Compressed Text Indexes with Fast Locate

Rodrigo González[⋆] and Gonzalo Navarro[⋆⋆]

Dept. of Computer Science, University of Chile
{rgonzale,gnavarro}@dcc.uchile.cl

Abstract. 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 compression scheme for suffix arrays which permits locating the occurrences extremely fast, while still being much smaller than classical indexes. In addition, our index permits a very efficient secondary memory implementation, where compression permits reducing the amount of I/O needed to answer queries.

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* [13,1,4,14,12,2]. 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 supporting 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 [15] and suffix arrays [10], 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 $k \le \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$, where $H_k \le \log \sigma$ is the k-th order empirical entropy of T [11,13][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

[⋆] Work supported by Mecesup Grant UCH 0109, Chile.

[⋆⋆] This work was funded by a grant from Yahoo! Research Latin America.

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

is to *locate* the occurrences, that is, to 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 number 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 [12], 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 [13]. This regularity has been exploited in several ways in the past [7,14,8], 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 [6], a compression technique that builds a dictionary of phrases and permits fast local decompression using only the dictionary (whose size one can control at will, at the expense of losing some compression). We then introduce some novel techniques to further compress the Re-Pair dictionary, which can be of independent interest. We also use specific properties of suffix arrays to obtain a much faster compression losing only 1%–14% of compression.

As a result, for several text types, we reduce the suffix array to 20–70% of its original size, depending on its compressibility. This reduced index can still extract any portion of the suffix array very fast by adding a small set of sampled absolute values. We prove that the size of the result is $O(H_k \log(1/H_k) n \log n)$ bits 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 [2] for counting, our index needs $O(H_k \log(1/H_k) n \log n + n)$ bits of space, counts in time $O(m(1+\frac{\log \sigma}{\log \log n}))$ and locates the *occ* occurrences of P in time $O(occ + \log n)$.

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 can be done over the samples first and then 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.

On the ther 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 upgrade this structure to an efficient secondary-memory self-index.

We experimentally explore the compression performance we achieve, the time for locating, and the simplified suffix array implementation, comparing against previous work. Our structure stands out as an excellent practical alternative.

2 Compressing the Suffix Array

Given a text $T = t_1 \ldots t_n$ 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 $P = p_1 \ldots p_m$ 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 $O(m + \log n)$ [10].

Suffix arrays turn out to be compressible whenever T is. The k-th order empirical entropy of T, H_k [11], 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 [8,13].

This property has been used several times in the past to compress A. Mäkinen's Compact Suffix Array (CSA) [7] replaces runs with pointers to their definition elsewhere in A, so that the run can be recovered by (recursively) expanding the definition and shifting the values. Mäkinen and Navarro [8] 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 [14].

We present a completely different method to compress A. We first represent A in differential form: $A'[1] = A[1]$ and $A'[i] = A[i] - A[i-1]$ if $i > 1$. Take now a run of A of the form $A[j+s] = A[i+s] + 1$ for $0 \leq s \leq \ell$. It is easy to see that $A'[j+s] = A'[i+s]$ for $1 \leq s \leq \ell$. We have converted the runs of A into true repetitions in A'.

The next step is to take advantage of those repetitions in a way that permits fast local decompression of A'. We resort to Re-Pair [6], 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 $s \geq n$ larger than all existing symbols in A' and add rule $s \rightarrow ab$ to a dictionary; (3) replace every occurrence of ab in A by s[2]; (4) iterate until every pair has frequency 1. The result of the compression is the table of rules (call it R) plus the sequence of (original and new) symbols into which A' has been compressed (call it C). Note that R can be easily stored as a vector of pairs, so that rule $s \rightarrow ab$ is represented by $R[s-n+1] = a : b$.

Any portion of C can be easily decompressed in optimal time and fast in practice. To decompress $C[i]$, we first check if $C[i] < n$. If it is, then it is an original symbol of A' and we are done. Otherwise, we obtain both symbols from $R[C[i] - n + 1]$, 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 can 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.

A few more structures are necessary to recover the values of A: (1) a sampling of absolute values of A at regular intervals l; (2) a bitmap $L[1, n]$ marking the positions where each symbol of C (which could represent several symbols of A') starts in A'; (3) $o(n)$ further bits to answer *rank* queries on L in constant time [5,13]: $rank(L, i)$ is the number of 1's in $L[1, i]$. Thus, to retrieve $A[i, j]$ we: (1) see 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) make sure we expand an integral number of symbols in C, extending i to the left and j to the right until $L[i] = 1$ and $L[j+1] = 1$; (3) use the mechanism described above to obtain $A'[i, j]$ by expanding $C[rank(L, i), rank(L, j)]$; (4) use any absolute sample of A included in $[i, j]$ to obtain, using the differences in $A'[i, j]$, the values $A[i, j]$; (5) return the values in the original interval $[i, j]$ requested.

