CSA++: Fast Pattern Search for Large Alphabets*

Simon Gog[†] Alistair Moffat[‡] Matthias Petri[§]

Abstract

Indexed pattern search in text has been studied for many decades. For small alphabets, the FM-Index provides unmatched performance for Count operations, in terms of both space required and search speed. For large alphabets – for example, when the tokens are words – the situation is more complex, and FM-Index representations are compact, but potentially slow. In this paper we apply recent innovations from the field of inverted indexing and document retrieval to compressed pattern search, including for alphabets into the millions. Commencing with the practical compressed suffix array structure developed by Sadakane, we show that the Elias-Fano code-based approach to document indexing can be adapted to provide new trade-off options in indexed pattern search, and offers significantly faster pattern processing compared to previous implementations, as well as reduced space requirements. We report a detailed experimental evaluation that demonstrates the relative advantages of the new approach, using the standard Pizza&Chili methodology and files, as well as applied use-cases derived from large-scale data compression, and from natural language processing. For large alphabets, the new structure gives rise to space requirements that are close to those of the most highly-compressed FM-Index variants, in conjunction with unparalleled Count throughput rates.

1 Introduction and Background

We study a well-known problem: given a static text $\mathsf{T}[0,n-2]$ over an alphabet Σ of size σ followed by a symbol $\mathsf{T}[n-1] = \$$, with $\$ \notin \Sigma$, preprocess T so that a sequence of patterns $\mathsf{P}[0,m-1]$, also over Σ, can be efficiently searched for, with the purpose of each search being to identify the number of occurrences *nocc* of P in T via a Count query. A variety of options exist for this problem, providing different trade-offs between construction cost, memory space required during pattern search operations, and search cost, both asymptotically and in practical terms. Example structures include the *suffix tree* [3, 26] and *suffix array* [16]. The suffix array of T, denoted SA, requires $O(n\log n)$ bits of space in addition to the $O(n\log\sigma)$ bits occupied by T, and uses that space to store the offsets $\mathsf{SA}[0,n-1]$ of all n suffixes of T (denoted as $\mathsf{T}[i,])$ in lexicographic order such that $\mathsf{T}[\mathsf{SA}[i],] < \mathsf{T}[\mathsf{SA}[i+1],]$ for $i \in [0,n-1]$. Using SA, the number of occurrences of P in T can be identified in $O(m\log n)$ time, via two binary searches that determine the range (sp,ep) such that all suffixes $\mathsf{SA}[sp,ep]$ are prefixed by P. Thus, $nocc = ep - sp + 1$. The search cost can be reduced to $O(m+\log n)$ if information about longest common prefixes is also available. Storing this information for all possible intervals $\mathsf{SA}[i,j]$ occurring in the binary search process requires $O(n\log n)$ bits of additional space.

Compressed Indexes. In a *compressed suffix array*, or CSA, the space required is proportional to the compressed size of T. Sadakane [23] (see also Grossi and Vitter [12]) describes a CSA based on the observation that the function $\psi[i] = \mathsf{SA}^{-1}[(\mathsf{SA}[i]+1) \bmod n]$ consists of σ increasing sequences (or *segments*) of integers, and that each of those segments is likely to be compressible, yielding a space usage of $nH_k(\mathsf{T}) + O(n\log\log\sigma)$ bits [19], where H_k denotes to the order-k empirical entropy of T. To implement Count, occurrences of P are located by performing a *backward search* to find the range $\mathsf{SA}[sp_i, ep_i]$ matching each suffix $\mathsf{P}[i,]$, stopping if $ep_i < sp_i$, or if all of P has been processed.

An alternative compressed index is due to Ferragina and Manzini [6], and is based on the Burrows Wheeler Transform (BWT), defined as $\mathsf{BWT}[i] = \mathsf{T}[\mathsf{SA}[i]-1 \bmod n]$. In an *FM-Index* the BWT is generally encoded using a *wavelet tree* [11], and accessed using $\mathsf{Rank}(\mathsf{BWT},i,c)$, which returns the number of times symbol c occurs in the prefix $\mathsf{BWT}[0,i-1]$. Again, P is processed in reverse order. Suppose $\mathsf{SA}[sp_i,ep_i]$ refers to the range in SA prefixed by $\mathsf{P}[i,]$, and that $\mathsf{P}[i-1] = c$. An array C of $\sigma\log n$ bits stores the number of symbols c in T smaller than c; using it, $sp_{i-1} = C[c] + \mathsf{Rank}(\mathsf{BWT},sp_i,c)$ and $ep_{i-1} = C[c] + \mathsf{Rank}(\mathsf{BWT},ep_i+1,c) - 1$ can be computed. Overall, $\mathsf{SA}[sp,ep]$ is identified using $2m$ Rank operations on the BWT; and when stored using a wavelet tree, $O(m\log\sigma)$ time. For more information about these structures, and the time/space trade-offs that they allow, see Navarro and Mäkinen [19] and Ferragina et al. [5].

In Practice. Implementations of the CSA and the FM-Index have been developed and measured using a range of data. When σ is small – for example, $\sigma = 4$ for DNA, and $\sigma \approx 100$ for plain ASCII text – both provide fast pattern search based on compact memory footprints, usually requiring half or less of the space initially occupied by T, depending on

*This work was supported under the Australian Research Council's Discovery Projects funding scheme (project number DP140103256).

†Institute for Theoretical Informatics, Karlsruhe Institute of Technology, Germany; and Department of Computing and Information Systems, The University of Melbourne, Australia.

