Computing and Informatics, Vol. 38, 2019, 555–574, doi: 10.31577/cai_2019_3_555

SUFFIX ARRAYS WITH A TWIST

Tomasz M. Kowalski, Szymon Grabowski

Łódź University of Technology
Institute of Applied Computer Science
Al. Politechniki 11, 90–924 Łódź, Poland
e-mail: {tkowals, sgrabow}@kis.p.lodz.pl

Kimmo Fredriksson

School of Computing, University of Eastern Finland
P.O.B. 1627, FI-70211 Kuopio, Finland
e-mail: kimmo.k.k.fredriksson@gmail.com

Abstract. The suffix array is a classic full-text index, combining effectiveness with simplicity. We discuss three approaches aiming to improve its efficiency even more: changes to the navigation, data layout and adding extra data. In short, we show that i) the way how we search for the right interval boundary impacts significantly the overall search speed, ii) a B-tree data layout easily wins over the standard one, iii) the well-known idea of a lookup table for the prefixes of the suffixes can be refined with using compression, iv) caching prefixes of the suffixes in a helper array can pose another practical space-time tradeoff.

Keywords: Suffix array, data structures, text indexes, hashing

Mathematics Subject Classification 2010: 68W32

1 INTRODUCTION

Everybody knows the suffix array (SA) [1], a simple full-text index data structure capable of finding the *occ* occurrences of a pattern P of length m in $O(m \log n + occ)$

time, where n is the length of the indexed text. The search mechanism consists in two binary searches, for the left and the right boundary of the interval of text suffixes starting with P, in the array of suffix offsets arranged in the lexicographical order of their text content. The performance of the suffix array can serve as a measuring stick for more advanced (e.g., compressed) text indexes [2] and at least for this reason it is important to know how to implement it efficiently and what space-time tradeoffs are possible.

The suffix array can be perceived as a simplification of the suffix tree (ST) [3], a tree 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. Indeed, a suffix array can be obtained from a suffix tree by visiting its leaves in order (from left to right, obtained by depth-first traversal of the ST). Depending on the implementation, the pattern search over ST takes either $O(m \log \sigma + occ)$ or $O(m + occ)$ time (where the latter variant involves perfect hashing). ST can be built in linear time for integer alphabets [4]; a result that directly translates to linear-time SA construction (albeit more direct and economical linear-time SA construction algorithms were found later). The practical performance of ST and SA is rather comparable, yet implementation details may be important; for example, if the number of matches is large (which is typical for short patterns), the suffix array may be even by an order of magnitude faster than the suffix tree [5].

A number of attempts have been made to improve the time complexities of full-text indexes. For example, the suffix tray by Cole et al. [6], which can be seen as a cross of the suffix tree and the suffix array, allows to achieve $O(m + \log \sigma)$ search time, with $O(n)$ worst-case time construction and $O(n \log n)$ bits of space. Later, Fischer and Gawrychowski [7] reduced the search time to (deterministic) $O(m + \log \log \sigma)$, with preserved construction cost complexities. Even better time complexity, $O(m + \log \log_w \sigma)$ (where $w \geq \log n$ is the machine word size), for a compressed (sic!) index and deterministic linear time construction was recently achieved by Munro et al. [8]. Bille et al. [9] showed how to search for a *packed* pattern in a (standard) suffix array in $O(m/\alpha + \log n)$ time, where α is the number of characters one can pack in a machine word. In the same work, they presented a more involved construction allowing to search for a packed pattern in $O(m/\alpha + \log m + \log \log \sigma)$ time; the index size is still $O(n)$ words (or $O(n \log n)$ bits). We are not aware of any implementations of the algorithms mentioned in this paragraph, which means that they remain theoretical achievements so far.

The body of research on engineering the suffix array is surprisingly scarce. Although the basic SA idea can be easily grasped even by high-school students, many design choices from the implementor's point of view are not obvious. Let us pose a few questions:

1. Can the binary search strategy be replaced with a faster one, e.g., based on interpolation search?
2. As *occ* is usually small, what is practically the best way to find the right interval boundary once the left boundary is known?

3. Can we change the data layout of suffixes in order to obtain more local memory accesses?
4. How can we augment the suffix array with a moderate amount of extra data, to initially reduce the search interval and/or speed up string comparisons?

The answer to some of them is known, yet in this work we are dealing with the mentioned issues in a more systematic way.

The contributions of our paper are as follows. We show that the idea of k-ary heap layout of a sorted array, known from the earlier works [12, 13], makes practical sense also for the suffix array, due to increased data access locality. We discuss the impact of the technique for finding the right interval boundary in the suffix array on the overall performance. It is also noticed that augmenting the suffix array with extra data boosts the performance; novel techniques presented in this work include caching prefixes of the suffixes in a helper array and using a lookup table with Huffman-compressed keys.

2 IDEAS AND INCARNATIONS

The considered ideas are divided into three groups and each of them is described in a separate subsection. First we discuss non-standard SA traversal strategies. Later we advocate for alternative data layouts, beneficial for the search speed. In the last subsection some ways to augment the suffix array with extra data, to make the pattern search even faster, are proposed.

