA offers space-time tradeoffs, whereas those of our index (sample rate for absolute values) did not significantly affect the time. Our structure stands out as a relevant space/time tradeoff, especially when locating many occurrences (i.e., on short patterns). In particular, LCSA is usually noticeably faster than MakCSA for the same space, yet the latter is able of using less space (at least on `english`). Compared to a plain suffix array, LCSA requires 0.9–2.4 times the text size (as opposed to 4) plus the text, at the price of being 2–28 times slower for locating. Compared to current state of the art in compressed indexing, this slowdown is rather modest.

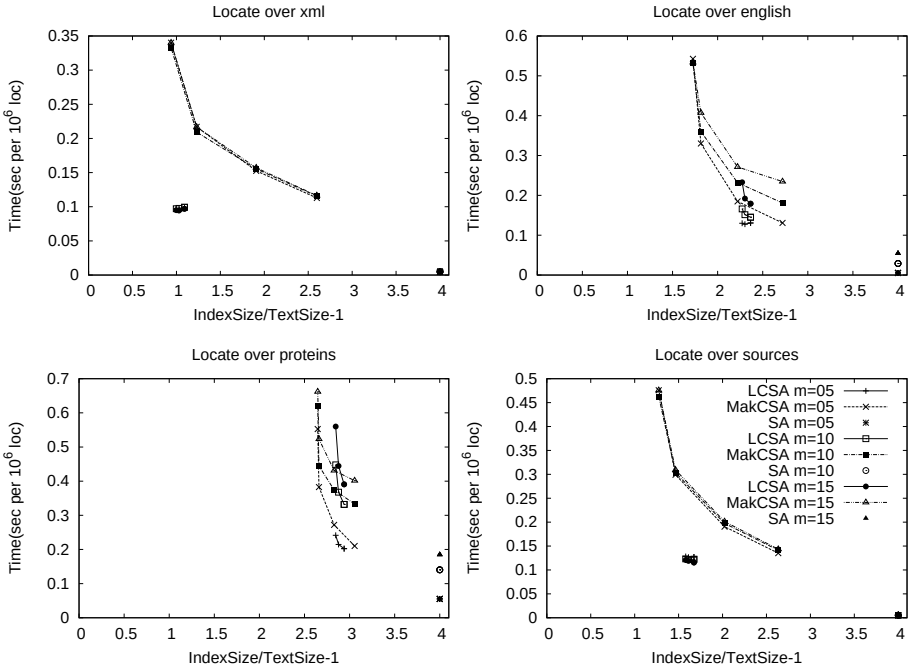

Fig. 4. Simulating a classical suffix array to binary search and locate the occurrences. Each file is of size 100MB.

A plugin for self-indexes. Section 4.2 considers using our reduced suffix array as a plugin to provide fast locate on existing self-indexes. In this experiment we plug our structure to the counting structures of the alphabet-friendly FM-index (AFI [Ferragina et al. 2007]), and compare the result against the original AFI, Sadakane's CSA [Sadakane 2003] and the SSA [Ferragina et al. 2007; Mäkinen and Navarro 2005], all from *PizzaChili*. We increased the sampling rate of the locating structures of AFI, CSA and SSA, to match the size of our LCSA.

Fig. 5 shows the results. We only show four texts, as the others yield similar conclusions. The experiment consists in choosing random ranges of the suffix array and obtaining the values. This simulates a locating query where we can control

the amount of occurrences to locate. Our reduced suffix array has a constant time overhead (which is related to column 6 in Table III and the sample rate of absolute values) and from then on the cost per cell located is very low. As a consequence, it crosses sooner or later all the other indexes. For example, it becomes the fastest on `xml` after locating 2 occurrences, and after 8 occurrences it becomes the fastest on `proteins`. Particularly on `xml`, this success owes to the fact that our LCSA uses the RPΨSP variant (cutting phrases at length 256 as explained). If instead we used RPΨ to gain a little further compression, the result would be very inefficient due to the long phrases that need to be uncompressed. In the case of `xml`, RPΨ becomes the fastest only after locating 3,800 occurrences, not 2. In other texts where compression is not so good, there is not much difference between RPΨ and RPΨSP (both in time and space).