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 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.

2.1 Faster Compression

A weak point in our scheme is compression speed. Re-Pair can be implemented in $O(n)$ time, but needs too much space [6]. We have used instead an $O(n \log n)$ time algorithm that requires less memory. We omit the details for lack of space.

We note 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 only slightly less than 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 $0 \le s \le \ell$ (and thus $A'[j+s] = A'[i+s]$ for $1 \le s \le \ell$), then $\Psi(i+s) = j+s$ for $0 \le s \le \ell$. Thus, by following permutation Ψ we have a good chance of finding repeated pairs in A' (although, as explained, Re-Pair does a slightly better job).

The algorithm is thus as follows. Let $i_1 = A^{-1}[1]$. We start at $i = i_1$ and see if $A'[i]A'[i+1] = A'[\Psi(i)]A'[\Psi(i)+1]$. If this does not hold, we move on to $i \leftarrow \Psi(i)$ and iterate. If the equality holds, we start a chain of replacements: We add a new pair $A'[i]A'[i+1]$ to R, make the replacements at i and $\Psi(i)$ and move on with $i \leftarrow \Psi(i)$, replacing until the pair changes. When the pair changes, that is $A'[i]A'[i+1] \neq A'[\Psi(i)]A'[\Psi(i)+1]$, we restart the process with $i \leftarrow \Psi(i)$, looking again for a new pair to create. When we traverse the whole A' without finding any pair to replace, we are done. With some care (omitted for lack of space) this algorithm runs in $O(n)$ time.

2.2 Analysis

We analyze the compression ratio of our data structure. Let N be the number of runs in Ψ. As shown in [8,13], $N \le H_k n + \sigma^k$ for any $k \ge 0$. 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 now. 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', and thus 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 $n - \beta n$ 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 remove at least $\beta(n - \beta n) = \beta n(1-\beta)$ cells. The third replacement removes at least $\beta(n - \beta n - \beta n(1-\beta)) = \beta n(1-\beta)^2$ cells. It is easy to see by induction that the i-th iteration removes $\beta n(1-\beta)^{i-1}$ cells.

After M iterations we have removed $\sum_{i=1}^{M} \beta n(1-\beta)^{i-1} = n - n(1-\beta)^M$ cells, and hence the length of C is $n(1-\beta)^M$ and the length of R is $2M$. 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 $\frac{2(\ln n + \ln \ln \frac{1}{1-\beta} - \ln 2 + 1)}{\ln \frac{1}{1-\beta}}$. Since $\ln \frac{1}{1-\beta} = \ln \frac{4N}{4N-1} = \frac{1}{4N}(1 + O(\frac{1}{N}))$, 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 the total space is $O(H_k \log \frac{1}{H_k} n \log n) + o(n)$ bits, as even after the M^* replacements the numbers need $O(\log n)$ bits.

Theorem 1. *Our structure representing A' using R and C needs $O(H_k \log \frac{1}{H_k} n \log n) + o(n)$ bits, for any $k \leq \alpha \log_\sigma n$ and any constant $0 < \alpha < 1$.*

As a comparison, Mäkinen's CSA [7] needs $O(H_k n \log n)$ bits [13], 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.

We can also show that the simplified replacement method of Section 2.1 reaches the same asymptotic space complexity (proof omitted for lack of space).

2.3 Compressing the Dictionary

We now develop some techniques to reduce the dictionary of rules R without affecting C. Those can be of independent interest to improve Re-Pair in general.

A first observation is that, if we have a rule $s \rightarrow ab$ and s is only mentioned in another rule $s' \rightarrow sc$, then we could perfectly remove rule $s \rightarrow ab$ and rewrite $s' \rightarrow abc$. 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 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 $R = \{s \rightarrow ab,\ t \rightarrow sc,\ u \rightarrow ts\}$, and $C = tub$. We could first represent the rules by the bitmap R_B = `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 the bitmap positions as the identifiers of created symbols[3]). 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 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.