2.1 Navigating over the Suffix Array

A textbook alternative to binary search is interpolation search, which performs a number of "guesses" concerning the query's location based on the query value and the assumed distribution of keys. It is well-known that interpolation search over constant-size keys achieves $O(\log \log n)$ expected time not only for the simplest case, i.e., uniformly random distribution [10], yet we are not aware of any published experiments regarding text suffixes. Unfortunately, a straightforward interpretation of string prefixes (which have a lot of duplicates) as integers and standard linear interpolation yielded rather disappointing results.

Another question concerning the navigation over the SA is how the right interval boundary should be found. We examine two methods: a naive one performs the binary search over the range $left \ldots n$ of suffixes, where $left$ is the position of the least suffix greater or equal to the pattern, and the doubling (galloping) algorithm, 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. Note that the time complexity of the right interval boundary search improves in this way from $O(m \log n)$ to $O(m \log occ)$.

We should also mention here using non-standard CPU instructions for binary search. Wide registers together with single-instruction multiple-data (SIMD) in-

struction sets are a popular extension of modern CPUs, including Intel's Pentium 4, Core 2, Nehalem and more recent architectures, Intel's Xeon, AMD's Phenom, Bulldozer and Ryzen, and ARM Cortex-A mobile processors. Zhou and Ross [11] proposed a SIMD-ized version of binary search (and other database operations) that is geared towards small datasets, up to a few hundred keys. Significant speedups were obtained as a result of the elimination of branch misprediction effects.

searchTree(pat, n, N, $step$)

```
(01)  node ← 0; beg ← n
(02)  while node < N do  /* search down the tree from the top */
(03)     (c, beg) ← searchNode(pat, beg, node, 0, step)
(04)     node ← childNode(node, c)
(05)  if isMaxPattern(pat) then  /* pattern is lexicographically the greatest */
(06)     return (beg, n)
(07)  pat ← incPattern(pat)
(08)  end ← beg; i ← end; endNode ← node(end)
(09)  while true do  /* search up the tree from the current node */
(10)     if pat < T[karySA[end]] then break
(11)     i ← end + 1; node ← endNode
(12)     while true do  /* search for previous beg value */
(13)          if endNode = 0 then
(14)               end ← n; break 2
(15)          (endNode, c) ← parent(endNode)
(16)          if c < k − 1 then break
(17)     end ← index(endNode) + c
(18)  if end = beg then  /* pattern not found */
(19)     return (beg, beg)
(20)  c ← elemOffset(i)
(21)  (c, end) ← searchNode(pat, end, node, c, step)
(22)  node ← childNode(node, c)
(23)  while node < N do  /* search down the tree from the current node */
(24)     (c, end) ← searchNode(pat, end, node, 0, step)
(25)     node ← childNode(node, c)
(26)  return (beg, end)
```

Figure 1. The *searchTree*$(pat, n, N, step)$ function, returning the first and the last index in the search tree corresponding to the range of suffixes of the indexed text starting with the string pat. The parameters n and N ($N \leq n$) refer to the number of suffixes and the number of nodes in the tree, respectively. The parameter $step$ is passed to the searchNode function.

2.2 Linearized k-ary Tree Data Layout

Binary search over a sorted array is equivalent to walking down a path in a complete binary search tree. Schleger et al. [12] noticed that changing the tree layout from binary to k-ary ($k > 2$), together with linearization of the search tree, may be more cache-friendly and also convenient for SIMD processing. In their experiments (Intel Core i7) it achieved a speedup of as much as 3 up to 4.5 for 32-bit numbers and 2 to 2.5 for 64-bit numbers, compared to a plain binary search. This data organization can also be called an (implicit) B-tree layout [13], where the case of $B = 1$ (a complete binary tree with the root going first, then followed by its both children, etc.) is called the Eytzinger layout (dating back to old history) or the heap-order layout, as this method was proposed by Williams for an implementation of binary heaps [14]. We apply the presented idea to the suffix array, which, to our knowledge, has not been tried before. Note that setting the B-tree layout for a suffix array cannot be comparably successful as for, e.g., integers, as the accesses to the text are still at "random" areas.

The pseudocodes of algorithms on the non-standard layout are presented in Figures 1–3. The used notation and primitives (i.e., helper functions) need to be explained beforehand. The term "index" will refer to the position in a linearized k-ary tree, while "offset" to the position relative to the beginning of the node (i.e., the index relative to the beginning of the node). We use the following symbols and helper function names:

- n is the number of SA elements, i.e., the text length,
- N is the number of nodes in the tree, i.e., $N = \lceil n/B \rceil$,
- $index(node)$ returns the index of the first element in the given node,
- $node(index)$ returns the number of the node containing the given index,
- $childNode(node, c)$ returns the number of the c^{th} child node of the node,
- $childNum(node)$ returns the number of the node among its parent's children,
- $elemOffset(idx)$ returns the offset of the element,
- $parent(node)$ returns a pair (p, off), where p is the parent node number and off is the smallest offset of an element in the parent node referring to a suffix not smaller than suffixes in $node$,
- $incPattern(pat)$ returns the next pattern of the same length in lexicographical order. In the (very rare) case when pat is the lexicographically greatest pattern, the lexicographically smallest pattern of the same length is returned (however, such cases do not occur in our code),
- $isMaxPattern(pat)$ tests if pattern is lexicographically the greatest.

