BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 65, No. 4, 2017
DOI: 10.1515/bpasts-2017-0046

Compact and hash based variants of the suffix array

S. GRABOWSKI* and M. RANISZEWSKI

Institute of Applied Computer Science, Lodz University of Technology, 11 Politechniki Ave., 90-924 Łódź, Poland

Abstract. Full-text indexing aims at building a data structure over a given text capable of efficiently finding arbitrary text patterns, and possibly requiring little space. We propose two suffix array inspired full-text indexes. One, called SA-hash, augments the suffix array with a hash table to speed up pattern searches due to significantly narrowed search interval before the binary search phase. The other, called FBCSA, is a compact data structure, similar to Mäkinen's compact suffix array (MakCSA), but working on fixed size blocks. Experiments on the widely used Pizza & Chili datasets show that SA-hash is about 2–3 times faster in pattern searches (counts) than the standard suffix array, for the price of requiring $0.2n-1.1n$ bytes of extra space, where n is the text length. FBCSA, in one of the presented variants, reduces the suffix array size by a factor of about 1.5–2, while it gets close in search times, winning in speed with its competitors known from the literature, MakCSA and LCSA.

Key words: string matching, full-text indexing, suffix array, compact indexes, hashing.

1. Introduction

The field of text-oriented data structures continues to bloom. Curiously, in many cases several years after ingenious theoretical solutions their more practical (which means: faster and/or simpler) counterparts are presented, to mention only recent advances in rank/select implementations [1] or the FM-index reaching the compression ratio bounded by k-th order entropy with very simple means [2].

Despite the great interest in compact or compressed[1] full-text indexes in recent years [3], we believe that in some applications search speed is more important than memory savings, thus different space-time tradeoffs are worth being explored. The classic suffix array (SA) [4], combining speed, simplicity and often reasonable memory use, may be a good starting point for such research.

In this paper we present two SA-based full-text indexes. One augments the standard SA with a hash table to speed up searches, for a moderate overhead in the memory use, the other can be considered a byte-aligned variant of Mäkinen's compact suffix array [5, 6]. The proposed algorithms, in their most successful variants, turn out to be competitive with other text indexes described in the literature.

2. Preliminaries

We use 0-based sequence notation, that is, a sequence S of length n is written as $S[0 \dots n-1]$. If not stated otherwise, all logarithms throughout the paper are in base 2.

One may define a full-text index over text T of length n as a data structure supporting at least two types of queries, both with respect to a pattern P of length m, where T and P share an integer alphabet of size σ. One query type is count: return the number $occ \geq 0$ of occurrences of P in T. The other query type is locate: for each pattern occurrence report its position in T, that is, such j that $P\ [0 \dots m-1] = T\ [j \dots j+m-1]$.

The suffix array $SA\ [0 \dots n-1]$ for text T is a permutation of the indexes $\{0, 1, \dots, n-1\}$ such that $T[SA\ [i] \dots n-1] \prec T[SA\ [i+1] \dots n-1]$ for all $0 \leq i < n-1$, where the "$\prec$" relation is the lexicographical order. The inverse suffix array SA^{-1} is the inverse permutation of SA: $SA^{-1}\ [j] = i \Leftrightarrow SA\ [i] = j$. The Burrows-Wheeler transform of the text T, denoted as T^{BWT}, can be obtained from T and SA using the formula $T^{BWT}[i] = T\ [(SA\ [i]-1) \bmod n]$.

3. Related work

The full-text indexing history starts with the suffix tree (ST) [7], a trie whose string collection is the set of all the suffixes of a given text, with an additional requirement that all non-branching paths of edges are converted into single edges.

Each ST path is terminated as soon as it points to a unique suffix, whose start position is kept in the corresponding leaf. As there are n leaves, up to $n-1$ internal nodes (as each internal node must have at least two children) and edge labels are represented with pointers to the text, it is easy to see that the suffix tree takes $O(n)$ words of space, i.e., $O(n \log n)$ bits.

Suffix trees can be built in linear time for integer alphabets [8]. Assuming constant-time access to any child of a given node, the search in the ST takes only $O(m + occ)$ time in the worst case. In practice, this is cumbersome for a large alphabet, of size $n^{\omega(1)}$, as it requires using perfect hashing, which also makes the construction time linear only in expectation. A small alphabet is easier to handle, which is one of the reasons of the wide use of suffix trees in bioinformatics.

[1]By the latter we mean indexes with space bounded by $O(nH_0)$ or even $O(nH_k)$ bits, where n is the text length, and H_0 (H_k) the order-0 (order-k) entropy. The former term, compact full-text indexes, is less definite, and may fit any structure with less than $n\log_2 n$ bits of space, at least for "typical" texts.

*e-mail: sgrabow@kis.p.lodz.pl

Manuscript submitted 2016-11-01, revised 2016-12-16, initially accepted for publication 2017-01-23, published in August 2017.

The main problem concerning the discussed index is its large space requirement. Even in the most economical version [9] the ST space use reaches almost $9n$ bytes on average and $16n$ in the worst case, plus the text, for $\sigma \leq 256$, and even more for large alphabets. Most implementations need at least $20n$ bytes.

An important alternative to the suffix tree is the suffix array (SA) [4]. It is an array of n pointers to all text suffixes sorted according to the lexicographic order of these suffixes. The SA needs $n\log n$ bits for its n suffix pointers (indexes), plus $n\log\sigma$ bits for the text, which typically gives $5n$ bytes in total. The pattern search time is $O(m\log n)$ in the worst case and $O(m\log_\sigma n + \log n)$ on average, which can be improved to $O(m + \log n)$ in the worst case using the longest common prefix (lcp) table. Yet Manber and Myers in their seminal paper [4] presented a way of saving several first steps in the binary search: if we know the SA intervals for all the possible first k symbols of the pattern, we can immediately start the binary search in a corresponding interval. We can set k close to $\log_\sigma n$, with $O(n\log n)$ extra bits of space, but constant expected size of the interval, which leads to $O(m)$ average search time and only $O(\lceil m/CL \rceil)$ cache misses on average, where CL is the cache line length expressed in symbols, typically 64 symbols/bytes in a modern CPU. Unfortunately, real texts are far from random, hence in practice, if text symbols are bytes, we can use k up to 3, which offers a limited (yet, non-negligible) benefit. This idea, later denoted as using a lookup table (LUT), is fairly well known, see e.g. its impact in the search over a suffix array on words [10].

The suffix array can be built from the suffix tree by visiting its leaves in order (hence preserving $O(n)$ construction time), yet this approach is impractical. Only in 2003 several algorithms building the SA directly in linear time were presented, e.g., [11], and currently the fastest $O(n)$-time construction algorithm is the one given by Nong [12].

A number of suffix tree or suffix array inspired indexes have been proposed as well, including the suffix cactus [13] and the enhanced suffix array (ESA) [14], with space use usually between SA and ST, yet they are not generally faster than their famous predecessors in the count or locate queries. For example, according to an interesting experimental work [15], ESA may be moderately faster than SA if the alphabet is small (up to around 8 symbols) but SA dominates for larger alphabets.

On a theoretical front, the suffix tray by Cole et al. [16] allows to achieve $O(m + \log\sigma)$ search time, with $O(n)$ worst-case time construction and $O(n\log n)$ bits of space, which was recently improved by Fischer and Gawrychowski [17] to $O(m + \log\log\sigma)$ deterministic time, with preserved construction cost complexities.

Since around 2000 a great surge of interest in succinct data structures, in particular, text indexes, can be observed. Two main ideas in this area are the compressed suffix array (CSA) [18, 19] and the FM-index [20]; see the survey [3] for details.

It was noticed in extensive experimental comparisons [21, 1] that compressed indexes are not much slower, and sometimes comparable, to the suffix array in count queries, but locate is 2–3 orders of magnitude slower if the number of matches is large. This instigated researchers to follow one of two paths in order to mitigate the locate cost for succinct indexes. One, pioneered by Mäkinen [5, 6] and addressed in a different way by González et al. [22, 23], exploits repetitions in the suffix array (the idea is explained in Section 5). The other approach is to build semi-external data structures (see [24, 25] and references therein).

4. Suffix array with deep buckets

The mentioned idea of Manber and Myers with precomputed interval (bucket) boundaries for k starting symbols tends to bring more gain with growing k, but also precomputing costs grow exponentially. Obviously, σ^k integers are needed to be kept in the lookup table. Our proposal is to apply hashing on relatively long strings, with an extra trick to reduce the number of unnecessary references to the text.