‡Department of Computing and Information Systems, The University of Melbourne, Australia.

§Department of Computing and Information Systems, The University of Melbourne, Australia.

a range of secondary structures and parameter choices [9, 13], and with the FM-Index typically requiring less space that the CSA. But when σ is large – for example, when the alphabet is words in a natural language and $\sigma \approx 10^6$ or greater – the situation is more complex. In particular, the $O(\log \sigma)$ factor associated with the FM-Index's wavelet tree is a count of random accesses (as distinct from cache-friendly accesses) and means that search costs increase with alphabet size, negating its space advantage. In contrast, standard CSA implementations are relatively unaffected by σ, but each backward search step in a CSA has a dependency on $\log n_c$, where n_c is the frequency in T of the current symbol $c = \mathsf{P}[i]$. Hence, if σ is fixed and does not grow with n, CSA pattern match times will grow as T becomes longer.

Our Contribution. We introduce several improvements to the CSA index:

- We adapt and extend the uniform partitioned Elias-Fano (UEF) representation of Ottaviano and Venturini [21] to the storage of the ψ function, allowing faster backwards search compared to previous implementations;
- We add a fourth UEF block type compared to Ottaviano and Venturini, and include the option of coding sections of the ψ function in a runlength mode;
- We describe a way of segregating the short segments in ψ, allowing improved compression when σ is large and many of the symbols in Σ are rare;
- We carry out detailed "at scale" experiments, including both synthetic query streams and logs derived from use-cases, covering all of small, medium, and large alphabets, comparing a broader range of techniques and combinations for supporting Count than has any previous empirical evaluation.

The result is a pattern search index that we refer to as "CSA++". It represents a significant shift in the previous relativities between compressed index structures; and, for large alphabets in particular, gives rise to space needs close to those of the most highly-compressed FM-Index variants, with unparalleled search throughput rates.

2 Storing Integer Lists

Operations Required. The function

$$\psi[i] = \mathsf{SA}^{-1}[(\mathsf{SA}[i]+1) \bmod n]$$

is a critical – and costly – component of a CSA. It can be thought of as consisting of a concatenation of σ *segments*, the c th of which is a sorted list of the locations in BWT at which the c th symbol in Σ appear. That is, each segment of ψ can be interpreted as a *postings list* of occurrences of symbol c. The key operation required to enable backwards search is that of $\mathsf{GEQ}(c, pos)$, which returns the smallest position pos' such that $\psi[pos']$ is in the c th segment, and such that $\psi[pos'] \geq pos$. Starting with $sp = 0$ and $ep = n-1$, the (sp, ep) bounds are narrowed via a sequence of m pairs of $sp = \mathsf{GEQ}(c, sp)$ and $ep = \mathsf{GEQ}(c, ep+1) - 1$ operations, as c takes on values from $\mathsf{P}[m-1]$ through to $\mathsf{P}[0]$. The equivalence of the CSA and FM-Index search processes can be seen by noting that $\mathsf{GEQ}(c, pos) = C[c] + \mathsf{Rank}(\mathsf{BWT}, pos, c)$, and that all of the (sp, ep) pairs computed are identical between the two. Note also that, by construction, symbol occurrences in the BWT string are likely to appear in clusters, and hence ψ is likely to contain runs of consecutive or near-consecutive integers, separated by large intervals, and to contain at most σ "disruption" points at which $\psi[i] > \psi[i+1]$.

Integer Codes. One common way of storing postings lists is to compute *gaps*, or *differences*, and then store them using a suitable code for integers; clusters in BWT then gives rise to runs of small or unit gaps in ψ. A range of integer codes have been developed for this type of distribution, including Elias γ and δ codes, Rice and Golomb codes, and the Binary Interpolative Code (see Moffat and Turpin [18, Chapter 3] for descriptions). Several of these have been used in previous CSA implementations [23].

There has been recent interest in *Elias-Fano encodings* (EF encodings) for postings list compression, a result of work by Vigna [25] (see also Anh and Moffat [2] for earlier application, and Gog et al. [9] for preliminary experimentation with compressed suffix arrays). Given a non-decreasing set of k integers in the range $0 \ldots 2^U - 1$ for some universe size 2^U, a parameter ℓ is selected, and each integer is split into a high part (the most significant $U - \ell$ bits) and a low part (the ℓ low-order bits). Groups are formed for values that have the same high parts. A code for the block of k values is then constructed by representing the size of each of the $2^{U-\ell}$ possible groups in unary, followed by the concatenation in order of the k low parts. For example, if $U = 4$ and $k = 3$, the sequence $[6, 7, 10]$ (that is, $[0110, 0111, 1010]$ in binary) would be coded using $\ell = 2$, and split into high parts, $[01, 01, 10]$, coded as group sizes in unary as `0:110:10:0`; and into low parts coded in binary, `10:11:10`, where the "`:`"s are purely indicative, and do not appear in the output. The EF encoding achieves representations close to the combinatorial minimum if $\ell = \lfloor \log_2(2^U/k) \rfloor$; moreover, the length of the coded block is easily computed: $k + 2^{U-\ell}$ bits are required for the high/unary parts, and $k \cdot \ell$ bits for the low/binary parts.

One useful aspect of the EF encoding is that the unary parts can be searched via Select operations over their "0" bits, and then the number of binary parts through until that point computed. For example, in the unary sequence shown above, any elements from the underlying sequence in the range $8 \ldots 11$ must fall in the third bucket, and $\mathsf{Select}_0(2) - 2 = 4 - 2 = 2$ indicates that there are in total two binary parts contained within the first two buckets, and hence that the binary parts associated with the third bucket (if any) must commence from the third element of the low/binary part. On average there is $O(1)$ item per bucket, and linear search can be used to scan them; if a worst-case bound is required, binary search can be used if there are more than $\log_2 n$ "1" bits between the relevant pair of consecutive "0" bits, and linear search employed otherwise.