Figure 1 presents a pseudocode of the function $searchTree(pat, n, N, step)$, traversing an n-element B-tree structure comprised of N nodes, in order to return the pair (beg, end). The value of beg (resp. end) is the index of an element in the

tree corresponding to the lexicographically smallest suffix not smaller (resp. suffix greater) than the pattern *pat*. As the tree element IDs have values in $\{0, 1, \ldots, n-1\}$, the special case of *end* set to n means that there are no suffixes lexicographically greater than *pat*.

searchNode(*pat*, *idx*, *node*, *startOff*, *step*)

```
(01)  c ← startOff + (step − 1); j ← index(node) + c
(02)  while c < k − 1 do
(03)     if pat < T[karySA[j]] then
(04)           idx ← j; break
(05)     c ← c + step; j ← j + step
(06)  guard ← c
(07)  if guard > k − 1 then guard ← k − 1
(08)  c ← c − (step − 1); j ← j − (step − 1)
(09)  while c < guard do
(10)     if pattern < T[karySA[j]] then
(11)           idx ← j; break
(12)     c ← c + 1; j ← j + 1
(13)  return (c, idx)
```

Figure 2. The *searchNode*(*pat*, *idx*, *node*, *startOff*, *step*) function for locating the smallest element in the passed node (the third parameter) referring to a suffix lexicographically not smaller than the pattern *pat*

This function makes use of *searchNode*(*pat*, *idx*, *node*, *startOff*, *step*) (Figure 2), which returns the smallest index of an element in the passed node (the third parameter) referring to a suffix not smaller than the pattern *pat*. The current index, *idx*, is updated only if a better candidate is found in the node. The parameter *startOff* stores the number of node elements which are skipped (as being lexicographically smaller than *pat*). The presented code for searchNode refers to the case of large nodes ($B > 8$), when a two-pass node lookup (the first pass with the step given as the last parameter of the function) is used.

Finally, the function count(*beg*, *end*) (Figure 3) returns the number of pattern occurrences in the range determined by the *beg* and *end* indexes. For simplicity, the presented code deals only with the case of $beg < end$, and $end < n$, i.e., when the *beg* index is located in a tree layer not lower than the layer of the index *end*. In the following paragraph we comment the main phases of this code.

In lines 01–02 we initialize the key variables, where res is the count to be eventually returned. The loop in lines 03–07 traverses down the tree until the layer just below the layer of the *end* index is reached. After the loop, the helper array *bOff* stores the left boundaries of the intervals from all the layers in which the elements from *beg* to *end* (inclusively) belong to. Lines 08–17 add the number of elements in the bottom layers of the tree, i.e., in the layers below the one to which *end* be-

count(beg, end)

```
(01)  res ← 0;   bOff[0] ← beg;   b ← beg;   l ← 1
(02)  bNode ← childNode(node(beg), elemOffset(beg))
(03)  while bNode < N do
(04)     b ← index(bNode) + k − 1
(05)     if b > end then break
(06)     bOff[l] ← b;   l ← l + 1
(07)     bNode ← childNode(node(beg), elemOffset(beg))
/* adding interval widths in tree layers below end index */
(08)  e ← end
(09)  eNode ← childNode(node(end), elemOffset(end))
(10)  while eNode < N do
(11)     e ← index(eNode) + k − 1
(12)     if e > n then break
(13)     res ← res + e − b
(14)     eNode ← childNode(eNode, k − 1)
(15)     bNode ← childNode(bNode, k − 1)
(16)     b ← index(bNode) + k − 1
(17)  if bNode ≤ N & b ≤ n then res ← res + n − b
/* adding interval widths in tree layers between beg and end indexes */
(18)  e ← end; eNode ← node(end)
(19)  while l > 0 do
(20)     l ← l − 1
(21)     res ← res + e − bOff[l]
(22)     eChild ← childNum(eNode);   eNode ← parent(eNode)
(23)     e ← index(eNode) + eChild
(24)  bNode ← node(bOff[0])
/* adding interval widths in tree layers above beg index */
(25)  if bNode = 0 then return res
(26)  while true do
(27)     bChild ← childNum(bNode);   bNode ← parent(bNode)
(28)     b ← index(bNode) + bChild
(29)     res ← res + e − b
(30)     if bNode = eNode then return res
(31)     eChild ← childNum(eNode);   eNode ← parent(eNode)
(32)     e ← index(eNode) + eChild
```

Figure 3. The function *count*(beg, end), which returns the number of pattern occurrences in the range determined by the *beg* and *end* indexes of the tree structure. It is assumed in the presented code that $beg < end$ and $end < n$ (handling the other cases is similar, but would make the pseudocode much longer).

longs. The variable b (resp. e) is set to the first element from (resp. beyond) the considered interval in the current layer. Special care must be taken not to exceed the last stored element (line 17). In lines 18–24 we handle the layers between the last considered layer (containing *end*) and the first considered layer (containing *beg*). Finally, an analogous procedure continues up to the top, terminating when b and e are in the same node (lines 30).