We start with building the hash table HT (Fig. 1). The keys inserted to the HT are distinct k-symbol ($k \geq 2$) prefixes of suffixes from the (previously built) suffix array. That is, we process the suffixes in their SA order and if the current suffix shares its k-long prefix with its predecessor, it is skipped (line 08). The value written to HT (line 11) is a pair: (the position in the SA of the first suffix with the given prefix, the position in the SA of the last suffix with the given prefix), denoted in the codes with fields 1 and r, respectively. Linear probing is used as the collision resolution method (lines 15–16). As for the hash function, we used xxhash (https://github.com/Cyan4973/xxHash). We tested also a few alternatives: MurmurHash (http://en.wikipedia.org/wiki/MurmurHash) is practically as good as xxhash, CRC (http://rosettacode.org/wiki/CRC-32) is slightly slower overall (with up to about 3% slower searches), while the loss of sdbm (http://www.cse.yorku.ca/ oz/hash.html) is greater, often exceeding 10%. A more radical approach is to apply a minimal perfect hash function (mphf), which maps a static set of n keys

HT_build($T[0\ldots n-1]$, $SA[0\ldots n-1]$, k, z, $h(.)$)
Precondition: $k \geq 2$

```
(01)  allocate HT[0...z-1]
(02)  for j ← 0 to z-1 do HT[j] ← NIL
(03)  prevStr ← ε
(04)  j ← NIL
(05)  l ← NIL; r ← NIL
(06)  for i ← 0 to n-1 do
(07)      if SA[i] > n-k then continue
(08)      if T[SA[i]...SA[i]+k-1] ≠ prevStr then
(09)          if j ≠ NIL then
(10)              r ← i-1
(11)              HT[j] ← (l,r)
(12)          l ← i
(13)          prevStr ← T[SA[i]...SA[i]+k-1]
(14)          j ← h(prevStr)
(15)          while HT[j] ≠ NIL do
(16)              j ← (j+1) % z
(17)  HT[j] ← (l,n-1)  /* the last SA interval */
(18)  return HT
```

Fig. 1. Building the hash table of a given size z

into $[0 \ldots n-1]$, i.e., obtains the load factor of 100%, without any collisions. This is an attractive option from the theoretical point, unfortunately it is unclear if mphfs can be really competitive if hash computation time is of primary concern. We experimented with the cmph library [26] (https://github.com/zvelo/cmph) to find out that its performance is not satisfactory to our application.

Figure 2 presents the pattern search (locate) procedure. It is assumed that the pattern length m is not less than k. First the range of rows in the suffix array corresponding to the first two symbols of the pattern is found in a lookup table (line 1); an empty range immediately terminates the search with no matches returned (line 2). Then, the hash function over the pattern prefix is calculated and a scan over the hash table performed until no extra collisions (line 5; return no matches) or found a match over the pattern prefix, which give us information about the range of suffixes starting with the current prefix (line 6). In this case, the binary search strategy is applied to narrow down the SA interval to contain exactly the suffixes starting with the whole pattern. As an implementation note: the binary search could be modified to ignore the first k symbols in the comparisons, but it did not help in our experiments, due to specifics of the used A_strcmp function from the asmlib library[2].

```
Pattern_search(T[0...n-1], SA[0...n-1], HT[0...z-1], k, h(.),
               P[0...m-1])
Precondition: m ≥ k ≥ 2
(1)  beg, end ← LUT2[P[0], P[1]]
(2)  if end < beg then report no matches; return
(3)  j ← h(P[0...k-1])
(4)  while true do
(5)      if HT[j] = NIL then report no matches; return
(6)      if (beg ≤ HT[j].l ≤ end) and
           (T[SA[HT[j].l]...SA[HT[j].l]+k-1] = P[0...k-1]) then
(7)          binSearch(T[0...n-1], SA, HT[j].l, HT[j].r, P[0...m-1])
(8)          return
(9)      j ← (j+1) % z
```

Fig. 2. Pattern search with SA-hash

4.1. Reducing the memory for the hash table. Each slot in the hash table (HT) constructed in Fig. 1 contains two 32-bit integers, for the start and the end position of the range of suffixes starting with the corresponding prefix of length k. Yet, it is possible in practice to reduce the second value to 16 bits. To this end, we make use of a lookup table over pairs of symbols (LUT_2) to initially narrow down the interval related to which the range in the HT will be encoded. Then the actual range will be written approximately, with quantized right boundary of the range.

For clarity, let us denote the new hash table with HT_{approx}. The code for building HT_{approx} is shown in Fig. 3. The construction resembles Fig. 1, with the only difference concerning the right boundary of each interval, stored in variable r.

[2] http://www.agner.org/optimize/asmlib.zip, v2.34.

```
HT_approx_build(T[0...n-1], SA[0...n-1], k, z, h(.))
Precondition: k ≥ 2, n ≤ 2^32 - 2^16
(01)  allocate HT_approx[0...z-1]
(02)  for j ← 0 to z-1 do HT_approx[j] ← NIL
(03)  prevStr ← ε
(04)  j ← NIL
(05)  l ← NIL; r ← NIL
(06)  for i ← 0 to n-1 do
(07)      if SA[i] > n-k then continue
(08)      if T[SA[i]...SA[i]+k-1] ≠ prevStr then
(09)          if j ≠ NIL then
(10)              r ← ⌈(i-beg)/step⌉
(11)              HT_approx[j] ← (l, r)
(12)          l ← i
(13)          prevStr ← T[SA[i]...SA[i]+k-1]
(14)          beg, end ← LUT2[prevStr[0], prevStr[1]]
(15)          step ← ⌈(end+1-beg)/(2^16-1)⌉
(16)          j ← h(prevStr)
(17)          while HT_approx[j] ≠ NIL do
(18)              j ← (j+1) % z
(19)  HT_approx[j] ← (l, ⌈(n-1-beg)/step⌉)  /* the last SA interval */
(20)  return HT_approx
```

Fig. 3. Building the hash table with reduced memory

We notice that the range of suffixes starting with any k-gram, where $k \geq 2$, must be nested in the range of suffixes starting with the first two symbols of this k-gram. Such ranges over all possible character pairs are stored in LUT_2, and the access to it (line 14) allows to set the variables *beg* and *end*. In the next line (15), the granularity with which we approximate the right boundary of the suffix range for each k-gram is set and stored in *step*. As we assume that $n \geq 2^{32} - 2^{16}$, then also the values in LUT_2 have this limit. Let us take at look at line 10 (setting r), referring to step, set in line 15 in an earlier iteration of the for loop. For the involved values of i and *end* we notice that $i \leq end + 1$, which in turns implies that $r \in \{0, \ldots, 2^{16} - 1\}$, i.e., can be stored as a 2-byte value. During the pattern search (for which we do not provide a pseudocode) we thus usually have a slightly wider interval for a binary search than in the baseline variant, which means that we trade some search speed for lower space use.

Let us now explain why a similar saving cannot be applied also to the start position of the range. This is because a collision which (unluckily) points to a subrange of the actual HT range that we are looking for could pass undetected. Here is a counterexample. Let us assume that we have two k-long prefixes ($k = 8$): "somethin" and "once in", which have the same hash value (collision). The SA range for "somethin" is [30 200, 30 700] and LUT_2 stores (for "so") the range [30 000, 31 000]. The SA range for "once in" is [10 300, 10 600] and LUT_2 table stores (for "on") the range [10 000, 11 000]. Now we are decoding the range of "somethin" suffixes and there is a collision with "once in". Hence we obtained the SA range $[30\,000 + 300, 30\,000 + 600] = [30\,300, 30\,600]$, which is a subrange of [30 200, 30 700] and we cannot detect a collision. We are searching in a narrower range, so the results may be wrong. Quantizing only the right boundary of the range does not imply a similar problem.

5. Fixed block based compact suffix array

Mäkinen's compact suffix array [5, 6] finds and succinctly represents repeating suffix areas. We propose a variant of this index whose key feature is finding approximate repetitions of suffix areas of predefined size. More precisely, our (fixed size) suffix areas are mostly assembled with up to three references to other runs of suffixes in the suffix array. Choosing the fixed area size allows to maintain a byte-aligned data layout, beneficial for speed and simplicity. Moreover, by setting a natural restriction on one of the key parameters we force the elementary components of the structure to be multiples of 32 bits, which prevents misaligned access to data.

Mäkinen's index was the first {\em opportunistic} scheme for compressing a suffix array, that is such that uses less space on compressible texts. The key idea was to exploit runs in the SA, that is, maximal segments $SA[i \ldots i+\ell-1]$ for which there exists another segment $SA[j \ldots j+\ell-1]$, such that $SA[j+s] = SA[i+s]+1$ for all $0 \le s < \ell$. This structure still allows for binary search, only the accesses to SA cells require local decompression. In our algorithm, called fixed block based compact suffix array (FBCSA), we make use of Mäkinen's observation in a different way. We take suffix areas of fixed size, e.g., 32 bytes: $SA[i \ldots i+31]$, and find for them $u > 0$ other suffix array segments $SA[j_h \ldots j_h+\ell_h-1]$ such that for each $h \in \{0, \ldots, u-1\}$ and $s \in \{0, 1, \ldots, \ell_h - 1\}$ there exists $m \in \{0, 1, \ldots, 31\}$ for which $SA[j_h+s]+1 = SA[i+m]$. Moreover, for a given pair (j_h, ℓ_h) the sequence of chosen values of m is ascending.