Another feature of the EF representation is that in the binary part all components are of the same bit-length ℓ, meaning that there are no dependencies that would hinder vectorized processing and loop-unrolling techniques and prevent them from achieving their full potential. This is not the case with, for example Elias δ codes, which are based upon gaps and are also of variable length, and hence must be decoded sequentially.

Partitioned Elias-Fano Codes. The term occurrences in long postings lists tend to be clustered, a pattern that has been used as the basis for a range of improved index compression techniques [18]. Ottaviano and Venturini [21] demonstrated that EF encodings could capture much of this effect if postings lists were broken into blocks of k values, and then the document identifiers in each block mapped to the range $0 \ldots 2^U - 1$ for some suitable per-block choice of U. Ottaviano and Venturini further observed that in some cases EF encodings are less efficient than other options, and that it was helpful for blocks to be coded in one of three distinct modes: (i) those consisting of an ascending run of k consecutive document identifiers, in which case no further code bits are required at all (NIL blocks); (ii) those where the document identifiers are sufficiently clustered (but not consecutive) that a 2^U-bit vector is the most economical approach (BV blocks); and (iii) those that are best represented using EF encodings, taking $2^{U-\ell} + k \cdot (1 + \ell)$ bits. Note that the decision between these options can be made based solely on k and U.

The combination of fixed-k blocks and range-based code selection is referred to as *Uniform Elias-Fano* (UEF) encoding. Ottaviano and Venturini also describe a mechanism for partitioning postings lists into approximately-homogeneous variable-length blocks in a manner that benefits EF encodings. We do not employ that additional method here.

Overall Structure of a CSA. With gaps in ψ represented by variable-length codewords, the ability to directly identify and then search segments of ψ is lost. Instead, pseudo-random access is provided via a set of *samples*: ψ is broken into fixed-length blocks; the first ψ value in each block is retained uncompressed in a *sample index*; and the remaining values in that block are coded as gaps starting from that first value [20, 23]. Computation of $\mathsf{GEQ}(c, pos)$ then involves identification of the region in the sample index associated with the segment for symbol c, binary search in that section of the sample index to identify the single block that contains pos or the next ψ value greater than it; and then sequential decoding of that whole block, to reconstruct values of ψ in order to determine the exact value. If symbol c occurs n_c times in T, and if samples are extracted every k values, then searching the sample index requires $O(\log(n_c/k))$ time, a cost that must be balanced against the $O(k)$ cost of linear search within the block. Small values of k give faster $\mathsf{GEQ}(c, pos)$ operations, but also increase the size of the sample index, and hence the size of the CSA.

3 Representing ψ

We store the ψ function of a CSA using the UEF approach of Ottaviano and Venturini, using a blocksize of k as the basis for both the UEF encoding and the sample index [9]. A number of further enhancements to previous implementations are now described.

Independent Structures. Rather than storing the whole of ψ as a single entity split into blocks, we treat each segment independently, and in doing so, explicitly form an inverted index for the symbols c in BWT. The $\sigma \log n$-bit array C of cumulative symbol frequencies is retained, and hence $n_c = C[c+1] - C[c]$. A UEF-structured postings list of $\lceil n_c/k \rceil$ blocks is then created for symbol c, with its own sample index constructed from the first (smallest) value in each of the blocks, and also represented using the EF approach, with $U' = \lceil \log_2 n \rceil$ as the universe size for this "top level" structure, and $k' = \lceil n_c/k \rceil$ the number of values to be coded within it.

One risk with this "separate structures" approach is that symbols c for which n_c is small may incur relatively high overheads; a mechanism for addressing this concern is presented shortly. Another potential issue is the cost of the mapping needed to provide access to the cth of these structures, given a symbol identifier c; that process is also described in more detail later in this section.

RL Blocks. Ottaviano and Venturini [21] employ three block types, to which we add a fourth: *run-length encoded* blocks (RL blocks). The NIL blocks of Ottaviano and Venturini account for runs of k consecutive ψ values; but there are also many instances of shorter runs that do not span a whole block. In an RL block, the (strictly positive) gaps between consecutive ψ values are represented using the Elias δ code. Any unit gaps are followed by a second δ code to indicate a repeat counter, while non-unit gaps are left as is. For example, $[27, 28, 29, 45, 46, 47, 48, 70, 71, 73]$ would be represented as $[(+1, 2), +16, (+1, 3), +22, (+1, 1), +2]$, with the plus symbols and parentheses indicative only, and with the sampled value 27 held in the top-level structure.

To decide whether to apply RL mode to any given block, the space that it would consume is found by summing the lengths of the δ codes, and comparing against the (calculated) cost of the BV and EF alternatives. Because δ is slower to decode than EF, a "relative advantage" test is applied, and blocks are coded using the RL approach only if the RL size is less than half the size of the smaller of an equivalent BV or EF block. A flag bit at the start of each block informs the decoder which mode is in use for that block.