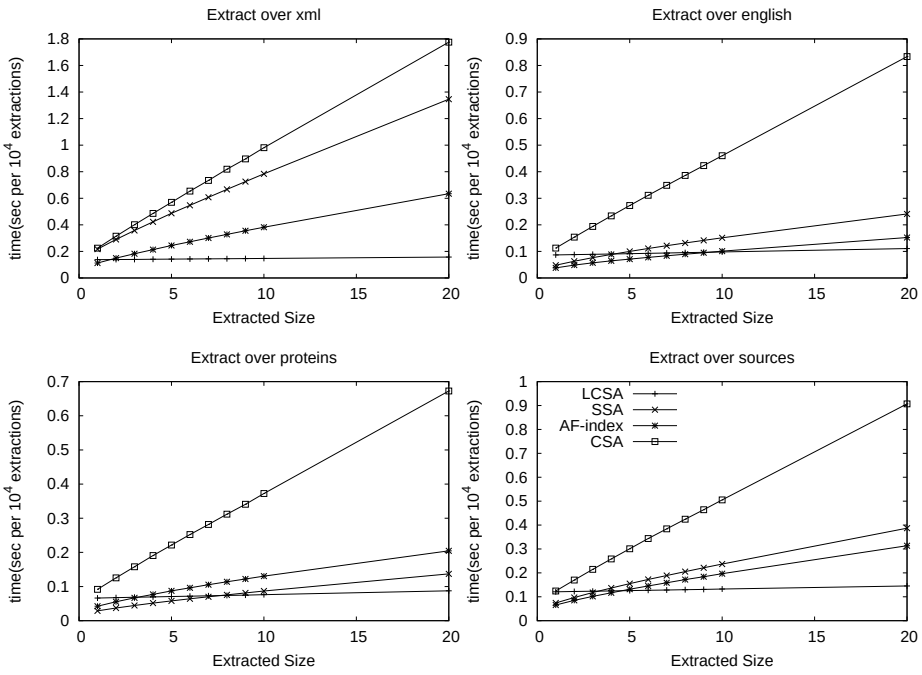

Fig. 5. Time to locate occurrences, as a function of the number of occurrences to locate. Each file is of size 100MB.

6. LCSA CONSTRUCTION IN SECONDARY MEMORY

Particularly for the application described in Section 4.3, where the LCSA does not fit in main memory, a natural question is whether is it possible to efficiently build it in secondary memory. Secondary memory algorithms to build the suffix array A are well-known [Kärkkäinen and Rao 2003; Crauser and Ferragina 2002; Dementiev et al. 2005], yet the algorithms we have presented in Section 2 for compressing A' are highly non-local. We now show that the algorithms to compress A' by using

Ψ can be adapted efficiently to secondary memory, and also how to compress the dictionary in secondary memory.

6.1 Compressing the Differential Suffix Array

When we compress in main memory by using Ψ (Sections 2.2 and 2.3), we notice that Ψ traverses the suffix array in increasing values of $A[\cdot]$. That is, if j is the position where $A[j] = 1$, then $A[\Psi[j]] = 2$, $A[\Psi[\Psi[j]]] = 3$ and so on. The idea is to store for each position i of A' the information that permits us to compress $A'[i]$ after we sort it by increasing values of A, that is, by text order. For each position in A' we define:

—$A'[i] = A[i] - A[i-1], 1 < i \leq n; A'[1] = A[1]$. This is the differential form of A.
—$A''[i] = A'[i+1], 1 < i < n$; $A''[n] = \perp$ denotes an invalid value.
—$V'[i] = A[i], 1 \leq i \leq n$. We will use this array to sort the rest.
—$NV'[i] = A[i+1], 1 \leq i < n$; $NV'[n] = \perp$. This is the position of the next valid symbol of $A'[i]$ after sorting.
—$N^2V'[i] = A[i+2], 1 \leq i < n-1$; $NV'[n-1] = NV'[n] = \perp$. This is the position of the next-next valid symbol of $A'[i]$ after sorting.
—$PV'[i] = A[i-1], 1 < i \leq n$; $PV'[1] = \perp$, the position of the previous valid symbol of $A'[i]$ after sorting.