Let us now present the idea of FBCSA in more detail. Notice first that if we prepend each suffix from the range $SA[j \ldots (j+bs-1)]$ with the previous symbol in the text, the resulting suffixes form at most σ contiguous groups (segments) in the suffix array. In real texts, however, for most blocks the number of such (non-empty) groups will be much smaller. In FBCSA, the three largest such groups are identified and their start positions in the suffix array are stored. Yet, the predecessors of the "current" suffixes are also kept in a referentially compressed form. To eventually break a reference chain in the Find(i) operation (Fig. 6), every value in $SA[0 \ldots n-1]$ which is a multiple of ss is stored verbatim, where ss is a space-time tradeoff.

The construction algorithm for FBCSA is presented in Fig. 4. As a result, we obtain two arrays, arr_1 and arr_2, which are empty at the beginning, and their elements are always appended at the end during the construction. The elements appended to arr_2 are suffix array indexes (32-bit integers), while arr_1 stores more varied data: single bits, pairs of bits, and 32-bit integers with the current size of arr_2.

The construction makes use of the suffix array SA of text T, the inverse suffix array SA^{-1} and T^{BWT}. Additionally, there are two construction-time parameters: block size bs and sampling step ss. The block size tells how many successive suffix array cells are encoded together and is assumed to be a multiple of 32, for int32 alignment of the structure layout. The parameter ss means that every SA value which is a multiple of ss will be represented verbatim. This sampling parameter is a time-space tradeoff; using larger ss reduces the overall space but decoding a particular SA cell typically involves more recursive invocations.

FBCSA_build($SA[0 \ldots n-1]$, SA^{-1}, T^{BWT}, bs, ss)

```
/* assume n is a multiple of bs */
(01)  arr1 ← [ ]; arr2 ← [ ]
(02)  j ← 0
(03)  while j < n do
          /* current block of the suffix array is SA[j ... j+bs-1] */
(04)      a2s = |arr2|  /* arr2 size in bytes */
(05)      find 3 most frequent symbols in T^BWT[j ... j+bs-1]
              and store them in M[0...2]
          /* if there are less than 3 distinct symbols,
              the trailing cells of M[0...2] are set to NIL) */
(06)      for i ← 0 to bs-1 do
(07)          if T^BWT[j+i] = M[0] then arr1.append(00_2)
(08)          elif T^BWT[j+i] = M[1] then arr1.append(01_2)
(09)              elif T^BWT[j+i] = M[2] then arr1.append(10_2)
(10)                  else arr1.append(11_2)
(11)      pos0 = T^BWT[j ... j+bs-1].pos(M[0])
(12)      pos1 = T^BWT[j ... j+bs-1].pos(M[1])
          /* set NIL if M[1] = NIL */
(13)      pos2 = T^BWT[j ... j+bs-1].pos(M[2])
          /* set NIL if M[2] = NIL */
(14)      arr2.append(SA^-1[SA[j+pos0]-1])
(15)      arr2.append(SA^-1[SA[j+pos1]-1])
          /* append -1 if pos1 = NIL */
(16)      arr2.append(SA^-1[SA[j+pos2]-1])
          /* append -1 if pos2 = NIL */
(17)      for i ← 0 to bs-1 do
(18)          if (T^BWT[j+i] ∉ {M[0],M[1],M[2]}) or
              (SA[j+i] % ss = 0)
(19)              then arr1.append(1_2); arr2.append(SA[j+i])
(20)          else arr1.append(0_2)
(21)      arr1.append(a2s)
(22)      j ← j+bs
(23)  return (arr1, arr2)
```

Fig. 4. Building the fixed block based compact suffix array (FBCSA)

Let us describe the encoding procedure for one block, $SA[j \ldots j+bs-1]$, where j is a multiple of bs (see also a small example in Fig. 5). Encoding this block requires also access to the corresponding block of T^{BWT}, i.e., to $T^{BWT}[j \ldots j+bs-1]$. The following description will be given in points; below the points we explain how we handle some special cases.

1. Let $a2s$ be the size of the array arr_2, expressed in bytes (line 04); initially arr_2 is empty but grows with successive processed blocks.
2. The three most frequent symbols in $T^{BWT}[j \ldots j+bs-1]$ are stored (in arbitrary order) in a small helper array $M[0 \ldots 2]$ (line 05). The symbols from $T^{BWT}[j \ldots j+bs-1]$, read from left to right, are mapped to 2-bit codes: 00_2, 01_2 or 10_2, if they correspond to any of the symbols stored in M, and to 11_2 otherwise. The obtained sequence of bs 2-bit codes is appended to array arr_1 (lines 06–10).
3. Three integers are appended to arr_2: $SA^{-1}[SA[j+pos_0]-1]$, $SA^{-1}[SA[j+pos_1]-1]$ and $SA^{-1}[SA[j+pos_2]-1]$ (lines 14–16), where pos_0, pos_1 and pos_2 are the positions of the first occurrences of $M[0]$, $M[1]$ and $M[2]$, respectively, in $T^{BWT}[j \ldots j+bs-1]$ (lines 11–13).

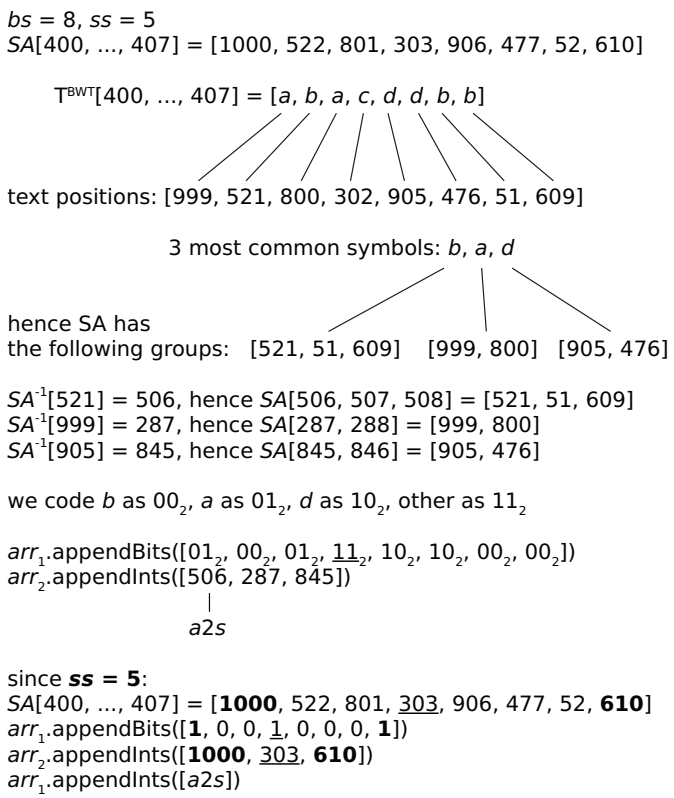

Fig. 5. The FBCSA mechanism and the output arrays arr_1, arr_2 for a single block, presented on a toy example. Note that $bs = 8$ here, while in the real implementation bs must be a multiple of 32. The boldface numbers 1000 and 610 are the SA entries stored explicitly, as their values are multiples of ss. The underlined number 303 is the SA entry whose preceding symbol (namely, $T[302] = c$) is not among the three most common symbols preceding the current block.

4. The sequence $T^{BWT}[j \ldots j + bs - 1]$ is scanned again, from left to right, and for each symbol that belongs to M and whose position in T^{BWT} is such $j + i$ that $SA[j + i] \% ss \neq 0$, we set bit 0, and set bit 1 for the other symbols. Such a bit per suffix is used to distinguish between referentially encoded and explicitly written suffix offsets. The latter ones are those whose stored value (i.e., suffix location in T) modulo ss is 0, or those prepended with a locally rare (i.e., not from M) symbol. The resulting bit-string is appended to arr_1 (lines 17–20).
5. During the scan from the previous point we also append to array arr_2 the values $SA[j + i]$, $0 \leq i < bs$, for those symbols to which bit 1 was assigned (line 19).
6. The value of $a2s$, set in Point 1, is appended to arr_1 (line 21). Note that we used here the size of arr_2 (in bytes) as it was before processing the current block, to allow for easy synchronization between the portions of data in arr_1 and arr_2.

The signalled "special cases" occur when there are less than three distinct symbols in a block. We then write one or two *NIL* values to M, and also use *NIL* or (dummy) -1 values when dealing with the integers written to arr_2; see the comments following the lines 12, 13, 15 and 16 in the code.

Figure 6 presents the function Find(i), which returns $SA[i]$. The helper arrays $bitB$ and $dbitB$ contain respectively bits and pairs of bits (extracted from one or several integers) for the block. The function pc_c (popcount) returns the number of occurrences of symbol (integer) c in the given array of symbols (integers). In modern CPUs pc_1 for a bit-vector of size e.g. 64 is usually available as a single op-code.