Low-Frequency Symbols. When σ is large it is likely that many symbols in Σ have relatively low frequencies and hence notably different values in ψ; and having a small number of widely-spaced values in a block that is otherwise tightly clustered increases the cost of every codeword in the block, because of the non-adaptive nature of the EF representation. In the "separate structures" approach we are adopting, there is also a level of per-segment overhead that is relatively expensive for short segments. To address this issue, we add a further option for storing the ψ values for low-frequency symbols, and do not build an independent UEF structure for them. For example, consider a symbol c of frequency $n_c = 2$. Its segment in ψ is only two symbols long, and it is far more effective to segregate those two values into two elements of a separate array using $\lceil \log_2 n \rceil$ bits each than it is to construct a UEF structure and the associated sample index. In particular, if those two elements are within a larger array in which all of the values for all symbols for which $n_c = 2$ are stored, the overhead space can be kept small.

The array C of size σ elements has already been mentioned, it allows n_c to be computed for a symbol c. A bitvector D of size σ with Rank support is added, with $D[c] = 1$ if symbol c is being stored as a full UEF structure, and $D[c] = 0$ if $n_c \leq L$ for some threshold L. We use D to map from Σ to $\Sigma' = \{c \mid n_c \leq L\}$. The next component required is a wavelet tree over the values n_c, where $c \in \Sigma'$, to support Rank operations and hence determine how many symbols $c' < c$ in Σ' have $n_{c'} = n_c$. Finally, a set of L arrays are maintained, one for each symbol frequency between 1 and L. We suppose that A_i is the i th of those arrays. With those components available, locating the segment of ψ values corresponding to symbol c is carried out as follows. First, $D[c]$ is accessed and $n_c = C[c+1] - C[c]$ is determined. If $D[c]$ is zero, the wavelet tree is used to compute $s = |\{c' \mid 1 \leq c' < c \text{ and } n_{c'} = n_c\}|$, and the n_c required values of ψ are at $A_{n_c}[n_c \cdot s \ldots n_c \cdot s + n_c - 1]$. On the other hand, if $D[c] = 1$, then $s = \mathsf{Rank}(D, c, 1)$ is computed, and the s th of the full UEF structures is used to access the c th segment of ψ.

In the experiments reported in the next section we take $L = k$, where k is the UEF block size and also the sample interval. That is, any symbols c for which $n_c \leq k$ and less than one full UEF block would be required are stored in uncompressed form as binary values in the range $0 \ldots n-1$, in contiguous sections of shared arrays. Note that as a further small optimization the groups of n_c elements that collectively comprise each of the arrays A_{n_c} could themselves be stored using EF encodings when $n_c \geq 2$, since the EF-compressed length of each such group is both readily calculable and identical. However, given that naturally-occurring large-alphabet frequency distributions typically have long tails of very low symbol frequencies, the average cost of such EF encodings might be close to $\lceil \log_2 n \rceil$ bits per ψ value anyway, in which case we would expect the additional gains to be modest. We leave detailed exploration of this idea for future work.

Eliminating Double Search. As described earlier, each symbol that is processed in P generates two GEQ operations over ψ. It is thus easy to compute these via two calls to the same function. But much of the computation between the calls can be shared, and it is more efficient to perform the first GEQ call to identify $\mathsf{GEQ}(c, sp)$, and then perform a finger-search from that point to compute $\mathsf{GEQ}(c, ep)$. While this is a relatively simple change from a logical point of view, it is non-trivial from an engineering perspective, and not currently implemented in existing software packages.

4 Experiments

We now present results obtained for small- and large-alphabet experimentation. We also gives results for two applications that reflect the different characteristics of these two different types of alphabet.

Methodology and Test Environment. The baselines and CSA++ are written in `C++14` on top of the SDSL library [7] and compiled with optimizations using `gcc 5.2.1`. The experimental results were generated using a Intel Xeon E5640 CPU using 144 GiB RAM. All timings reported are averaged over five runs; the variance was low and all measurements lie within approximately 10% of each reported value. All space usages reported are those of the serialized data structures on disk. To ensure the reproducibility of our results, our complete experimental setup, including data files, is available at `github.com/mpetri/benchmark-suffix-array/`.

Baselines. We compare the new CSA++ approach to a wide range of highly optimized baselines. The method CSA reflects the description of [23], as implemented in the SDSL library; it stores the ψ function using Elias γ codes as a single stream of gaps, with the disruptive elements at the start of each segment represented as very large values rather than as negative gaps, and with the samples stored uncompressed. Method CSA-SADA is Sadakane's implementation of the same mechanism, available from the Pizza&Chili web site.

We also compare against two versions of each of two FM-Index approaches. The first pair, prefixed FM-HF, use a Huffman-shaped wavelet tree (WT) for the whole BWT [15]. The first version of this approach represents the WT by an uncompressed bitvector and a cache-friendly rank structure (FM-HF-BVIL), and seeks to provide fast querying at the expense of memory space; the second one uses entropy-compressed bitvectors (FM-HF-RRR) to represent the WT, and is at the other extreme of the space/speed tradeoff. The second pair of FM-Indexes are based on fixed-block compression boosting, prefixed FM-FB. The BWT is partitioned into fixed-length blocks and a WT is created for each block. We use a recent implementation by Gog et al. [8], and plug-in an uncompressed bitvector and rank structure (FM-FB-BVIL), and a hybrid bitvector (FM-FB-HYB) [14]. We did not have access to an implementation of another recent CSA proposal [1]. We additionally enhance the OptPFD [27] based CSA (CSA-OPF) of Gog et al. [9] to support Count queries. Note that 32-bit limitations in the OptPFD codes used for this restrict the CSA-OPF index to at most 4 GiB.