Let us now reduce the dictionary by expanding the definition of s within t (even when s is used elsewhere). The new bitmap is R_B = `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 = `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

[3] In practice letters are numbers up to $n-1$ and the bitmap positions are distinguished by adding them $n - 1$.

from C[4]. At the cost of storing at most $2|R|$ bits, 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 the example. We first count how many times each rule is used within R. Then we traverse R and only write down (the bits of R_B and the sequence R_S for) the entries with zero count. 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. 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.

3 Towards a Text Index

As explained in the Introduction, the reduced suffix array is not by itself a text index. We explore now different alternatives to upgrade it to full-text index.

3.1 A Main Memory Self-index

One possible choice is to add one of the many self-indexes able of counting the occurrences of P in little space [1,2,14,4]. Those indexes actually find out the area $[i, j]$ where the occurrences of P lie in A. Then locating the occurrences boils down to decompressing $A[i, j]$ from our structure.

To fix ideas, consider the alphabet-friendly FM-index [2]. It takes $nH_k + o(n \log \sigma)$ bits of space for any $k \le \alpha \log_\sigma n$ 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 $O(H_k \log(1/H_k) n \log n) + o(n)$ bits for the representation of A'. To this we must add $O((n/l) \log n)$ bits for the absolute samples, and the extra cost to limit the formation of symbols that represent very long sequences. If we limit such lengths to l as well, we have an overhead of $O((n/l) \log n)$ bits, as this can be regarded as inserting a spurious symbol every l positions in A' to prevent the formation of longer symbols. By choosing $l = \log n$ we have $O(H_k \log(1/H_k) n \log n + n)$ bits of space, and time $O(occ + \log n)$ for locating the occurrences. Other tradeoffs are possible, for example having $n \log^{1-\varepsilon} n$ bits of extra space and $O(occ + \log^\varepsilon n)$ time, for any $0 < \varepsilon < 1$.

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 the structure proposed in [3] 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 2. *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 $O(H_k \log(1/H_k) n \log n +$*

[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.

$n \log^{1-\varepsilon} n) + o(n \log \sigma)$ *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* $O((\ell + \log^{\varepsilon} n)(1 + \frac{\log \sigma}{\log \log n}))$.

3.2 A Smaller Classical Index

A simple and practical alternative is to use our reduced suffix array 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 anymore 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 data structure. 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* $O(H_k \log(1/H_k) n \log n + n)$ *bits of space in addition to* T, *for 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* $O(occ + \log n)$.

3.3 A Secondary Memory Index

In [9], an index of size $nH_0 + O(n \log \log \sigma)$ bits is described, which can identify the area of A containing the occurrences of a pattern of length m (and thus count its occurrences) using at most $2m(1 + \lceil \log_B n \rceil)$ accesses to disk, where $B \log n$ is the number of bits in a disk block. However, this index is extremely slow to locate the occurrences: each locate needs $O(\log^{\varepsilon} n)$ random accesses to disk, where in practice $\varepsilon = 1$. This is achieved by storing the inverse of Ψ [14].

If, instead, we keep only the data structures for counting, and use our reduced suffix array, we can obtain $\lceil \frac{occ}{B} \rceil$ accesses to report the *occ* occurrences, which is worst-case optimal. Assume table R is small enough to fit in main memory (recall we can always force so, losing some compression). Then, we read the corresponding area of C from disk, and uncompress each cell in memory without any further disk access (the area of C to read can be obtained from an in-memory binary search over an array storing the absolute position of the first C cell of each disk block). 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 would not perform well at all under this scenario, as the decompression process is highly non-local.

To extract text passages of length ℓ we could use compressed sequence mechanisms like [3], which easily adapt to disk and have local decompression.

4 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 plug-in for boosting the locating performance of a self-index, and the third the use of our technique as a classical index using reduced space. We use text collections obtained from the *PizzaChili* site, `http://pizzachili.dcc.uchile.cl`.

Compression performance. In Section 2.1 we mentioned that compression time of our scheme would be an issue and gave an approximate method based on Ψ which should be faster. Table 1 compares the performance of the exact Re-Pair compression algorithm (RP) and that of the Ψ-based approximation (RPΨ). We take absolute samples each 32 positions.

Table 1. Index size and build time using Re-Pair (RP) and its Ψ-based approximation (RPΨ). For the xml case, we also include a Re-Pair version (RPC) with rules up to length 256. Compression ratio compares with the $4n$ bytes needed by a suffix array.