Find(arr_1, arr_2, bs, i)

```
/* assume bs is a multiple of 32 */
(01)  of1 ← bs/16
(02)  of2 ← (bs/16) + (bs/32)
(03)  cb_beg ← ⌊i/bs⌋ ∗ (of2 + 1)
(04)  cb_currpos ← i % bs
(05)  d0 ← ⌊cb_currpos/32⌋
(06)  c ← arr1[cb_beg + of2]
(07)  bitB ← int2bits(arr1[cb_beg + of1 ... cb_beg + of1 + d0])
(08)  if bitB[cb_currpos] = 1 then
(09)     return arr2[c + 3 + pc1(bitB[0 ... cb_currpos − 1])]
(10)  else
(11)     d1 ← ⌊cb_currpos/16⌋
(12)     dbitB ← int2dibits(arr1[cb_beg ... cb_beg + d1])
(13)     sym ← dbitB[cb_currpos]
(14)     return Find(arr1, arr2, bs,
            arr2[c + int(sym)] + pc_sym(dbitB[0 ... cb_currpos − 1])) + 1
```

Fig. 6. Find(i) extracts $SA[i]$ from the FBCSA structure

FBCSA simulates the binary search over a plain suffix array. Alas, each lookup for an SA value typically translates to several so-called LF-mappings over the Burrows-Wheeler transform (BWT) of the indexed text. Each LF-mapping is likely to incur a cache miss, which hampers the search performance. To mitigate this effect, we propose a simple yet effective hybrid of FBCSA and the plain SA. Assume, for presentation clarity, that the suffix array size n is a power of 2. We sample out every h-th suffix from SA (where h is a power of 2 and $h \leq n/2$) and the first $\log(n/(2h)) + 1 = \log n - \log h$ binary search steps are performed with reference to this SA subset. Only the last several steps make use of the FBCSA component. The extra space, corresponding to the sampled SA offsets, is n/h words, which is quite small for e.g. $h = 32$. Moreover, the sampled offsets are arranged according to B-tree layout (with B set to 1), for cache friendliness, as advocated in [27, 28]. The search for the right boundary in this variant is performed with the doubling (galloping) technique, which peeks the locations $SA[left + 2^i]$, $i = 0, 1, 2, \ldots$, until it reaches too far and the search continues in the binary manner over the last considered interval. We denote this variant as FBCSA-hyb.

6. Experimental results

All experiments were run on a computer with an Intel i7–4930K 3.4 GHz CPU, equipped with 64 GB of DDR3 RAM and running Ubuntu 15.10 64-bit. The RAM modules were 8×8 GB DDR3–1600 with the timings 11–11–11 (Kingston

KVR16R11D4K4/64). All codes were written in C++ and compiled with g++ 5.2.1 with -O3 option (and for the FBCSA search algorithms with the additional -mpopcnt option). One CPU core was used for the computations. The source codes for the SA-hash and FBCSA algorithms can be downloaded from https://github.com/mranisz/sa/releases/tag/v1.0.0 and https://github.com/mranisz/fbcsa/releases/tag/v1.0.0, respectively.

The test datasets were taken from the popular Pizza \& Chili site (http://pizzachili.dcc.uchile.cl/). For most experiments we used the 200-megabyte versions of the files dna, english, proteins, sources and xml. Only to compare search times of FBCSA variants against Mäkinen's CSA we used 50-megabyte datasets, due to text size limitations of the MakCSA implementation.

In order to test the search algorithms, we generated 500 thousand (count queries) and 10 thousand (locate queries) patterns for each used pattern length. The patterns were extracted randomly from the corresponding datasets (i.e., each pattern returns at least one match).

In a number of experiments, we compared pattern search speeds using the following indexes:

- plain suffix array (SA),
- suffix array with a lookup table over the first 2 symbols (SA-LUT2),
- the proposed suffix array with deep buckets, with hashing the prefixes of length $k = 8$ (only for dna $k = 12$ and for proteins $k = 5$ is used); the load factor α in the hash table was set to 90% (SA-hash, shortened in the figure legends to SA-h),
- a more compact variant of SA-hash, with 6 bytes rather than 8 bytes per entry in the hash table (SA-hash-dense, shortened in the figure legends to SA-hd),
- the proposed fixed block based compact suffix array (FBCSA),
- FBCSA with a lookup table over the first 2 symbols (FBCSA-LUT2),
- FBCSA with a hash of prefixes of length $k = 8$ (only for dna $k = 12$ and for proteins $k = 5$ is used); the load factor in the hash table was set to 90% (FBCSA-hash, shortened in the figure legends to FBCSA-h),
- a more compact variant of FBCSA-hash, with 6 bytes rather than 8 bytes per entry in the hash table (FBCSA-hash-dense, shortened in the figure legends to FBCSA-hd).
- a hybrid of the standard FBCSA and an evenly sampled SA, with approximately $n/32$ sampled suffixes (FBCSA-hyb),
- MakCSA [6],
- LCSA [23],
- FM-V5, a fast FM-index variant with uncompressed rank [1].

The results of the faster indexes are presented in Fig. 7. As expected, SA-hash is the fastest index among the tested ones, followed rather closely by its somewhat more memory frugal variation, SA-hash-dense. The reader may also look at Table 1

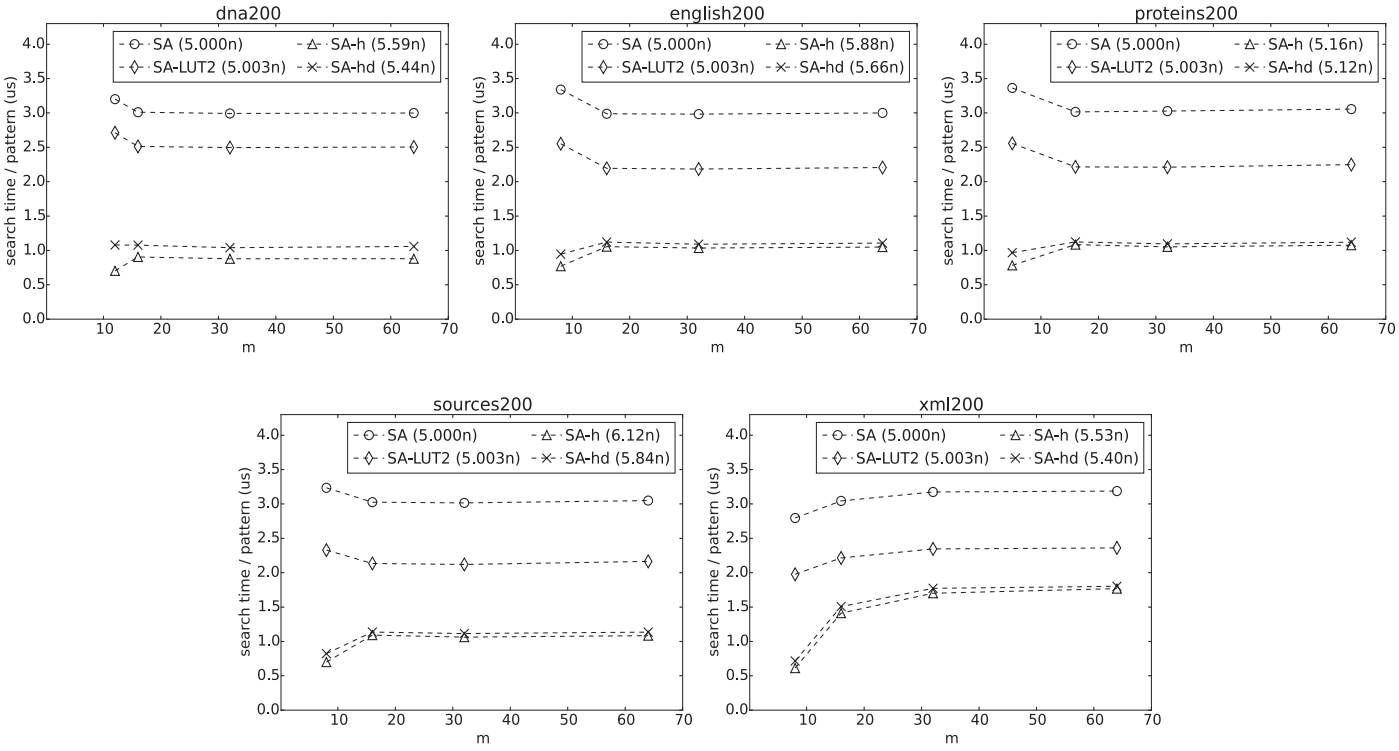

Fig. 7. Pattern search time (count query) for SA-related indexes. All times are averages over 500K random patterns of the same length $m = \{m_{min}, 16, 32, 64\}$, where m_{min} is 8 for most datasets except for dna (12) and proteins (5). The numbers in parentheses are the space uses of the respective indexes (including the text) as multiples of the text length n. The patterns were extracted from the respective texts