For large alphabets we additionally compare against an alphabet partitioned (FM-AP) index [4] which provides $O(\log\log\sigma)$ rank time, and a variant FM-AP-HYB which uses a hybrid bitvector [14]; two versions of Golynski et al.'s [10] rank structure (GMR-RS and GMR); and against a Huffman shaped WT using either a plain bitvector (FM-HF-BVIL) or a hybrid bitvector (FM-HF-HYB).

Appendix A gives the exact specification of the various methods compared.

Data Sets and Queries. Our experiments make use of texts T from two different sources: four 200 MiB files drawn from the Pizza&Chili corpus[1], selected to illustrate a range of alphabet sizes σ; plus two 2 GiB files of natural language text, one in German, and one in Spanish. The latter were extracted from a sentence-parsed prefix of the German and Spanish sections of the CommonCrawl[2]. The four 200 MiB Pizza&Chili files are treated as byte streams, with $\sigma \leq 256$ in all cases; the two larger files are parsed in to word tokens, and then those tokens mapped to integers. There were $\sigma = 5{,}039{,}965$ distinct words (integers) in the German-language file, and $\sigma = 2{,}956{,}209$ distinct words in the Spanish-language file.

The primary query streams applied to these files were generated by randomly selecting 50,000 locations in T and extracting $m = 20$-character strings for the Pizza&Chili files, and extracting $m = 4$-symbol/word strings for the two natural language files. This follows the methodology adopted by other similar experimentation carried out in the past. As secondary query streams, we also make use of the strings generated by two specific use-cases, described later in this section, in part as a response to the concerns explored by Moffat and Gog [17].

Pattern Search, Small Alphabets. Figure 1 depicts the relative performance of the CSA++, three previous CSA implementations, and a total four of FM-Index options. Index size on the horizontal axis is expressed as a percentage relative to the original text size, which in the case of these four files, is always 200 MiB. To measure search times, plotted on the vertical axis, the corresponding query streams were executed in entirety to determine an *nocc* count for each query, and then the overall execution time for the stream was divided by the total number of query characters, to obtain a computation time per query byte. Where there is more than one point shown for a method, the blocksize k is the parameter being varied. As can be seen from the four graphs in the figure, in general, the best of the FM-Indexes tested were the two FM-FB variants, and they also outperformed the two CSA implementations. The CSA++ outperforms the three earlier implementations of the CSA approach on all four files.

Pattern Search, Large Alphabets. Figure 2 shows the same experiment, applied to the large-alphabet natural-language texts. A total of six FM-Index methods suited to large alphabets are compared to the previous CSA (the SDSL version), the CSA-OPF of Gog et al. [9] extended as part of this work to support Count queries, and the new CSA++. In this environment the CSA++ dominates all of the alternative mechanisms, requiring either substantially less space, or offering greatly improved query rates. The careful attention paid to the representation of infrequent terms is clearly beneficial.

Figure 3 helps explain the situation. The great majority of the symbols in Σ occur fewer than $k = 128$ times; indeed, 25% of them appear only once. Reducing the per-term overhead is thus very important. However, as is shown in the right pane, those terms are a small percentage of the ψ array, and storing them in binary is not detrimental to overall performance.

Detailed Space Breakdown. Table 1 provide details of the space required by various components of the CSA++, for a small-alphabet file, a mid-alphabet file, and a large-alphabet file. The two columns associated with each of the three files show the space required by the named component, preceded by, where appropriate, the fraction of the values in ψ that are handled via that option. The EF-encoded samples require around 1–2% of the original space; and various other access structures, including the wavelet tree for low-n_c symbols, require a further 2–3%. The four different block types play different roles across the three files. For the DNA data, the great majority of ψ values are included in BV blocks; for the XML data, the emphasis is on NIL blocks; and for the word-based large-alphabet data it is EF blocks that dominate. In the latter case, a small but important fraction of the ψ values are coded in plain binary, as shown above in Figure 3. The effect of this alphabet partitioning is better compression for the EF-encoded values, which on this file are the dominant type; confirming that this option is an important component of the

[1] See `http://pizzachili.dcc.uchile.cl/texts.html`.