Collection, size	Method	Index Size	Compr.	Re-Pai	Expected	Dict.	Main	Compr. with
(MB), H_3/H_0		(MB)	Ratio	Time (s)	decompr.	compr.	memory	5% in RAM
xml, 100,	RP	94.04	23.51%	25986	6939.99	57%	49%	34.29%
26.28%	RPΨ	102.76	25.69%	260	7570.49	57%	51%	81.85%
	RPC	99.82	24.96%	25129	134.99	58%	47%	35.86%
dna, 100,	RP	333.96	83.55%	11150	5.01	79%	19%	95.52%
97.02%	RPΨ	339.45	84.86%	546	4.73	78%	20%	101.4%
english, 100,	RP	221.31	55.33%	93421	238.31	59%	43%	87.98%
53.05%	RPΨ	241.33	60.33%	485	202.79	60%	44%	99.33%
pitches, 50,	RP	115.54	57.77%	15371	33.71	70%	21%	67.54%
61.37%	RPΨ	124.32	62.16%	180	26.78	67%	25%	85.36%
proteins, 100,	RP	286.66	71.67%	3143	58.97	80%	10%	79.58%
97.21%	RPΨ	295.15	73.78%	641	52.52	75%	13%	91.83%
sources, 100,	RP	151.81	37.95%	106173	2046.80	58%	48%	64.03%
40.74%	RPΨ	176.15	44.04%	377	1778.79	58%	50%	95.67%

The approximation runs 5 to 280 times faster and just loses 1%–14% in compression ratio. RP runs at 3 to 100 sec/MB, whereas RPΨ needs 0.26 to 0.65 sec/MB. Most of the indexing time is spent this compression; the rest adds up around 120 sec overall in all cases.

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 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 in the uncompressed text, otherwise DNA is almost incompressible).

Other statistics are available. 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 those cases, we can limit the maximum length of a C cell. To show how this impacts compression ratio and decompression speed, we include a so-called RPC method for xml (which has the largest C lengths). RPC forbids a rule to cross a 256-cell boundary. We can see that compression ratio is almost the same, worsening by 6.17% on xml (and less on others, not shown).

In column 7 we show the compression ratio achieved with the technique of Section 2.3, charging it the bitmap introduced as well. It can be seen that the technique is rather effective. Column 8 shows the percentage of the compressed structure (i.e., the compressed version of R) that should stay in RAM in order to be able to access C and the samples in secondary memory, as advocated in Section 3.3. Note that the percentage is not negligible when compression is good, and that 100 minus the percentage almost gives the percentage taken by C. The last column shows how much compression we would achieve if the structures that must reside on RAM were limited to 5% of the original suffix array size (this is measured before dictionary compression, so it would be around 3% after compression). 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 does a much poorer job here, as it does not choose the best pairs early.

A plugin for self-indexes. Section 3.1 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 [2]), and compare the result against the original AFI, the Sadakane's CSA [14] and the SSA [2,8], all from *PizzaChili.* We increased the sampling rate of the locating structures of AFI, CSA and SSA, to match the size of our index (RPT). To save space we exclude DNA and pitches.

Fig. 1 shows the results. 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 1 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 4,000 occurrences, but it

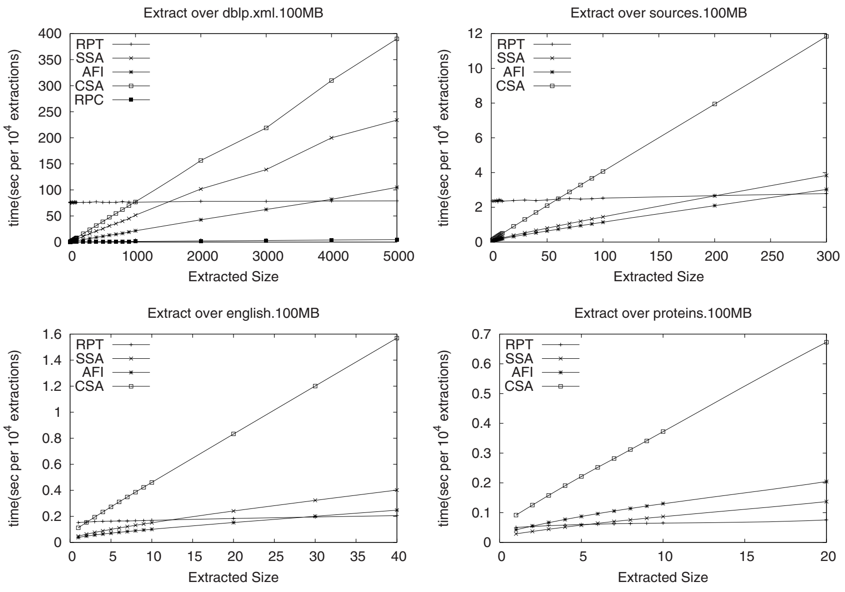

Fig. 1. Time to locate occurrences, as a function of the number of occurrences to locate. On xml, RPC becomes the fastest when extracting more than 2 results.