Now we sort $\{A'[i], A''[i], V'[i], NV'[i], N^2V'[i], PV'[i]\}_{1 \leq i \leq n}$ by the values of $V' = A$. Given that A is a permutation of $\{1, \ldots, n\}$, the effect of the sorting is that of composing the arrays with A^{-1}. We call the reordered arrays as follows:

—$\tilde{A}'[j] = A'[A^{-1}[j]] = A[A^{-1}[j]] - A[A^{-1}[j]-1], 1 \leq j \leq n, j \neq A[1]; \tilde{A}'[A[1]] = A[1]$.
—$\tilde{A}''[j] = A''[A^{-1}[j]] = A[A^{-1}[j]+1] - A[A^{-1}[j]], 1 \leq j \leq n, j \neq A[n]; \tilde{A}''[A[n]] = \perp$.
—$V[j] = V'[A^{-1}[j]] = A[A^{-1}[j]] = j, 1 \leq j \leq n$.
—$NV[j] = NV'[A^{-1}[j]] = A[A^{-1}[j]+1], 1 \leq j \leq n, j \neq A[n]; NV[A[n]] = \perp$.
—$N^2V[j] = N^2V'[A^{-1}[j]] = A[A^{-1}[j]+2], 1 \leq j \leq n, j \neq A[n], j \neq A[n-1]; NV[A[n-1]] = NV[A[n]] = \perp$.
—$PV[j] = PV'[A^{-1}[j]] = A[A^{-1}[j]-1], 1 \leq j \leq n, j \neq A[1]; PV[A[1]] = \perp$.

Now that we have the arrays $\{V[j], \tilde{A}'[j], \tilde{A}''[j], NV[j], N^2V[j], PV[j]\}_{1 \leq j \leq n}$, the sequential traversal of these arrays is equivalent to navigating the original ones using Ψ. More precisely, let $j = A[i]$ (and thus $i = A^{-1}[j]$), then $\tilde{A}'[j] = A'[i]$ and $\tilde{A}''[j] = A''[i] = A'[i+1]$. Moreover, $\tilde{A}'[j+1] = A'[\Psi(i)]$ and $\tilde{A}''[j+1] = A'[\Psi(i)+1]$. Hence the check for pair equality between $A'[i]A'[i+1]$ and $A'[\Psi(i)]A'[\Psi(i)+1]$ reduces to checking whether $\tilde{A}'[j]\tilde{A}''[j] = \tilde{A}'[j+1]\tilde{A}''[j+1]$, which can be carried out sequentially on $\tilde{A}'$ and $\tilde{A}''$.

The other arrays are used to maintain consistency upon changes in A': When we change $\tilde{A}'[j]$ and $\tilde{A}''[j]$, corresponding to $A'[i]$ and $A'[i+1]$, we have the problem that the pair $A'[i-1]A'[i]$, which is explicitly stored at $\tilde{A}'[PV[j]]$ and $\tilde{A}''[PV[j]]$, must be updated as well. Similarly, NV serves to locate the place where the pair corresponding to $A'[i+1]A'[i+2]$ is stored after the sorting. Thus, as the arrays

are now sorted by $j = A[i]$, arrays PV and NV serve as a doubly-linked list to let us move to $i-1$ and $i+1$ in the sorted array. Those lists must be updated upon removals across the compression process, and they are also useful to maintain the current values of $\tilde{A}''[j]$ up to date when elements in the chain are removed.

Let τ be the total size in integers of these arrays. We divide them into l chunks of size τ/l and, for each chunk, we keep in main memory a buffer of $\tilde{b}$ integers. Let M be the size in integers of the main memory, then $\tau/l + l \cdot \tilde{b} \leq M$ must hold (we consider later the case where M is smaller). The algorithm to carry out a single pass on A' in secondary memory is as follows. Note that this is the main subroutine of both approximate methods based on Ψ (Sections 2.2 and 2.3).

(1) We read the first chunk from disk and initialize empty buffers.