[2] See `http://data.statmt.org/ngrams/deduped/`

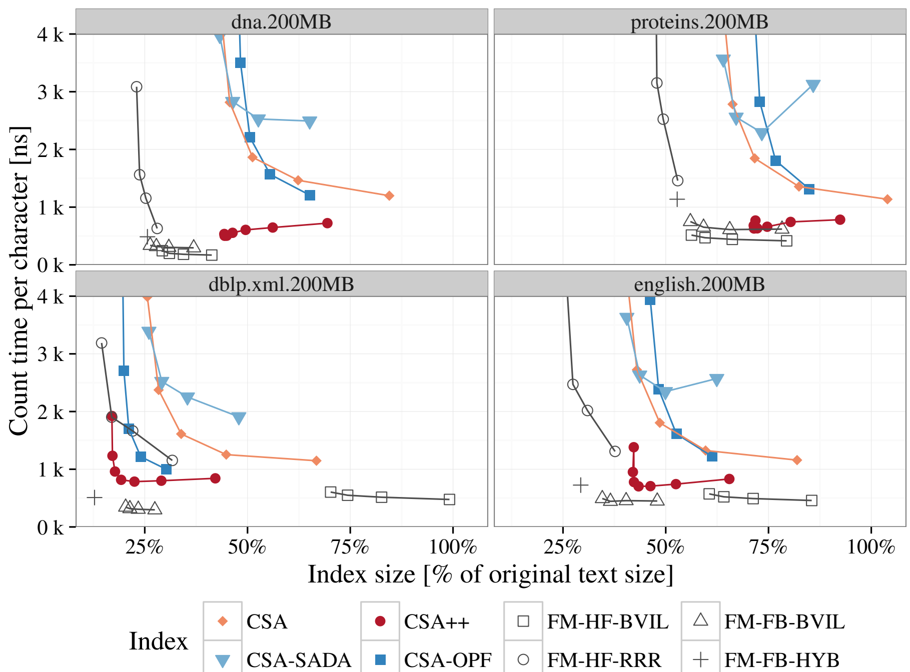

Figure 1: Cost of indexed pattern search for a set of small- and medium-alphabet Pizza&Chili files, each 200 MiB. The preferred zone is at the lower-left. The new approach is denoted as CSA++, the other methods are a broad range of baselines and are described in detail in Appendix A. **Alistair says:** *Can probably enlarge figures a bit (10%? 15%?) while keeping same aspect ratio, we have plenty of space, limit is 15 pages...*

large-alphabet situations handled so well by the CSA++. The introduction of the RL blocks results in a relatively saving on the DNA and German files, but makes a substantial relative difference for the XML file. If the RL threshold L is made vary large, so that no blocks employ that option, the three CSA structures grow to 93.2 MiB, 897.8 MiB, and 49.2 MiB respectively. Table 1 also lists the space needed by several other compressed pattern search structures, to provide further context for these results.

Case Study One: Text Factorization. The *Relative Lempel-Ziv* (RLZ) compression mechanism represents a string STR as a sequence of *factors* from a dictionary D, see Petri et al. [22] for a description and experimental results. To greedily determine longest factors using a CSA, we take $\mathsf{T} = D^{\mathsf{r}}$, the reverse of D, and build a compressed index. The string is then processed against T taking symbols from STR in left-to-right order, and performing a backward search in T; if a prefix of length p from STR is sufficient to ensure that the (sp, ep) range becomes empty, then the next factor emitted is of length $p-1$. That is, the factorization process can be regarded as applying variable-length patterns to a text T, with each pattern being as short as possible without appearing in T. To carry out an application-driven experiment, we took the 64 GiB prefix of the GOV2 document collection used by Petri et al., and built a set of patterns, each of which is one factor, plus the next character from STR. The first 1,901,131,365 patterns from that set, representing 4 GiB of text, were used as queries. The average factor length was 23.6 characters, with $nocc = 0$ in T in all cases. We then applied those patterns to a 256 MiB dictionary D constructed from the whole 64 GiB, to compute the per-character cost of performing the specified searches, and compared against the per-character cost associated with search for randomly selected patterns. Table 2 shows the cost of backward search step in both scenarios and confirms both that CSA++ significantly outperforms CSA, and also that for Count queries, random strings are a reasonable experimental methodology.

Case Study Two: Language Modeling. A common operation on natural language files is to identify informative phrases as sentences are parsed. We built variable-length queries for the file german-2048, and measured the per-symbol processing time, comparing actual-use queries and randomly-selected-string queries for CSA and CSA++. In total 1,521,869 queries of average length 3.4 words were extracted from the machine translation process described by Shareghi et al. [24], corresponding to 40,000 sentences randomly selected from the German part of Common Crawl. Table 3 shows the cost of those Count queries over the german-2048 file. The results again align with the performance of pattern searches for random queries extracted from the text, as was shown in Figure 2.

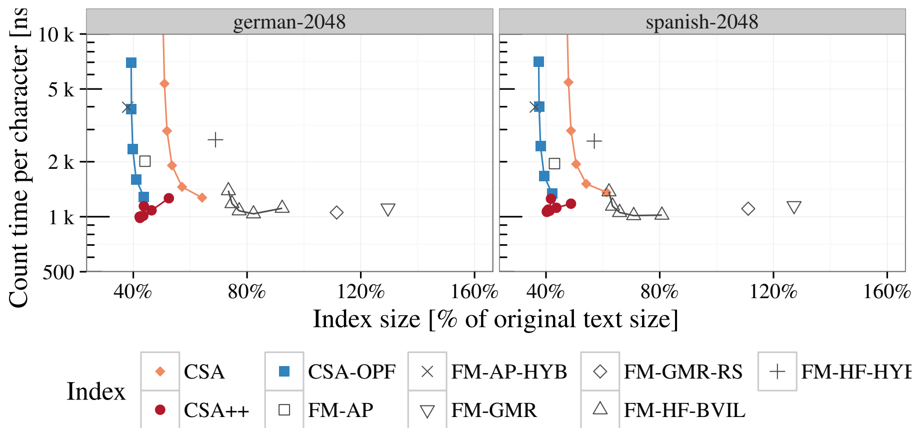

Figure 2: Cost of indexed pattern search for two 2 GiB files of natural language text, parsed into word tokens. The new approach is denoted as CSA++, the other methods are a broad range of baselines and are described in detail in Appendix A. The preferred zone is at the lower-left. Note that the vertical axis is logarithmic. **Alistair says:** *Enlarge by same factor?*