We have also a locate function, which is very similar to count, only instead of incrementing the counter of matching suffixes it adds them to a returned list.

2.3 Augmenting the Suffix Array

Manber and Myers in their seminal paper [1] presented a nice trick 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 to $\log_\sigma n$, where σ is the alphabet size, with $O(n \log n)$ extra bits of space and constant expected size of the interval. Unfortunately, real texts are far from random, hence in practice, we can use k up to 3 (assuming that text symbols are bytes), which offers a limited (yet, non-negligible) benefit. This idea will be referred in our experiments as using a lookup table, and more specifically we will denote the lookup table on pairs (resp. triples) of symbols with LUT2 (resp. LUT3).

In the same spirit, Grabowski and Raniszewski [15, 16] use a hash table to store the intervals for all k-symbol strings *occurring in the text*. This can significantly reduce the initial interval for real texts with relatively little extra space.

In this work we first propose a lookup table with keys being concatenations of Huffman codewords for the starting symbols of the text suffixes (Table 1), truncated to a specified length of b bits. Pattern search translates to finding the first b bits of Huffman encoding of the pattern, which is the LUT key, and then following with binary search over a range of suffixes read from the LUT. A look onto the rows, e.g., LUT-Huff-23b and LUT3, reveals that the resulting search intervals to go into are much narrower on average with the Huffman-based LUT, using the same amount of extra memory. A correct implementation of this idea, in combination with the B-tree SA layout, requires a reordering of the suffixes in the SA, to avoid nested LUT ranges (other options, like replacing Huffman with Hu-Tucker coding, are also possible, but we have not tried them out). Note also that the Huffman-based LUT entries store twice more data (both boundaries of the interval) than in the standard LUTs.

We also propose mixing the LUT or hash table interval narrowing with the B-tree layout, and also augmenting the search tree with prefixes of the suffixes in several top levels of the B-tree. Copying these text snippets into a helper array is beneficial due to more local memory accesses.

The last novelty is varying the parameter k, the length of the hashed strings. Using a fixed k results in having some intervals too wide (which deteriorates binary search) while some others are (too) narrow, which does not already help much.

space (MiB)	dna200	english200	proteins200	xml200
LUT-Huff-15b	0.25	14.45 ± 1.21	15.53 ± 2.20	12.86 ± 0.81	17.05 ± 2.84
LUT-Huff-19b	4.00	11.07 ± 1.52	13.46 ± 2.63	9.10 ± 1.31	15.91 ± 3.64
LUT-Huff-23b	64.00	7.79 ± 1.98	11.63 ± 2.93	5.63 ± 2.17	14.99 ± 4.28
LUT2	0.25	23.74 ± 0.46	19.50 ± 2.18	19.27 ± 0.94	18.91 ± 2.40
LUT3	64.00	21.82 ± 0.61	16.55 ± 2.78	15.11 ± 1.18	16.74 ± 3.49

Table 1. Average binary logarithms (with their standard deviations) of the search interval widths for different LUT variants (first three rows: order-0 Huffman encoding with 15–23 bits, next two rows: standard 2-/3-byte LUTs). The averages are taken over all suffixes of the text. Without a LUT the corresponding binary logarithms would be $\log_2(200 \cdot 2^{20}) = 27.64$ in all cases.

Varying k is expected to have more balanced interval widths, which in turn may translate into more preferable space-time tradeoffs.

To this end, we use three parameters, $k_0 < k_1 < k_2$, corresponding to suffixes' prefix lengths, and the parameter r as an interval width threshold. The first of the three parameters, k_0, is used in a standard lookup table and was fixed to 2 throughout the experiments. If a pair of successive symbols, c_1c_2, does not occur in the text more than r times (i.e., there are at most r suffixes starting with this prefix), the suffixes starting with c_1c_2 are not inserted into any other data structure; if the pattern matches such a prefix, one access to the mentioned LUT reveals that the range of suffixes to search is of size at most r and the binary search follows.

Those suffixes which do not fall into a narrow range according to their first two symbols, are then divided into two groups, based on whether their k_1-symbol prefix occurs at most r times in the text. Those for which the answer is positive are inserted into a hash table in the manner of the SA-hash index (see [16] for more details). If a k_1-long prefix occurs more than r times though, we extend its occurrences to length k_2 and insert such (distinct) strings into the same hash table. Additionally, a bit array V is maintained, initialized with zeros. For each distinct k_1-symbol string from the text its computed hash value tells the position in V to set a bit if its count is at most r. Collisions are not handled here, which means that several different strings may overwrite the same bit in V. Yet, V happens to be (relatively) very small for real texts, which allows for small load factors (e.g., LF = 0.1) and in turn translates into rather few collisions.