(2) We find sequentially in the chunk the first j satisfying $\tilde{A}'[j]\tilde{A}''[j] = \tilde{A}'[j+1]\tilde{A}''[j+1]$. If no such j is found we go on with the next chunk. In the stronger approximate method we require equality between several $\tilde{A}'[j+r]\tilde{A}''[j+r]$ pairs before proceeding to the next step.

(3) From that j, we start a chain of replacements: We add a new pair $s \leftarrow \tilde{A}'[j]\tilde{A}''[j]$ to R, make the replacements at j and $j+1$ and move on with $j \leftarrow j+1$, replacing until the pair changes. When the pair changes, that is $\tilde{A}'[j]\tilde{A}''[j] \neq \tilde{A}'[j+1]\tilde{A}''[j+1]$, we restart the search for pairs at Step (2).

(4) If we reach the end of the block, the replacement chain may continue at the next one.

To consistently perform a replacement $\tilde{A}'[j] : \tilde{A}''[j] \leftarrow s : \perp$ in Step (3), maintaining also the linked lists, we must carry out the following actions (in parallel; we overline variables to indicate that we use their original values prior to any assignment): (a) $\tilde{A}'[j] \leftarrow s$, (b) $\tilde{A}'[\overline{NV}[j]] \leftarrow \perp$, (c) $\tilde{A}''[j] \leftarrow \tilde{A}''[\overline{NV}[j]]$, (d) $NV[j] \leftarrow \overline{N^2V}[j]$, (e) $PV[\overline{N^2V}[j]] \leftarrow j$, (f) $N^2V[j] \leftarrow N^2V[\overline{NV}[j]]$, (g) $A''[PV[j]] \leftarrow s$, and (h) $N^2V[PV[j]] \leftarrow \overline{N^2V}[j]$. From those, only (a) and (d) can be executed locally, whereas the others may require reading/writing data from/to other chunks not yet in main memory. If this is the case, we "send messages" to read/update other chunks. Those will be stored in their corresponding buffers, and carried out right before those chunks are processed. (If a buffer gets full it is written out to disk into a log of actions the chunk must execute.) Some of those messages will then send messages back to the current chunk to update its values, and this update will be executed when the current chunk is read again. This is not a problem because we will not access cell j again until the next pass. (The updates that happen to belong to the current chunk, instead, must be executed immediately.)

Each message is of the form $action(dest, parameters)$, where $dest$ is the destination position that determines the chunk $\lceil dest/l \rceil$ that will execute it; see Table IV for the meaning of each action. The global instructions we described are then translated into the following instructions and messages sent: (a) $\tilde{A}'[j] \leftarrow s$, (b) send $DL(NV[j], j)$ (will solve (c) and (f) in the next pass), (d) $NV[j] \leftarrow N^2V[j]$, (e) send $UP(N^2V[j], j)$, (g) send $UA''(PV[j], s)$, and (h) send $UN^2(PV[j], N^2V[j])$. Figure 6 shows the operations that are performed after a replacement; this occurs in three steps. Thus, at the end of the pass, we must carry out two extra passes in order to process the messages sent and their responses, before finishing the pass