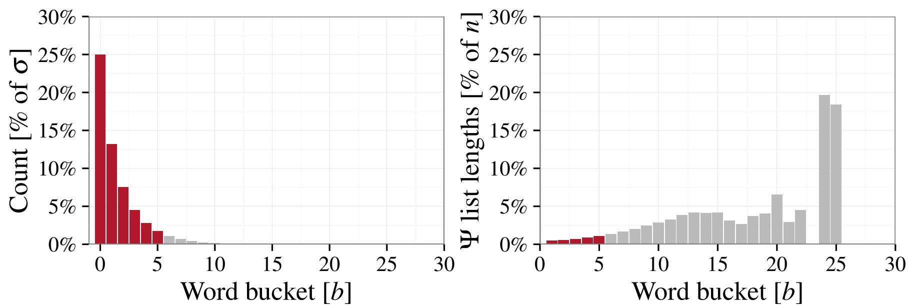

Figure 3: Frequency statistics for german-2048. Bucket b contains words which occur between 2^b and $2^{b+1}-1$ times, with bucket $b=0$ covering words that occur exactly once. The left pane shows the percentage of σ accounted for by each bucket, and the right pane the percentage of n. The low-frequency part of each distribution (less than $k=128$ word occurrences) is marked in red. **Alistair says:** *Enlarge by same factor?*

Note in particular that CSA++ performance is largely unaffected by k, whereas the performance of CSA substantially decreases as k increases. As pattern search is a major part of the cost of the machine translation process described by Shareghi et al. [24], utilizing CSA++ leads to a significant speedup in practical performance.

5 Conclusion

We have described several enhancement's to Sadakane's CSA, and have demonstrated improvements both in terms of compression effectiveness, and also in terms of query throughput for Count queries, especially for large-alphabet applications. If Locate queries are also required, all of the structures explored here must be augmented with SA samples, to allow (sp, ep) ranges to be converted to offsets in T; as future work, we plan to investigate space-speed tradeoffs in that regard as well.

Appendix A: Details of Experimentation

Table 4 provides details of the various methods compared in Section 4. The CSA-SADA results were obtained by executing code authored by Kunihiko Sadakane, available from the Pizza&Chili web site.

References

[1] R. Agarwal, A. Khandelwal, and I. Stoica. "Succinct: Enabling Queries on Compressed Data". In: *USENIX Symp. Networked Systems Design and Impl.* 2015, pp. 337–350.

[2] V. N. Anh and A. Moffat. "Compressed Inverted Files with Reduced Decoding Overheads". In: *Proc. ACM SIGIR Int. Conf. Information Retrieval*. 1998, pp. 290–297.

[3] A. Apostolico, M. Crochemore, M. Farach-Colton, Z. Galil, and S. Muthukrishnan. "Forty Years of Suffix Trees". In: *C. ACM* 59.4 (Mar. 2016), pp. 66–73.

Method	Component	DNA (200 MiB)		XML (200 MiB)		German (2 GiB)	
		% Ψ	MiB	% Ψ	MiB	% Ψ	MiB
CSA++	Samples	–	2.3	–	3.1	–	36.8
	NIL-blocks	0.2	0.0	62.0	0.0	15.7	0.0
	BV-coded blocks	78.0	61.7	10.8	5.9	16.4	32.1
	RL-coded blocks	2.5	0.2	14.4	4.3	3.8	13.6
	EF-coded blocks	19.3	22.6	12.7	20.4	59.6	626.3
	Binary values	0.0	0.0	0.0	0.0	4.4	115.5
	Other structures	–	5.9	–	4.8	–	43.0
	Total space	–	92.7	–	38.5	–	867.2
CSA		–	91.4	–	56.7	–	1061
CSA-OPF		–	111.0	–	48.0	–	839.9
FM-FB-HYB		–	51.3	–	25.6	–	–
FM-HF-HYB		–	51.8	–	32.4	–	1411
FM-AP		–	–	–	–	–	903.3
FM-AP-HYB		–	–	–	–	–	778.5

Table 1: Comparing the space costs of different pattern search indexes, using a blocksize of $k = 128$ throughout. The methods listed in the lower part of the table are from the SDSL library, and are described in detail in Appendix A. Note that not all of the methods are applicable to all of the files.

Blocksize	Random		RLZ factors	
	CSA	CSA++	CSA	CSA++
$k = 64$	1.84	0.74	1.68	0.56
$k = 128$	2.89	0.76	2.73	0.59
$k = 256$	5.10	0.88	4.90	0.73

Table 2: Per-character time in microseconds for RLZ factorization, compared to 23-character random patterns.

Blocksize	Random		NL search	
	CSA	CSA++	CSA	CSA++
$k = 64$	1.86	1.05	1.67	0.63
$k = 128$	2.90	0.99	2.98	0.62
$k = 256$	5.50	0.99	5.99	0.63

Table 3: Per-word time in microseconds for phrase search, compared to 4-word random patterns.

[4] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. "Alphabet Partitioning for Compressed Rank/Select and Applications". In: *Proc. Int. Symp. Alg. and Comp.* 2010, pp. 315–326.

[5] P. Ferragina, R. González, G. Navarro, and R. Venturini. "Compressed Text Indexes: From Theory to Practice". In: *J. Exp. Alg.* 13 (2008).

[6] P. Ferragina and G. Manzini. "Opportunistic Data Structures with Applications". In: *Proc. IEEE Symp. Foundations of Comp. Science*. 2000, pp. 390–398.

[7] S. Gog, T. Beller, A. Moffat, and M. Petri. "From Theory to Practice: Plug and Play with Succinct Data Structures". In: *Proc. Symp. Experimental Alg.* 2014, pp. 326–337.

[8] S. Gog, J. Kärkkäinen, D. Kempa, M. Petri, and S. J. Puglisi. "Faster, Minuter". In: *Proc. Data Compression Conf.* 2016, pp. 53–62.

[9] S. Gog, G. Navarro, and M. Petri. "Improved and Extended Locating Functionality on Compressed Suffix Arrays". In: *J. Discrete Alg.* 32 (2015), pp. 53–63.