Now, given a pattern to search, we first check its k_1-symbol prefix in V. If the accessed bit is 1, we assume that the prefix, although relatively short, is specific enough. We look for it in the hash table and the associated data is the range of suffixes in which we continue with binary search. If the bit accessed in V is 0 though, we look for the longer prefix, of length k_2, in the hash table and continue in the same manner.

Let us yet justify the presented idea using a small example. For a given (fixed) k, we (locally) obtain three adjacent intervals of width 900, 60 and 40, respectively. The numbers of binary search steps are: 10, 6 and 6, respectively. Yet, the average

is not 7.333; it is rather $900/1\,000 \cdot 10 + 60/1\,000 \cdot 6 + 40/1\,000 \cdot 6 = 9.6$. This is because entering the widest interval is more likely than any of the remaining two. Now, we introduce k_1 and k_2, and it may happen that our considered intervals are split into three different intervals, of widths: 400, 500, 100, respectively. The corresponding numbers of binary search steps are now: 9, 9, 7, respectively. The (weighted) average is thus: $400/1\,000 \cdot 9 + 500/1\,000 \cdot 9 + 100/1\,000 \cdot 7 = 8.8$, i.e., yields some improvement.

3 EXPERIMENTAL RESULTS

All experiments were run on a machine equipped with a 4-core Intel i7 4790 3.6 GHz CPU and 32 GB of 1 600 MHz DDR3 RAM (9-9-9-24), hosting Windows Server 2012 R2. One CPU core was used for the computations. All codes (`https://bitbucket.org/kowallus/sa-search-dev/`) were written in C++ and compiled with 64-bit gcc 4.9.3 with `-O3`.

For each experiments, we took 500K patterns sampled from the text in a uniformly random manner, calculated the average time per pattern, repeated this procedure 11 times and presented the median. The searches were performed over 200 MB datasets from the well-known Pizza & Chili corpus (`http://pizzachili.dcc.uchile.cl/`).

In the first experiment we show how the count times are affected by two things: using lookup tables, including the introduced Huffman-based ones (on 15 or 23 bits) (Figure 5) and choosing a proper interval's right boundary search (Figure 4). HT denotes the idea of combining the suffix array with a hash table [15, 16]; it involves the parameter k, which is the length of suffixes' prefixes inserted in the hash table. In our experiments we (arbitrarily) choose the smallest k for which the overall size of the index, including the text, exceeds $5.5n$.

The doubling trick reduces the times usually by 20–30 % for the standard and LUT2-boosted suffix array, yet the effect is smaller for short patterns (i.e., small m), especially for DNA (where short patterns tend to have thousands of occurrences). This can be explained by the relatively small difference between $\log n$ and $\log occ$ in those cases. The Huffman-based LUTs are more efficient than their traditional counterparts (when about the same amount of memory is sacrificed).

For `xml200` (Figure 4) one can observe a different trend than for other datasets: the time decreases for smaller m. There are at least two factors specific to this dataset that cause such an effect. The first one is the extremely large (average) width of resulting intervals. For `english200`, the average interval width for $m = 9$ is about 10^3, while for `xml200` it is as much as 10^5. The difference in `xml200` is due to repetitiveness in the data (long XML tag names, etc.). This dataset is sensitive to m as long as the character access is involved; in datasets like `english200`, the average matching prefix length during the comparisons is not very sensitive to m (grows only slightly with growing m). The second factor is only a slight reduction of the resulting interval width with growing m. Note that most methods work faster on narrow