Table 1
Speedups with regard to the search speed of the plain SA, for the five 200 MB datasets and pattern lengths $m \in \{16, 64\}$

	dna	english	prot.	sources	xml
$m = 16$					
SA	1.00	1.00	1.00	1.00	1.00
SA-LUT2	1.20	1.36	1.36	1.42	1.37
SA-hash	3.33	2.83	2.78	2.77	2.16
SA-hash-dense	2.80	2.66	2.68	2.66	2.02
$m = 64$					
SA	1.00	1.00	1.00	1.00	1.00
SA-LUT2	1.20	1.36	1.36	1.41	1.35
SA-hash	3.41	2.86	2.84	2.81	1.80
SA-hash-dense	2.83	2.71	2.73	2.69	1.77

Table 2
The number of distinct k-grams (1 … 10) in the 200 MB datasets. The number of distinct 12-grams for dna is 13,752,341

k	dna	english	prot.	sources	xml
1	16	225	25	230	96
2	152	10,829	607	9,525	7,054
3	683	102,666	11,607	253,831	141,783
4	2,222	589,230	224,132	1,719,387	908,131
5	5,892	2,150,525	3,623,281	5,252,826	2,716,438
6	12,804	5,566,993	36,525,895	10,669,627	5,555,190
7	28,473	11,599,445	94,488,651	17,826,241	8,957,209
8	80,397	20,782,043	112,880,347	26,325,724	12,534,152
9	279,680	33,143,032	117,199,335	35,666,486	16,212,609
10	1,065,613	48,061,001	119,518,691	45,354,280	20,018,262

with a rundown of the achieved speedups, where the plain suffix array is the baseline index and its speed is denoted with 1.00.

The SA-hash index has two drawbacks: it requires significantly more space than the standard SA and we assume (at construction time) a minimal pattern length m_{min}. The latter issue may be eliminated, but for the price of even more space use; namely, we can build one hash table for each pattern length up to m_{min} (note that counting queries for those short patterns are handled without any binary search).

We have not implemented this "all-HT" variant, but it is easy to estimate the memory use for each dataset. To this end, one needs to know the number of distinct k-grams for $k \leq m_{min}$ (Table 2). Note that the alphabet size, i.e., the number of 1-grams, for the DNA and proteins datasets is 16 and 25, respectively. These surprisingly large values are explained by the content of the files in the corpus, "polluted" slightly with textual headers, End-of-Line symbols, etc.

An obvious space-time factor in a hash table with open addressing is its load factor α. We checked several values of α on two datasets (Table 3) to conclude that using $\alpha = 90\%$ is a reasonable alternative to $\alpha = 50\%$, as the pattern search times grow by only about 10% or less.

The number of bytes for one hash table with z entries and $0 < \alpha \leq 1$ load factor is, in our implementation of SA-hash, $z \times 8 \times (1/\alpha)$, since each entry contains two 4-byte integers. For example, the hash table for english200 with $\alpha = 90\%$ needed $20{,}782{,}043 \times (8/0.9) = 184{,}729{,}272$ bytes, i.e., 88.1% of the size of the text itself. Note that the overhead in the SA-hash-dense variant with the same α is 66.1% of the text size.

Next, in Table 4 we present the overall space use for the four non-compact SA variants: plain SA, SA-LUT2, SA-hash and SA-hash-dense, plus SA-allHT(-dense), which is a (not implemented) structure comprising a suffix array, a LUT_2 and one hash table for each $k \in \{3, 4, \ldots, m_{min}\}$. The space is expressed as a multiple of the text length n (including the text), which is for example 5.000 for the plain suffix array. We note that the lookup table structures become a relatively smaller fraction when larger texts are indexed. For the variants with hash tables we take two load factors: 50% and 90%.

Table 3
Average pattern search times (in μ) in function of the HT load factor α for the SA-hash algorithm (xxhash function used). Each 200-megabyte dataset name is followed with the pattern length (m)

	HT load factor (%)					
	25	50	70	80	90	95
dna, 12	0.625	0.635	0.648	0.692	0.722	0.792
dna, 16	0.820	0.825	0.861	0.865	0.901	0.977
dna, 32	0.786	0.796	0.817	0.841	0.877	0.949
dna, 64	0.788	0.809	0.825	0.833	0.878	0.952
english, 8	0.742	0.750	0.761	0.762	0.762	0.785
english, 16	1.042	1.043	1.043	1.048	1.054	1.067
english, 32	1.014	1.017	1.025	1.029	1.039	1.047
english, 64	1.029	1.029	1.038	1.043	1.061	1.064

Table 4
Space use for the non-compact data structures as a multiple of the indexed text size (including the text), with the assumption that text symbols are represented in 1 B each and SA offsets are represented in 4 B. The datasets have 200 MB in size. The value of m_{min} for SA-hash-50 and SA-hash-90, used in the construction of these structures and affecting their size, is like in the experiments from Fig. 7. The index SA-allHT-* contains LUT_2 and one hash table for each $k \in \{3, 4, \ldots, m_{min}\}$, when m_{min} depends on the current dataset. The -50 and -90 suffixes in the names denote the hash load factors (in percent)

	dna	engl.	prot.	sources	xml
SA	5.000	5.000	5.000	5.000	5.000
SA-LUT2	5.001	5.001	5.001	5.001	5.001
SA-hash-50	6.050	6.587	5.278	7.010	5.958
SA-hash-90	5.583	5.882	5.154	6.117	5.532
SA-hash-dense-90	5.438	5.661	5.116	5.838	5.399
SA-allHT-50	6.472	8.114	5.296	9.736	7.353
SA-allHT-90	5.818	6.730	5.164	7.631	6.307
SA-allHT-dense-90	5.613	6.298	5.123	6.973	5.980

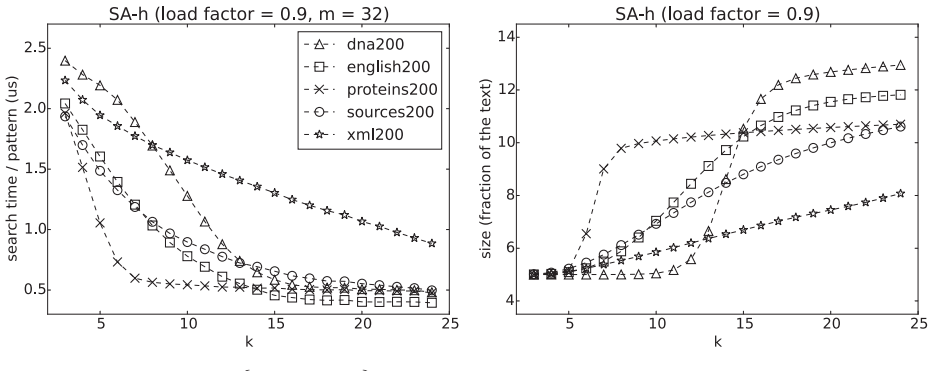

Fig. 8. SA-hash performance with varying k in $\{3, 4, \ldots, 24\}$. Left figure: count times, right figure: space usage as a multiple of the text size. All times are averages over 500K random patterns of the same length $m = 32$. The patterns were extracted from the respective texts

It may also be interesting to see how the SA-hash search performance and space usage change with varying the parameter k (Fig. 8). We assumed the load factor 90% and the timings are given for the pattern length $m = 32$. The results for each dataset are presented as separate series. From the right figure we can notice, for example, that the number of distinct k-grams for dna200 is rather small for k up to 10 or 11, but then suddenly starts to grow fast. An opposite phenomenon is observed for xml200. This is related to the fact that for DNA data even a high-order entropy is not much less than 2, while for natural language (and XML) data it is about 4–5 in order-0, but may drop even below 1 in high-order models (cf. the Pizza & Chili text statistics, available at http://pizzachili.dcc.uchile.cl/texts.html).

The performance of SA-hash depends on the variance of interval widths. If the frequency of the k-grams from a given text tends to be more or less the same, we can expect a good time-space tradeoff, which is the case of proteins200 (note the small overhead for this dataset in Table 4). In natural language texts, however, the frequency of k-grams varies significantly. It should be also noted that if the patterns are uniformly randomly selected from the text, it is more likely to draw out a pattern starting with a frequent k-gram (and thus gaining relatively little from the SA-hash idea) than a pattern starting with a rare k-gram. It is easy to underestimate this effect, therefore a reader is recommended to study Table 1 in [29]. For example, for english200 and pattern length 16, the median pattern frequency in the text is about 3, while the mean frequency is as large as 156 (for xml200 the gaps are even more striking). To combat this effect, we made up a SA-hash variant with double hashing, dividing the suffix array into intervals based on prefixes of varying lengths. More precisely, we set three integer values, b, k_1 and k_2, and examine all strings from the text starting with a given substring of length k_1: if and only if the interval width is more than b and the successor strings of length k_2 have large enough order-0 entropy, we use a second hash over the strings of length $k_1 + k_2$. The entropy criterion is natural in this application; low entropy means that the following k_2 symbols in the given context of length k_1 are not very diverse and splitting the interval into subintervals should not be profitable, considering both speed and space use. On the other hand, a high enough entropy implies a significant diversity of the following k_2 symbols and encourages to split the interval.