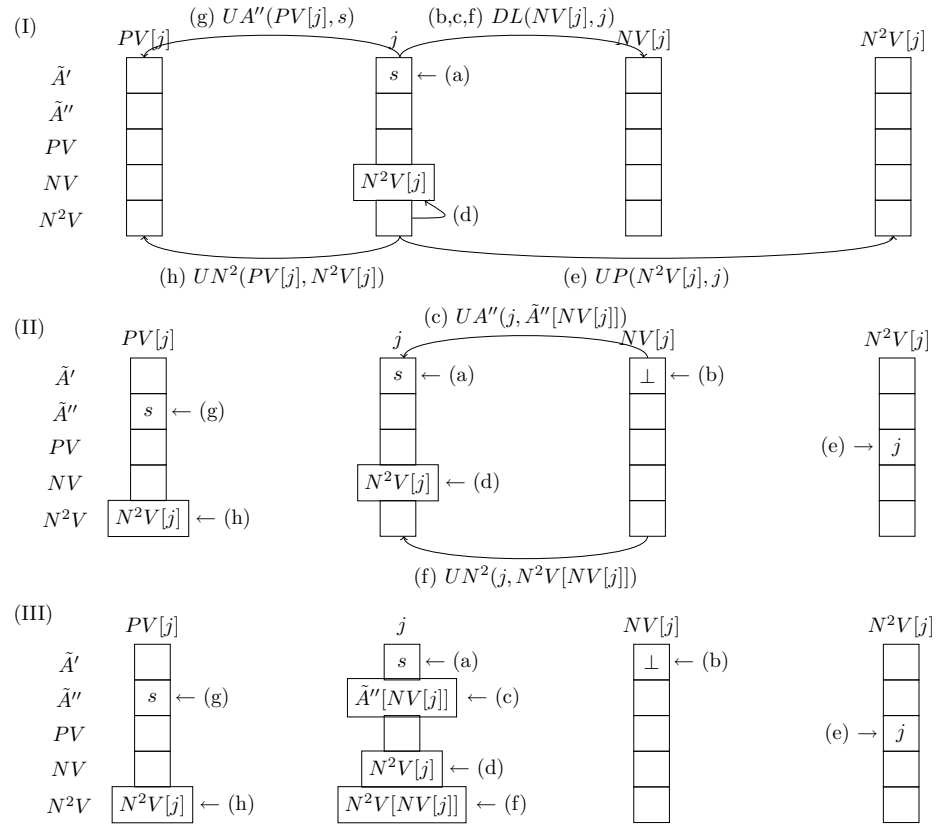

Fig. 6. Operations generated by a replacement. (I) shows the replacement and the messages generated by it. (II) shows the effect of the messages, one of which generates two more messages. (III) shows the final state product of a replacement.

properly.

action	parameter	what it does at position *dest*
UA'	sym	Updates A' to sym, $\tilde{A}'[dest] \leftarrow sym$.
UA''	sym	Updates $\tilde{A}''$ to sym, $\tilde{A}''[dest] \leftarrow sym$.
UN^2	$next$	Updates N^2V to $next$, $N^2V[dest] \leftarrow next$.
DL	$from$	Marks $dest$ as deleted and responds with $UA''(from, \tilde{A}''[dest])$ and $UN^2(from, N^2V[dest])$
UP	$prev$	Updates PV to $prev$, $PV[dest] \leftarrow prev$.

Table IV. Message types and meanings used by the secondary memory construction algorithm.

The invalid entries we produce must be compacted for the next passes. Apart from removing the invalid entries, we must update the pointers NV, N^2V, and PV. We first sort all the arrays by the NV values. The result will be an increasing sequence of $NV[i]$ values, with some missing integers due to the removed entries. Thus we assign $NV[i] \leftarrow i$ to effectively remove the invalid entries. We repeat the

process of sorting and reassigning for N^2V and PV. Finally we sort again by V, and are ready for the next pass.

Thus, each pass of the original algorithm over an array of size n' costs us, in I/O terms, $O(n'/\tilde{b})$ for the traversal plus $O((n'/\tilde{b}) \log_{M/\tilde{b}}(n'/M)) = O(Sort(n'))$ for the sortings. It is easy to see that, because the n's are of the form $(1-\alpha)^i n$, the linear-time algorithm of Section 2.2 costs $O(Sort(n))$ in secondary memory. Similarly, the $O(n \log n)$ time algorithm of Section 2.3 costs $O(Sort(n) \log n)$ time. This is almost I/O optimal with respect to the original algorithms.

As a practical note, Figure 3 shows that indeed 8 passes are sufficient, even on `xml`, to achieve most of the compression on the variants we tried.

With respect to the extra secondary memory needed, it is $O(n)$ words for the arrays, plus the message logs. Since each position can receive $O(1)$ messages from elsewhere, all the message logs add up to $O(n)$ words as well.