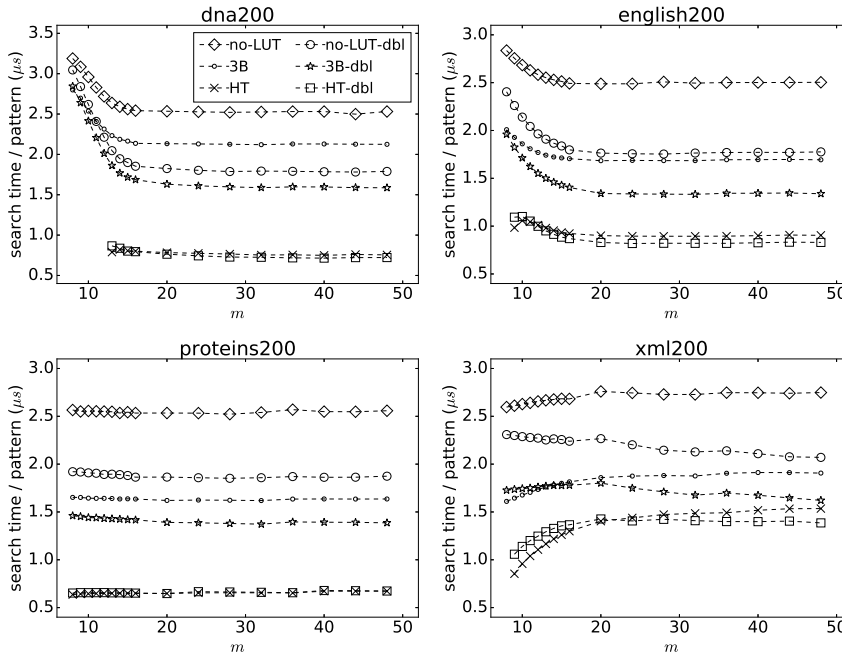

Figure 4. Count times for the standard suffix array and the SA augmented with a lookup table on triples of bytes and a hash table over k symbols ($k = 12$ for `dna200`, $k = 6$ for `proteins200` and $k = 8$ for the other datasets), using the standard and the doubling search for finding the right interval boundary. The standard SA size is $5n$ (bytes), the size with LUT3 is $5.321n$ and the size with a HT is $5.584n$, $5.882n$, $6.549n$ and $5.532n$, respectively, for `dna200`, `english200`, `proteins200` and `xml200`.

intervals, due to reduced time for finding the right boundary. Consider extending the pattern length from $m = 9$ to 10. For `english200`, the average interval length gets reduced to 56 %, while in `xml200` case to 83 % (in other words, the average interval for `xml200` shrinks by one sixth only, a really mild improvement). This effect occurs also in `proteins200`. Those two factors, taken together, may explain why for `xml200` using a smaller m may yield a shorter overall time, as opposed to other datasets. A similar reasoning also works for Figure 6.

Figure 5 shows how spending more space (from 2^{15} to 2^{25} array slots) for Huffman-based LUTs improves the count times. Apart from order-0 (i.e., context-free) Huffman coding also order-1 and order-2 Huffman models were used, to show that increasing the context order helps on compressible datasets (`english200`, `xml200`), but not on `dna200` and `proteins200` (comparable compression and more space used). Using, e.g., order-2 encoding means that the first input symbol is order-0 encoded, the second order-1 encoded and all the following ones are order-2 encoded. For `dna200`, biologically meaningful search patterns do not contain the

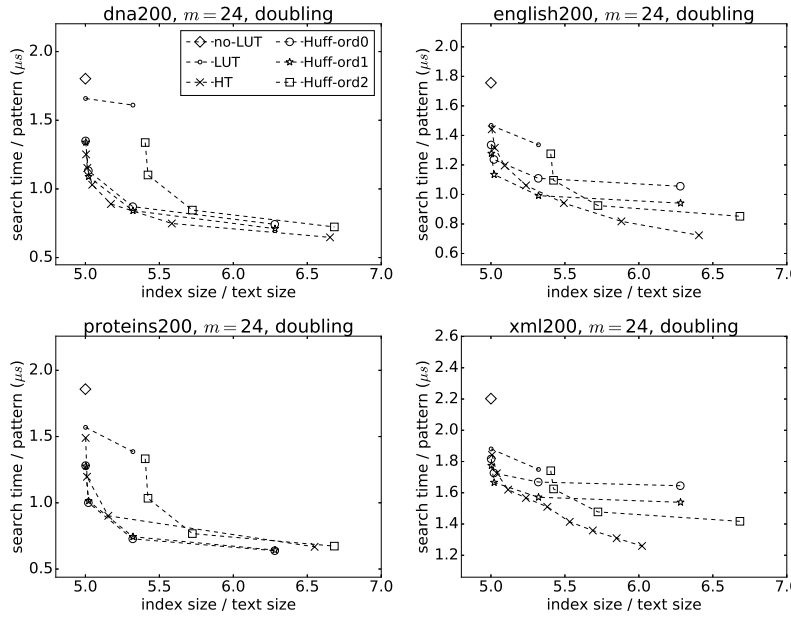

Figure 5. Count times for the standard suffix array, the SA augmented with a lookup table on pairs or triples of bytes (LUT), and on 15, 19, 23 or 25 bits of Huffman codewords (a series for a different coding order), and the hash table (HT) on varying k-grams from the text. The right interval boundary is found with the doubling technique.

N symbol and these should also be avoided in the searches with order-1 or order-2 Huffman-based LUT. The reason is that the N symbols tend to cluster, and N preceded by one or two Ns has high probability, which results in a one-bit Huffman codeword. In consequence, one of the inputs of our Huffman-based LUT can be a relatively long run of N symbols, which also means that the minimum pattern length should be relatively large (e.g., over 20), which is obviously undesirable.

Figure 6 shows the impact of the node size B in the B-tree layout on the count times, with varying pattern length. Even $B = 1$ results in a much faster search than with a standard SA (by a factor of 1.7–2.0; cf. also Figure 4) and growing B helps more, up to $B = 32$ (on all the datasets, $B = 64$ is slightly slower). Still, the speed gap between $B = 1$ and $B = 32$ rarely exceeds 10 %. This small improvement may seem disappointing, but is understandable. Using $B > 1$ improves access locality, concerning the suffix offsets, yet the accesses to the text are inevitable and those are generally not cached. This effects flattens all results. Moreover, the top levels of the tree (disregarding the choice of B) are cached across multiple patterns in the test collection.