Unfortunately, this variant gives little in practice, as briefly reported in Fig. 9 on three datasets. The possible speed improvement of a few percent also implies space usage by a few percent greater than of the baseline SA-hash version. Worse, the trends are not very clear (frequent 'spikes').

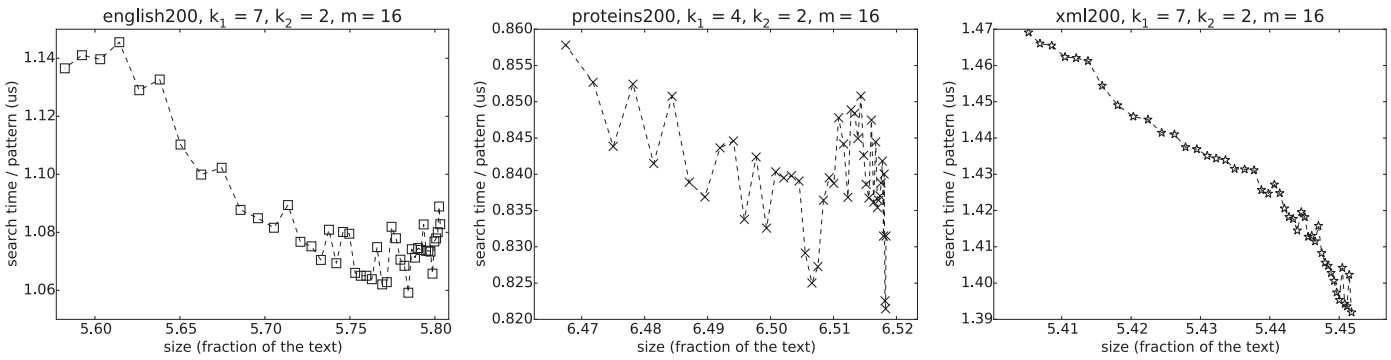

Fig. 9. Index sizes and pattern search times for a SA-hash variant with intervals based on prefixes of varying lengths. The interval width threshold b was set to 128 in all cases, the parameters k_1 and k_2 are shown at the top of the figures. Each series is obtained with changing the entropy threshold from 5.0 down to 0.5, with step 0.1

In the next set of experiments we evaluated the FBCSA index variants, considering the space use, pattern search times (Figs 10 and 11), times to access (extract) one random SA cell (Fig. 12), times to access (extract) multiple consecutive SA cells (Fig. 13).

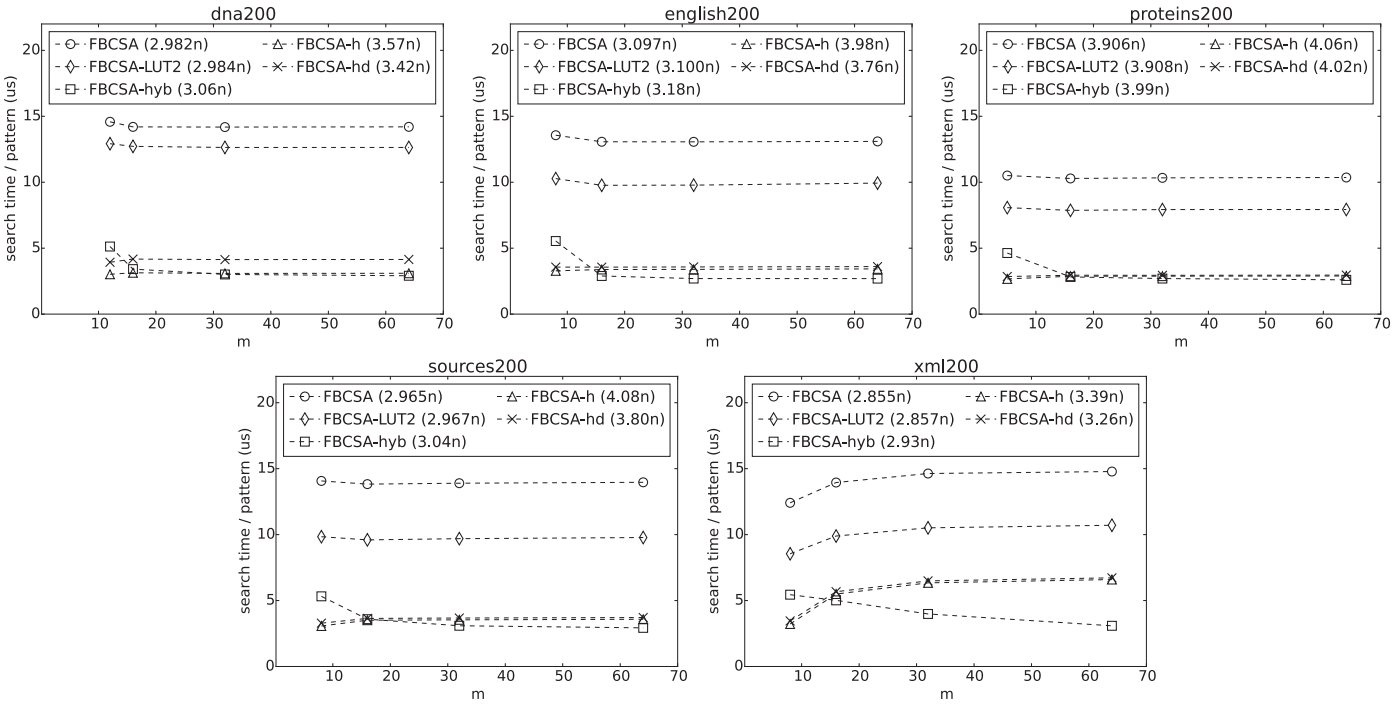

Fig. 10. Pattern search time (count query) for FBCSA-related indexes. All times are averages over 500K random patterns of the same length $m = \{m_{min}, 16, 32, 64\}$, where m_{min} is 8 for most datasets except for dna (12) and proteins (5). The numbers in parentheses are the space uses of the respective indexes (including the text) as multiples of the text length n. The patterns were extracted from the respective texts

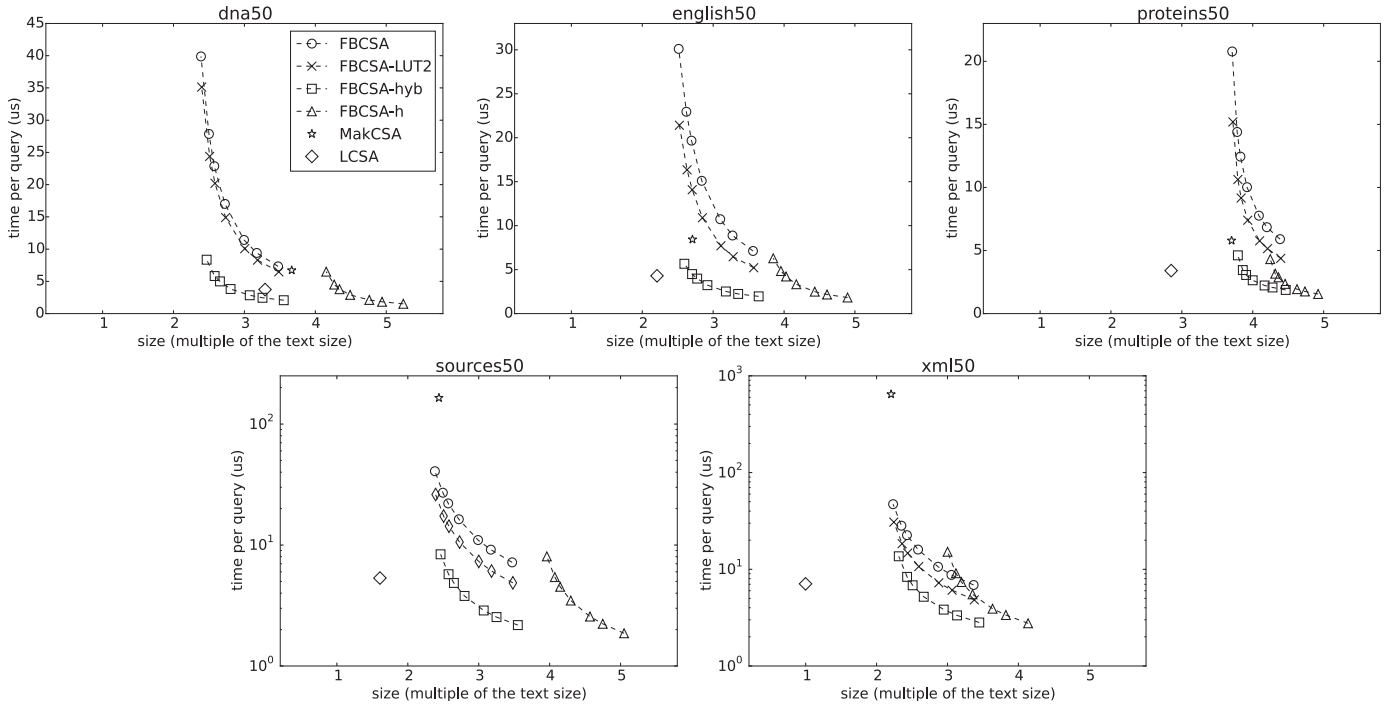

Fig. 11. Pattern search time (count query) for FBCSA-related indexes. The different results in a series are obtained from varying the sampling parameter ss in $\{3, 4, 5, 8, 12, 16, 32\}$, while bs is set to 32. All times are averages over 500K random patterns of the same length $m = 16$. The patterns were extracted from the respective texts. Note the logarithmic scale for the sources50 and xml50 datasets