Finally, we consider the case $\tau/l + l \cdot \tilde{b} > M$, that is, there is no space to store one disk page per chunk in main memory. In this case we replace the in-memory buffers by a secondary-memory priority queue, where the messages are inserted with priority given by *dest*. Those sent from a chunk i to a position $dest < i$ will be inserted with priority $n + dest$, so that they are processed in a second traversal. Each new chunk reads from the priority queue (by means of extracting minima) all the messages that correspond to it, and inserts new messages. As there are overall $O(n)$ messages, optimal-I/O priority queues (e.g. [Brengel et al. 2000]) yield $O((n'/\tilde{b}) \log_{M/\tilde{b}}(n'/\tilde{b}))$ overall time, which raises only slightly the $O(Sort(n'))$ time per pass we had.

6.2 Compressing the Dictionary

When we compress the dictionary in main memory, we start expanding the first (i.e. earliest) rule s_j that has not been used by another one, $U[j] = 0$ (see Section 2.4). Then we expand the next rule $s_{j'}$ that has $U[j'] = 0$, and so on. The idea here is to group together all the rules that are used in the same expansion of a rule with value $U[\cdot] = 0$, so that when we expand it we will have almost all the information needed to do the expansion in main memory.

We can regard the dictionary as $R = \{s_1 \to a_1 b_1, s_2 \to a_2 b_2, \ldots, s_\nu \to a_\nu b_\nu\}$, where $s_i = i + n$. For each rule we define:

—$R_0[i] = s_i$, the value of the i-th rule.

—$R_1[i] = a_i$, the left symbol of the i-th rule. It can be a rule or an original symbol.

—$R_2[i] = b_i$, the right symbol of the i-th rule. It can be a rule or an original symbol.

—$Q[i] = \begin{cases} \min\{s_j, s_i \text{ is used by } s_j \wedge U[j] = 0\} \text{ if } U[i] = 1, \\ s_i \text{ otherwise.} \end{cases}$
The min is the lowest (i.e. earliest) rule that contains s_i and has value $U[\cdot] = 0$. Rules with the same value of Q will be used in the same expansion, so Q will be a kind of group identifier.

Given the non-locality of reference between rules, to calculate the values of Q we need to send messages of the form $UQ(dest, parameter)$, to achieve $Q[dest] \leftarrow$

parameter, with $1 \leq dest, parameter \leq \nu$. Note that a position *dest* could receive multiple messages and that always $dest < parameter$.

We maintain a secondary-memory priority queue PQ, where the messages are inserted with priority given by *dest* (larger first). We process the arrays from higher values of R_0 first, i.e., in reverse order. To process cell i we extract from PQ all the messages for $dest = i$ and set $Q[i]$ to the minimum *parameter* for that i. If there are no messages for i, we assign a new $Q[i] \leftarrow s_i$ (i.e., we do not need to store U). Then we insert message $UQ(R_1[i], Q[j])$ ($UQ(R_2[i], Q[j])$) into PQ, if $R_1[i]$ ($R_2[i]$) is not an original symbol. Since any rule $R_0[j]$ that uses rule $R_0[i]$ has been visited before we reach position i, necessarily the message $UQ(R_0[i], Q[j])$ is in PQ by that time. Thus we compute array Q in one pass.

Now we sort these arrays by increasing values of the pair $(Q[i], R_0[i])$, i.e., by Q and using R_0 to break ties, obtaining $\tilde{Q}, \tilde{R}_0, \tilde{R}_1, \tilde{R}_2$. These arrays can be partitioned into η groups, each one with the same value of Q. Let η_k be the position where the k-th group finishes. Note that if the length of the longest phrase is μ then any group has at most μ elements. Now, for each group, we compress the dictionary almost the same way as in algorithm Expand_Rule (see Figure 2, page 10). There are four differences with the original algorithm:

(1) We write down R_B and R_S to disk instead of maintaining them in main memory. The array NV is not used.
(2) We expand $\tilde{R}_0[\eta_k]$, the last element of the k-th group, because by construction these rules use all the other rules in the same group.
(3) To process rule $\tilde{R}_0[i]$, which is expanded to $\tilde{R}_1[i]\tilde{R}_2[i]$, we do as follows. If $\tilde{R}_1[i]$ is not an original symbol, and it does not belong to the same group of $\tilde{R}_0[i]$, then it must have been defined in a previous group. Hence we must not expand it further, but instead write its final value in R_S. Yet this value is only know within the other group. We send message $NR_S(pos, \tilde{R}_1[i])$, where *pos* is the current position where we are writing in R_S. The same goes for $\tilde{R}_2[i]$.
(4) We also write down to disk the pair $(s_j, LR_B[s_j])$ (second line of algorithm Expand_Rule), where s_j is the rule and $LR_B[s_j]$ is its final value (i.e., the current value of variable LR_B).