The relative times of count and locate per found item, as a ratio of the respective results of the SA variant with the B-tree layout and the plain SA, are shown in

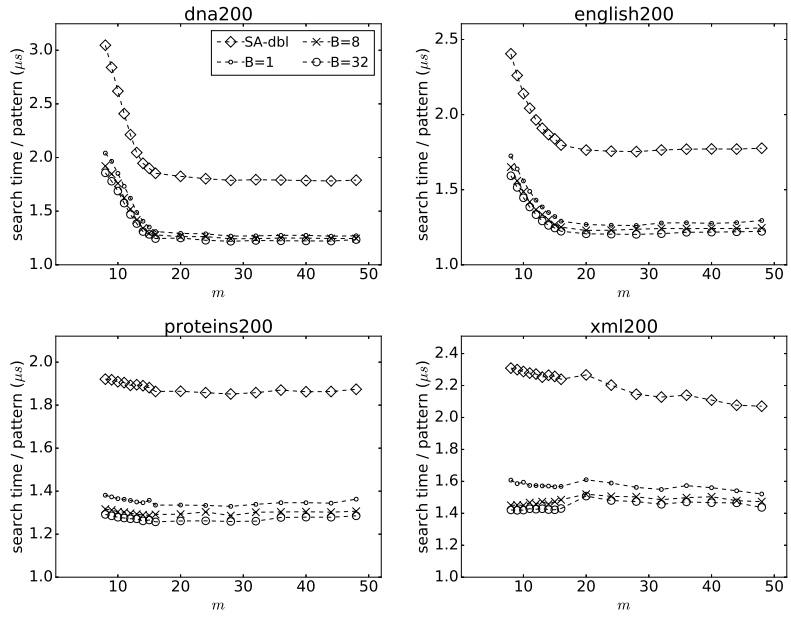

Figure 6. Count times for the standard SA and the SA with the B-tree layout, for selected node sizes B. The right interval boundary in the standard SA variant is found with the doubling technique (-dbl).

Figure 7. The locate operations in the B-tree SA variant suffer from the need to traverse over many layers of the tree, which is relatively more costly when the number of pattern occurrences tends to be larger (i.e., for small m). When m grows and *occ* is often a few (or even one), this overhead is relatively smaller and the cache-friendliness of the layout more than compensates the extra operations. The count operation, after the boundary suffixes are found, is also more costly in the B-tree layout, but the extra cost (logarithmic in N, as opposed to constant for the standard SA) is computational, without extra accesses (and thus resulting cache misses) to data.

In Figure 8 we show how augmenting the SA with various structures reducing the initial search interval affects the query times and the used space. The hash table (HT), based on the xxhash function (`https://github.com/Cyan4973/xxHash`), stores 8-grams from the text and was tried with two load factors (LF); the solution is called the SA-hash index in the original works [15, 16]. LUT2 gives a significant boost in a tiny space, yet it is the hash table (LF = 0.9) that excels here, speeding up the baseline variant by a factor of 1.5–2.

As our SAs with the B-tree layout can be augmented with prefixes of the suffixes visited in the first steps of the traversal of the tree (i.e., in the top levels), we test the impact of the prefix length on the performance and space of the resulting index

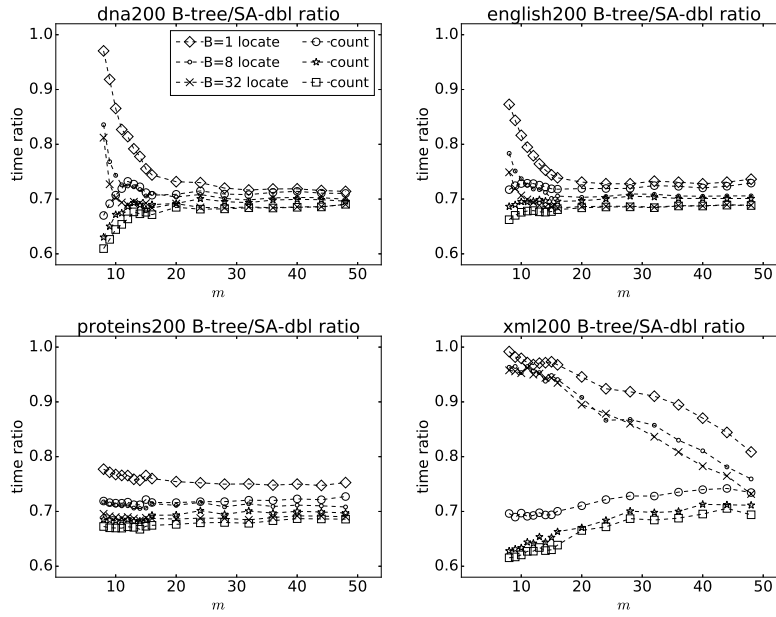

Figure 7. Count and locate relative times for the SA variant with the B-tree layout and the standard SA, for selected node sizes B