[10] A. Golynski, J. I. Munro, and S. S. Rao. "Rank/Select Operations on Large Alphabets: A Tool for Text Indexing". In: *Proc. ACM-SIAM Symp. Discrete Alg.* 2006, pp. 368–373.

[11] R. Grossi, A. Gupta, and J. S. Vitter. "High-Order Entropy-Compressed Text Indexes". In: *Proc. ACM-SIAM Symp. Discrete Alg.* 2003, pp. 841–850.

[12] R. Grossi and J. S. Vitter. "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching". In: *Proc. ACM Symp. Theory of Comput.* 2000, pp. 397–406.

[13] H. Huo, L. Chen, J. S. Vitter, and Y. Nekrich. "A Practical Implementation of Compressed Suffix Arrays with Applications to Self-Indexing". In: *Proc. Data Compression Conf.* 2014, pp. 292–301.

[14] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. "Hybrid Compression of Bitvectors for the FM-Index". In: *Proc. Data Compression Conf.* 2014, pp. 302–311.

Abbreviation	Composition
CSA	csa_sada<enc_vector<coder::elias_gamma,s_Ψ>, 1<<20,1<<20,sa_order_sa_sampling<>,isa_sampling<>>
CSA-OPF	csa_sada<optpfor_vector<s_Ψ>, 1<<20,1<<20,sa_order_sa_sampling<>,isa_sampling<>>
FM-HF-BVIL	csa_wt<wt_huff<bit_vector_il<*bs*>>,1<<20,1<<20>
FM-HF-HYB	csa_wt<wt_huff<hyb_vector<>>,1<<20,1<<20>
FM-HF-RRR	csa_wt<wt_huff<rrr_vector<*b*>>,1<<20,1<<20>
FM-AP	csa_wt_int<wt_ap< wt_huff<bit_vector,rank_support_v5<1>, select_support_scan<1>,select_support_scan<0>>, wm_int<bit_vector,rank_support_v5<1>, select_support_scan<1>,select_support_scan<0>>>, 1<<20,1<<20>
FM-AP-HYB	csa_wt_int<wt_ap< wt_huff<hyb_vector<>>, wm_int<hyb_vector<>> >,1<<20,1<<20>
FM-GMR	csa_wt_int<wt_gmr<>,1<<20,1<<20>
FM-GMR-RS	csa_wt_int<wt_gmr_rs<>,1<<20,1<<20>
FM-FB-BVIL	csa_wt<wt_fbb<bit_vector_il<*bs*>>,1<<20,1<<20>
FM-FB-HYB	csa_wt<wt_fbb<hyb_vector<>>,1<<20,1<<20>
CSA++	csa_sada2<hyb_sd_vector<*s*>,1<<20,1<<20, sa_order_sa_sampling<>,isa_sampling<>>

Table 4: SDSL descriptions of methods used in experiments. Sampling parameters $b \in \{15,31,63,127\}$, $bs \in \{128,256,512,1024\}$, $s \in \{16,32,64,128,256,512,1024\}$, and $s_\Psi \in \{16,32,64,128,512,1024\}$ were varied in the experiments to get different time-space trade-offs. The last three class definitions are available in the `hyb_sd_vector` branch of the library.

[15] V. Mäkinen and G. Navarro. "Succinct Suffix Arrays Based on Run-Length Encoding". In: *Proc. Symp. Combinatorial Pattern Matching*. 2005, pp. 45–56.

[16] U. Manber and G. W. Myers. "Suffix Arrays: A New Method For On-Line String Searches". In: *SIAM J. Comp.* 22.5 (1993), pp. 935–948.

[17] A. Moffat and S. Gog. "String Search Experimentation Using Massive Data". In: *Philosophical Trans. Royal Society A* 372.8 (2014).

[18] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Boston, MA: Kluwer Academic Publishers, 2002.

[19] G. Navarro and V. Mäkinen. "Compressed Full-Text Indexes". In: *ACM Comp. Surveys* 39.1 (2007).

[20] D. Okanohara and K. Sadakane. "Practical Entropy-Compressed Rank/Select Dictionary". In: *Proc. Wkshp. Alg. Engineering and Experiments*. 2007.

[21] G. Ottaviano and R. Venturini. "Partitioned Elias-Fano Indexes". In: *Proc. ACM SIGIR Int. Conf. Information Retrieval*. 2014, pp. 273–282.

[22] M. Petri, A. Moffat, P. C. Nagesh, and A. Wirth. "Access Time Tradeoffs in Archive Compression". In: *Proc. Asian IR Societies Conf.* 2015, pp. 15–28.

[23] K. Sadakane. "New Text Indexing Functionalities of the Compressed Suffix Arrays". In: *J. Alg.* 48.2 (2003), pp. 294–313.

[24] E. Shareghi, M. Petri, G. Haffari, and T. Cohn. "Compact, Efficient and Unlimited Capacity: Language Modeling with Compressed Suffix Trees". In: *Proc. Conf. Empirical Methods in Natural Language Proc.* 2015, pp. 2409–2418.

[25] S. Vigna. "Quasi-Succinct Indices". In: *Proc. ACM Conf. Web Search & Data Min.* 2013, pp. 83–92.

[26] P. Weiner. "Linear Pattern Matching Algorithms". In: *Proc. SWAT*. 1973, pp. 1–11.

[27] H. Yan, S. Ding, and T. Suel. "Inverted index compression and query processing with optimized document ordering". In: *Proc. WWW*. 2009, pp. 401–410.