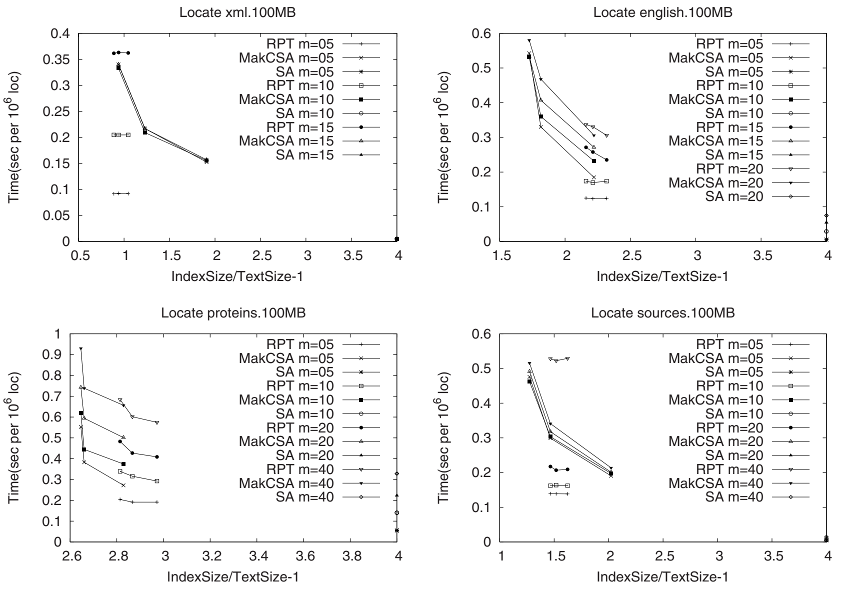

Fig. 2. Simulating a classical suffix array to binary search and locate the occurrences

needs just 6 occurrences to become the fastest on proteins. However, the RPC version shows an impressive (more than 500-fold) improvement on the cost per cell, standing as an excellent alternative when compression is so good.

A classical reduced index. Finally, we test our reduced suffix array as a replacement of the suffix array, that is, adding it the text and using it for binary searching, as explained in Section 3.2. We compare it with a plain suffix array (SA) and Mäkinen's CSA (MakCSA [7]), as the latter operates similarly.

Fig. 2 shows the result. The CSA 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 tradeoffs, especially when locating many occurrences (i.e. on short patterns).

References

1. Ferragina, P., Manzini, G.: Indexing compressed texts. J. of the ACM 52(4), 552–581 (2005)
2. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representation of sequences and full-text indexes. ACM Transactions on Algorithms, 2006. TR 2004-05, Technische Fakultät, Univ. Bielefeld, Germany (to appear)
3. González, R., Navarro, G.: Statistical encoding of succinct data structures. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 295–306. Springer, Heidelberg (2006)
4. Grossi, R., Gupta, A., Vitter, J.: High-order entropy-compressed text indexes. In: Proc. 14th SODA, pp. 841–850 (2003)
5. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th FOCS, pp. 549–554 (1989)
6. Larsson, J., Moffat, A.: Off-line dictionary-based compression. Proc. IEEE 88(11), 1722–1732 (2000)
7. Mäkinen, V.: Compact suffix array — a space-efficient full-text index. Fundamenta Informaticae 56(1–2), 191–210 (2003)
8. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic J. of Computing 12(1), 40–66 (2005)
9. Mäkinen, V., Navarro, G., Sadakane, K.: Advantages of backward searching — efficient secondary memory and distributed implementation of compressed suffix arrays. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 681–692. Springer, Heidelberg (2004)
10. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
11. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. of the ACM 48(3), 407–430 (2001)
12. Navarro, G.: Indexing text using the Ziv-Lempel trie. J. of Discrete Algorithms 2(1), 87–114 (2004)
13. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys (to appear)
14. Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. J. of Algorithms 48(2), 294–313 (2003)
15. Weiner, P.: Linear pattern matching algorithm. In: Proc. 14th IEEE Symp. on Switching and Automata Theory, pp. 1–11 (1973)