After carrying out the previous steps, we still need to execute the messages NR_S to update R_S. We sort the messages by *dest* (their first component) obtaining $\overline{NR}_S = (\overline{dest}, \overline{parameter})$, and sort the pairs $(s_j, LR_B[s_j])$ by its first coordinate, obtaining $(\overline{s}_j, \overline{LR}_B)$. Because the compressed dictionary will hold at least ν integers in RAM, we can use that RAM space across the process. From now on $\overline{LR}_B$ will reside completely in main memory, so we can access it at random. To apply the messages we traverse $\overline{NR}_S$ and R_S making the necessary replacements in R_S, that is, $R_S[\overline{dest}[i]] = \overline{LR}_B[\overline{parameter}[i] - n]$. These replacements are done by increasing values of $\overline{dest}$, so we traverse only once the arrays $\overline{NR}_S$ and R_S. Using again $\overline{LR}_B$ we traverse C, changing the values of the rules to their final ones.

Let M be the size in integers of the main memory. The breakdown of the cost is as follows:

—When we calculate Q there are overall $O(\nu)$ messages. Using optimal-I/O priority queues (e.g. [Brengel et al. 2000]) yields $O((\nu/\tilde{b}) \log_{M/\tilde{b}}(\nu/\tilde{b}))$ time.

—When we expand a rule we need to find both children. Each such search within their group takes at most $\log \mu$ CPU time. Overall there are at most 2ν of these searches, totalizing $O(\nu \log \mu)$ CPU time.

—We take $O(\nu/\tilde{b})$ I/Os to read/write all the needed arrays from disk.

—There are three sortings, which in total take $O(Sort(\nu))$ time.

—Updating C to the new rule values takes $O(n'/\tilde{b})$ I/Os .

Overall, time is $O((\nu/\tilde{b}) \log_{M/\tilde{b}}(\nu/\tilde{b}) + n'/\tilde{b})$ I/Os. The extra space on disk is $O(\nu)$ integers. Note that $4\mu \leq M$ must hold to be able to compress a group in main memory.

If $\overline{LR}_B$ does not fit in main memory, we pre-process the messages first, that is, we first sort NR_S by *parameter*, then we traverse it in synchronization with LR_B updating $parameter \leftarrow LR_B(parameter)$, and then we sort again NR_S by *dest*. Now we only need to traverse R_S and NR_S together, to update R_S. This add an extra $O(Sort(\nu))$ time to the cost, which does not affect our complexity. Something similar can be done to update C, which incurs an extra cost of $O(Sort(n'))$. This would affect the complexity, but is still within the cost paid in Section 6.1 to build C.

7. CONCLUSIONS AND FUTURE WORK

We have presented a suffix array compression method that retains fast locating of the occurrences of a pattern. We have proved analytically that the resulting size is related to the k-th order entropy of the text. The method has been used to obtain a compressed self-index with fast locate (where the norm is to be extremely slow), a small index that is a viable alternative to classical suffix arrays, and a secondary-memory version that works optimally and whose access time improves due to compression (where worsening is the norm). Our experiments show that the structure is very practical and relevant.

As a byproduct, we have presented Re-Pair algorithms tailored to suffix array differences, which exploit the structure of Ψ to run much faster and using much less memory than the general algorithm. Those new algorithms are approximations, yet we show that their compression loss is negligible. We also presented a secondary memory construction for those approximations, which run almost I/O-optimally and extends the applicability of the methods to compress suffix arrays that do not fit in main memory.

Another byproduct, which might be of general interest, is a compact data structure to represent the Re-Pair dictionary. This structure can reduce the dictionary space by up to 50%, and operates in compressed form, that is, it permits decompressing parts of the text without uncompressing the dictionary.