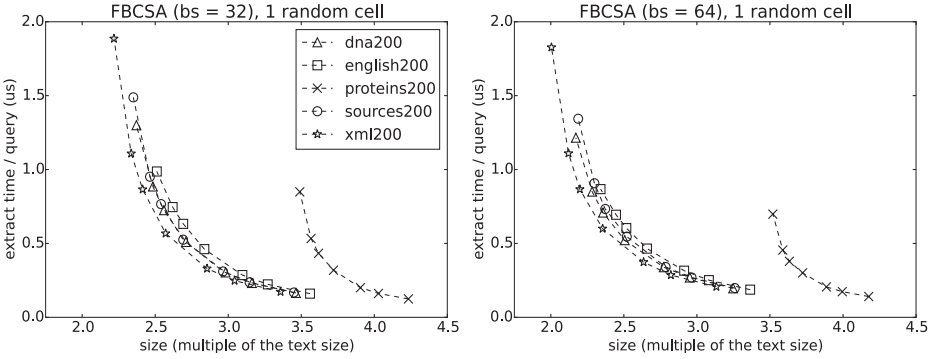

Fig. 12. FBCSA index sizes and cell access times with varying *ss* parameter ($\{3, 4, 5, 8, 12, 16, 32\}$). The parameter *bs* was set to 32 (left figure) or 64 (right figure). The times are averages over 10M random cell accesses

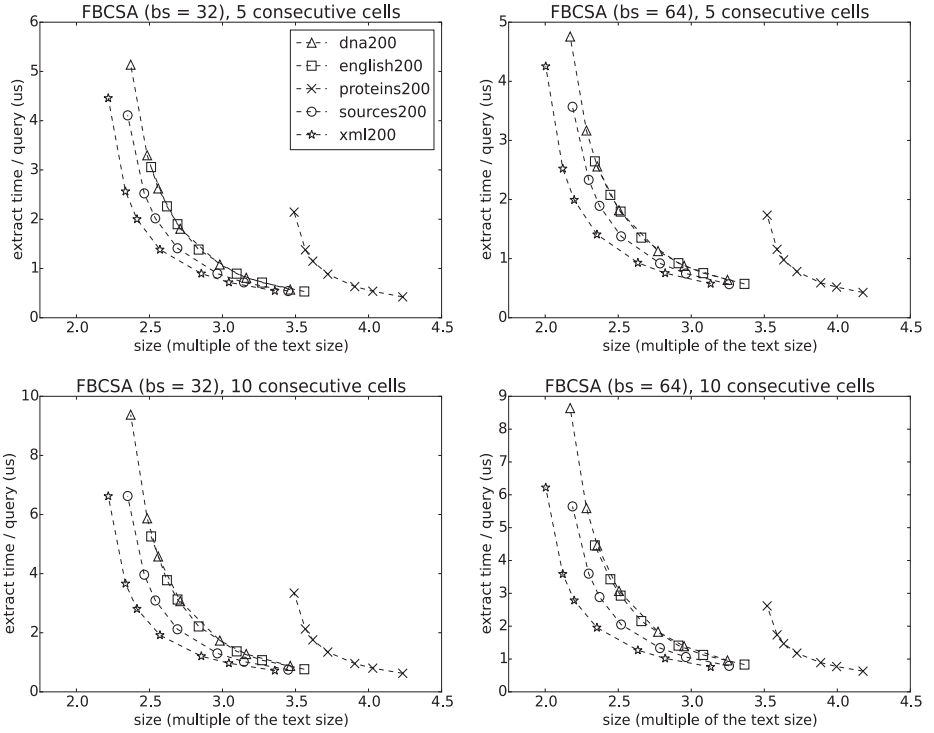

Fig. 13. FBCSA, extraction time for $c = 5$ (top figures) and $c = 10$ (bottom figures) consecutive cells, with varying *ss* parameter ($\{3, 4, 5, 8, 12, 16, 32\}$). The parameter *bs* was set to 32 (left figures) or 64 (right figures). The times are averages over 1M random cell run extractions

In Fig. 10 we vary the pattern length *m* for fixed *bs* (set to 32) and *ss* (set to 5). As we can see, both the hash-based and the hybrid variants boost significantly the performance of the standard FBCSA, with some penalty in the space (clearly smaller for the case of FBCSA-hyb).

We also compared FBCSA variants against MakCSA and LCSA. Alas, it was possible to use MakCSA only for 50-megabyte datasets (LCSA could be run on the 200-megabyte datasets, yet it crashes in all tests on dna200 and in several tests on some other datasets, as mentioned later). The results of our comparison in count queries are shown in Fig. 11. (for $m = 16$ and $bs = 32$, while *ss* varies from 3 to 32). MakCSA is slow on sources50 and xml50, needs relatively large space on dna50 yet obtains decent space-time tradeoffs on the remaining two datasets. It is generally not competitive with FBCSA-hyb though. LCSA, on the other hand, is a much stronger competitor, in most cases easily winning in the used space, yet FBCSA-hyb can be faster by a factor of 2 or more, if we agree to a significantly larger memory requirements. FBCSA-hash (denoted as FBCSA-h in the figures) is sometimes even faster than FBCSA-hyb, but uses even more space.

We tried to compare FBCSA against its competitors in extract queries (Figs 12, 13). In this experiment, *ss* varies

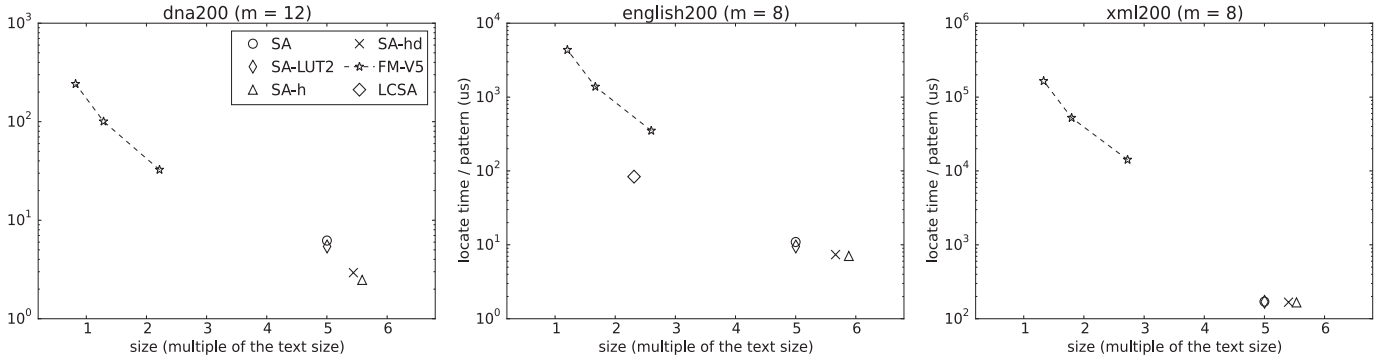

Fig. 14. Pattern locate time for SA-related indexes. All times are averages over 10K random patterns. The patterns were extracted from the respective texts. Note the logarithmic scale on the Y-axis