Our work leaves several future development lines. In the short term, we seek to improve the performance of the smaller classical index via algorithm engineering, and implement the secondary memory index, which is right now a theoretical proposal. In this latter line, we are working on merging the CPT structure presented by Clark and Munro [1996] with our structure. We believe the result would be extremely competitive in practice.

In the longer term, we believe this is a first step towards compressed text indexes with competitive locating times, in particular via locality of access. The key was to

build on the runs in Ψ, which have been used in the past to achieve compression yet not locality of access. We showed that the regularities that Re-Pair exploits on the differential suffix array are closely related to those runs in Ψ. Thus we could take advantage of the locality properties of Re-Pair, and also used the close relation with Ψ to analyze the compression achieved and design faster Re-Pair variants for this case. Our resulting index is still far from achieving the space used by the smallest self-indexes, which are however extremely slow to locate. Is there a fundamental lower bound to the tradeoff one can achieve between space and time for locating? Is there a limit to what can be achieved via local compression?

REFERENCES

Baeza-Yates, R., Barbosa, E. F., and Ziviani, N. 1996. Hierarchies of indices for text searching. *Information Systems 21,* 6, 497–514.

Brengel, K., Crauser, A., Ferragina, P., and Meyer, U. 2000. An experimental study of priority queues in external memory. *ACM Journal of Experimental Algorithmics 5,* 17.

Clark, D. and Munro, I. 1996. Efficient suffix trees on secondary storage. In *Proc. 7th SODA*. 383–391.

Crauser, A. and Ferragina, P. 2002. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica 32,* 1, 1–35.

Dementiev, R., Kärkkäinen, J., Mehnert, J., and Sanders, P. 2005. Better external memory suffix array construction. In *Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX)* (2007-01-30). 86–97.

Ferragina, P. and Grossi, R. 1999. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM 46,* 2, 236–280.

Ferragina, P. and Manzini, G. 2005. Indexing compressed texts. *J. of the ACM 52,* 4, 552–581.

Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG) 3,* 2, article 20.

González, R. and Navarro, G. 2006. Statistical encoding of succinct data structures. In *Proc. 17th CPM*. LNCS 4009. 295–306.

González, R. and Navarro, G. 2007. A compressed text index on secondary memory. In *Proc. 18th IWOCA*. College Publications, UK, 80–91.

Grossi, R., Gupta, A., and Vitter, J. 2003. High-order entropy-compressed text indexes. In *Proc. 14th SODA*. 841–850.

Grossi, R. and Vitter, J. 2006. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

Jacobson, G. 1989. Space-efficient static trees and graphs. In *Proc. 30th FOCS*. 549–554.

Kärkkäinen, J. and Rao, S. 2003. *Algorithms for Memory Hierarchies*. LNCS v. 2625. Springer, Chapter 7: Full-text indexes in external memory, 149–170.

Larsson, J. and Moffat, A. 2000. Off-line dictionary-based compression. *Proc. IEEE 88,* 11, 1722–1732.

Mäkinen, V. 2003. Compact suffix array — a space-efficient full-text index. *Fundamenta Informaticae 56,* 1–2, 191–210.

Mäkinen, V. and Navarro, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing 12,* 1, 40–66.

Mäkinen, V., Navarro, G., and Sadakane, K. 2004. Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In *Proc. 15th ISAAC*. LNCS v. 3341. 681–692.

Manber, U. and Myers, G. 1993. Suffix arrays: a new method for on-line string searches. *SIAM J. Computing 22,* 5, 935–948.

Manzini, G. 2001. An analysis of the Burrows-Wheeler transform. *J. of the ACM 48,* 3, 407–430.

Navarro, G. 2004. Indexing text using the Ziv-Lempel trie. *J. of Discrete Algorithms 2,* 1, 87–114.

Navarro, G. and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, article 2.

Sadakane, K. 2003. New text indexing functionalities of the compressed suffix arrays. *J. of Algorithms 48,* 2, 294–313.

Weiner, P. 1973. Linear pattern matching algorithm. In *Proc. 14th IEEE Symp. on Switching and Automata Theory.* 1–11.

Received Month Year; revised Month Year; accepted Month Year.