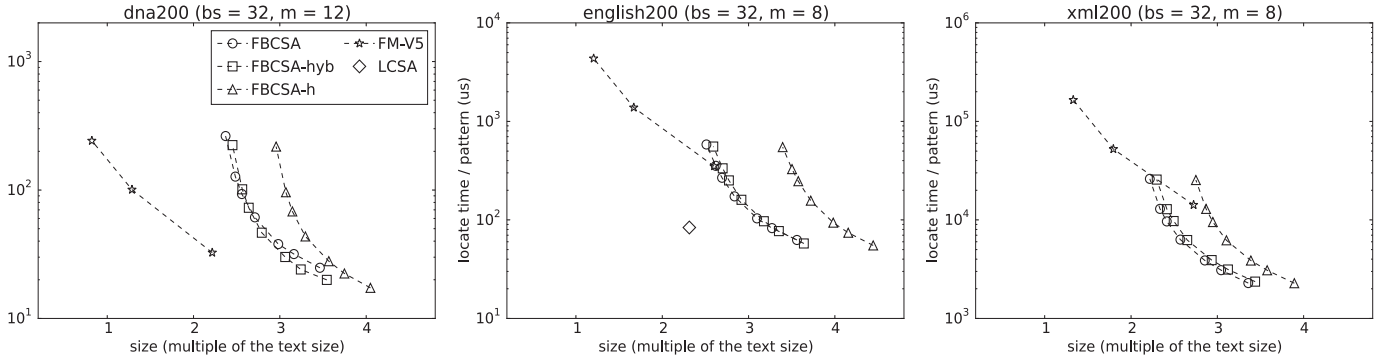

Fig. 15. Pattern locate time for FBCSA-related indexes. The different results in a series are obtained from varying the sampling parameter *ss* in $\{3, 4, 5, 8, 12, 16, 32\}$, while *bs* is set to 32. All times are averages over 10K random patterns. The patterns were extracted from the respective texts. Note the logarithmic scale on the Y-axis

from 3 to 32, and for *bs* we set the values 32 and 64. Using $bs = 64$ results in better compression but decoding a cell is also slightly slower (see Fig. 12). Unfortunately, MakCSA [6] cannot (directly) access single SA cells and we were unable to run LCSA [23] in this kind of queries (despite discussing this issue with the LCSA authors). From the comparison with the results presented in [23] we conclude that FBCSA is a few times faster in single cell access than the other related algorithms, MakCSA [6] (augmented with a compressed bitmap from [30] to extract arbitrary ranges of the suffix array) and LCSA [23]. Extracting *c* consecutive cells is not however an efficient operation for FBCSA (as opposed to MakCSA and LCSA, see Figs 5–7 in [23]), yet for small *ss* the time growth is slower than linear, due to a few sampled (and thus written explicitly) SA offsets in a typical block (Fig. 13). Therefore, in extracting only 5 or 10 successive cells our index is still competitive.

So far, we tested count and extract queries. In Figs 14 and 15 we present the locate results for the fast (SA-related) and more compact (FBCSA-related) indexes, respectively. The available LCSA implementation crashes on dna200 ($m = 12$), sources200 and xml200 ($m = 8$). We point out that in particular LCSA was targeted as a compact SA variant with fast locate, a property unavailable for most compressed indexes, e.g., from the FM-index family. For this reason, a comparison of our variants against LCSA may be interesting. We can see that the compressed solution, FM-V5, although most succinct, in locate queries is slower not only than SA-based indexes (which take much more space, but are faster by at least an order of magnitude), but also than FBCSA variants. LCSA is a practical choice, yet FBCSA may win in speed for the price of using more space. We also note that full evaluation of LCSA is difficult because of the mentioned crashes of the existing implementation.

7. Conclusions

We presented two simple full-text indexes. One, called SA-hash, speeds up standard suffix array searches by reducing significantly the initial search range, thanks to a hash table storing range boundaries of all intervals sharing a prefix of a specified length. The expected speedup by a factor around 3, compared to a standard SA, may be worth the extra space in many applications.

The other presented data structure is a compact variant of the suffix array, related to Mäkinen's compact SA [6]. Our solution works on blocks of fixed size, which provides int32 alignment of the layout. This index is rather fast in single cell access, but not competitive if many (e.g., 100) consecutive cells are to be extracted.

Several aspects of the presented indexes require further study. In case of plain text, the standard suffix array component may be replaced with a suffix array on words [10], with possibly new interesting space-time tradeoffs. The idea of deep buckets may be incorporated into some compressed indexes, e.g., to save on the several first LF-mapping steps in the FM-index.

Acknowledgement. The work was supported by the Polish National Science Centre under the project DEC-2013/09/B/ST6/03117 (both authors).

References

[1] S. Gog and M. Petri, "Optimized succinct data structures for massive data", *Soft. Pract. Exper.* 44 (11), 1287–1314 (2014).

[2] J. Kärkkäinen and S.J. Puglisi, "Fixed block compression boosting in FM-indexes", *18th Int. Symp. String Processing and Information Retrieval, SPIRE 2011*, 174–184 (2011).

[3] G. Navarro and V. Mäkinen, "Compressed full-text indexes", *ACM Comput. Surv.* 39 (1), Article 2 (2007).

[4] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches", *1st Annual ACM-SIAM Symp. Discrete Algorithms, SODA 1990*, 319–327 (1990).

[5] V. Mäkinen, "Compact suffix array", *11th Int. Symp. Combinatorial Pattern Matching, CPM 2000*, 305–319 (2000).

[6] V. Mäkinen, "Compact suffix array – a space-efficient full-text index", *Fund. Inform.* 56 (1–2), 191–210 (2003).

[7] P. Weiner, "Linear pattern matching algorithm", *14th Annual IEEE Symp. Switching and Automata Theory*, 1–11 (1973).

[8] M. Farach, "Optimal suffix tree construction with large alphabets", *38th IEEE Annual Symp. Foundations of Computer Science, FOCS 1997*, 137–143 (1997).

[9] S. Kurtz and B. Balkenhol, "Space efficient linear time computation of the Burrows and Wheeler transformation", In: Numbers, Information and Complexity, Kluwer Academic Publishers, 2000, pp. 375–383.

[10] P. Ferragina and J. Fischer, "Suffix arrays on words", *18th Int. Symp. Combinatorial Pattern Matching, CPM 2007*, 328–339 (2007).

[11] J. Kärkkäinen and P. Sanders, "Simple linear work suffix array construction", *30th Int. Coll. Automata, Languages and Programming, ICALP 2003*, 943–955 (2003).

[12] G. Nong, S. Zhang, and W. H. Chan, "Two efficient algorithms for linear time suffix array construction", *IEEE Trans. Comput.* 60 (10), 1471–1484 (2011).

[13] J. Kärkkäinen, "Suffix cactus: A cross between suffix tree and suffix array", *6th Int. Symp. Combinatorial Pattern Matching, CPM 1995*, 191–204 (1995).

[14] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "The enhanced suffix array and its applications to genome analysis", *2nd Int. Workshop Algorithms in Bioinformatics, WABI 2002*, 449–463 (2002).

[15] N. Grimsmo, *On Performance and Cache Effects in Substring Indexes*, Tech. Rep. IDI-TR-2007–04, Department of Computer and Information Science, NTNU, Trondheim, 2007.

[16] R. Cole, T. Kopelowitz, and M. Lewenstein, "Suffix trays and suffix trists: Structures for faster text indexing", *30th Int. Coll. Automata, Languages and Programming, Part I, ICAPL 2006*, 358–369 (2006).

[17] J. Fischer and P. Gawrychowski, "Alphabet-dependent string searching with wexponential search trees", *26th Int. Symp. Combinatorial Pattern Matching, CPM 2015*, 160–171 (2015).

[18] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching", *32nd ACM Symp. Theory of Computing, STOC 2000*, 397–406 (2000).

[19] K. Sadakane, "Succinct representations of lcp information and improvements in the compressed suffix arrays", *13th Annual ACM-SIAM Symp. Discrete Algorithms, SODA 2002*, 225–232 (2002).

[20] P. Ferragina and G. Manzini, "Opportunistic data structures with applications", *41th IEEE Annual Symp. Foundations of Computer Science, FOCS 2000*, 390–398 (2000).

[21] P. Ferragina, R. González, G. Navarro, and R. Venturini, "Compressed text indexes: From theory to practice", *ACM J. Exp. Algorithmics* 13, 12:1.12–12:1.31 (2009).

[22] R. González and G. Navarro, "Compressed text indexes with fast locate", *18th Int. Symp. Combinatorial Pattern Matching, CPM 2007*, 216–227 (2007).

[23] R. González, G. Navarro, and H. Ferrada, "Locally compressed suffix arrays", *ACM J. Exp. Algorithmics* 19 (1), Article 1 (2014).

[24] S. Gog and A. Moffat, "Adding compression and blended search to a compact two-level suffix array", *20th Int. Symp. String Processing and Information Retrieval, SPIRE 2013*, 141–152 (2013).

[25] S. Gog, A. Moffat, J. S. Culpepper, A. Turpin, and A. Wirth, "Large-scale pattern search using reduced-space on-disk suffix arrays", *IEEE Trans. Knowl. Data Eng.* 26 (8), 1918–1931 (2014).

[26] F.C. Botelho, *Near-Optimal Space Perfect Hashing Algorithms*, Ph.D. Dissertation, Federal University of Minas Gerais, Department of Computer Science, Belo Horizonte, Brazil (2008).

[27] B. Schlegel, R. Gemulla, and W. Lehner, "K-ary search on modern processors", *5th Int. Workshop Data Management on New Hardware, DaMoN 2009*, 52–60 (2009).

[28] P.-V. Khuong and P. Morin, "Array layouts for comparison-based searching", *CoRR*, abs/1509.05053 (2015).

[29] A. Moffat and S. Gog, "String search experimentation using massive data", *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical, and Engineering Sciences* 372 (2016), 20130135 (2014).

[30] R. Raman, V. Raman, and S.S. Rao, "Succinct indexable dictionaries with applications to encoding *k*-ary trees and multisets", *13th Annual ACM-SIAM Symp. Discrete Algorithms, SODA 2002*, 233–242 (2002).