(Figure 9). Adding the prefixes gives a noticeable speedup even if they are limited in length to 8 characters and are attached to a few tree levels only, while the space overhead is rather small. Longer prefixes and more levels help less for a much bigger space penalty. Combinations of all ideas presented in the paper are shown in Figure 10, where the best option is to combine the B-tree layout with LUT/HT (adding prefixes on top of it has a negligible effect). In total, the speed of the standard SA with the standard right interval boundary search was usually improved by a factor exceeding 3 (from 2.6 for `xml200` to 3.9 for `dna200`, for $m = 24$).

In the last experiment we used the SA-hash index in a variant with varying the length of the hashed prefix. Different lines (space-time tradeoffs) are obtained with varying the parameters k_1 and k_2 (Figure 11). The key lines to compare are denoted as "SA-dbl & HT" (which is SA-hash with the doubling technique for finding the right interval boundary) and "SA-dbl & HT-var-k" (which is the new variant).

Table 2 presents some details, grouped in pairs of rows. The value of k (prefix length) used in the SA-hash algorithm is presented in the top rows. The parameters k_1 and k_2, and r, the interval width threshold, found in a learning procedure, are presented in the bottom rows. Clearly, r for the given dataset and the parameters k, k_1, k_2, is such to make the sizes of the compared structures possibly equal, as the next column demonstrates. More precisely, we find r in the following way. Let s_1

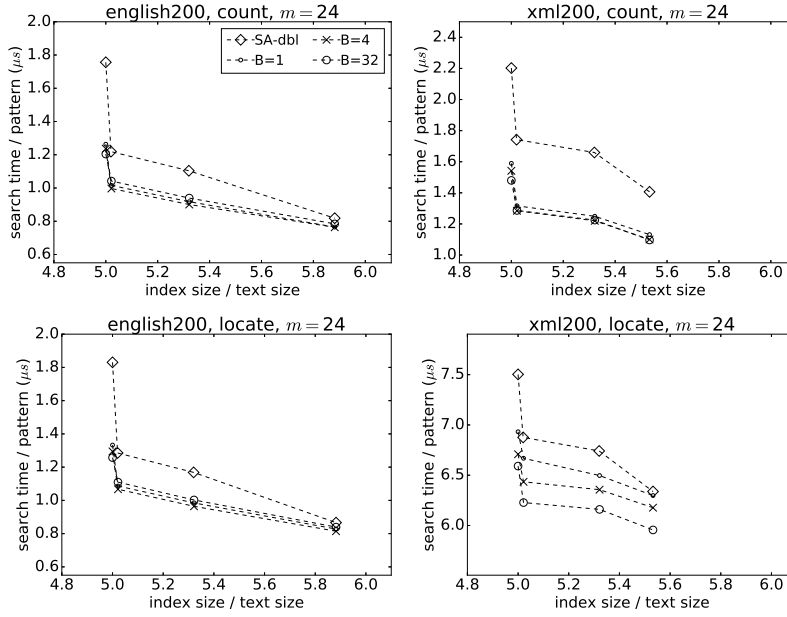

Figure 8. Augmenting the suffix array. The right interval boundary in the standard SA variant is found with the doubling technique (-dbl). The four points in each series correspond to: no extra data, LUT on 19 bit and 23 bits, respectively, using order-1 Huffman coding, and HT with LF = 0.9 and $k = 8$.

be the size of the SA-hash index for a given k. For fixed (arbitrarily chosen) values of k_1 and k_2, such that $k_1 < k$ and $k_2 > k$, we binary search for r in a way to obtain the size of the HT-var-k index not greater than s_1, but possibly close to it.

However, we see in Figure 11 that the net result of our efforts is mixed. In many cases the new variant is able to achieve a slight improvement in speed using

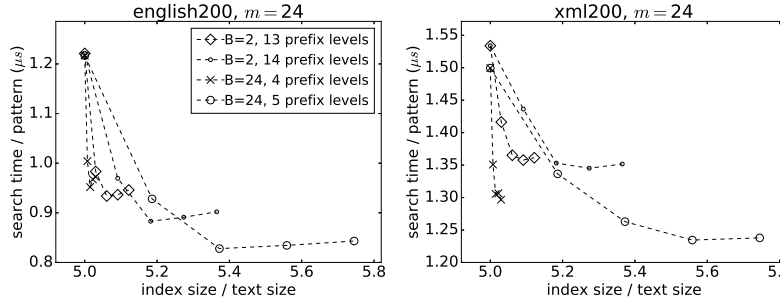

Figure 9. Index sizes and count times for the SA with the B-tree layout, when several top levels of the tree store the corresponding suffixes' prefixes of length $\{0, 4, 8, 12, 16\}$

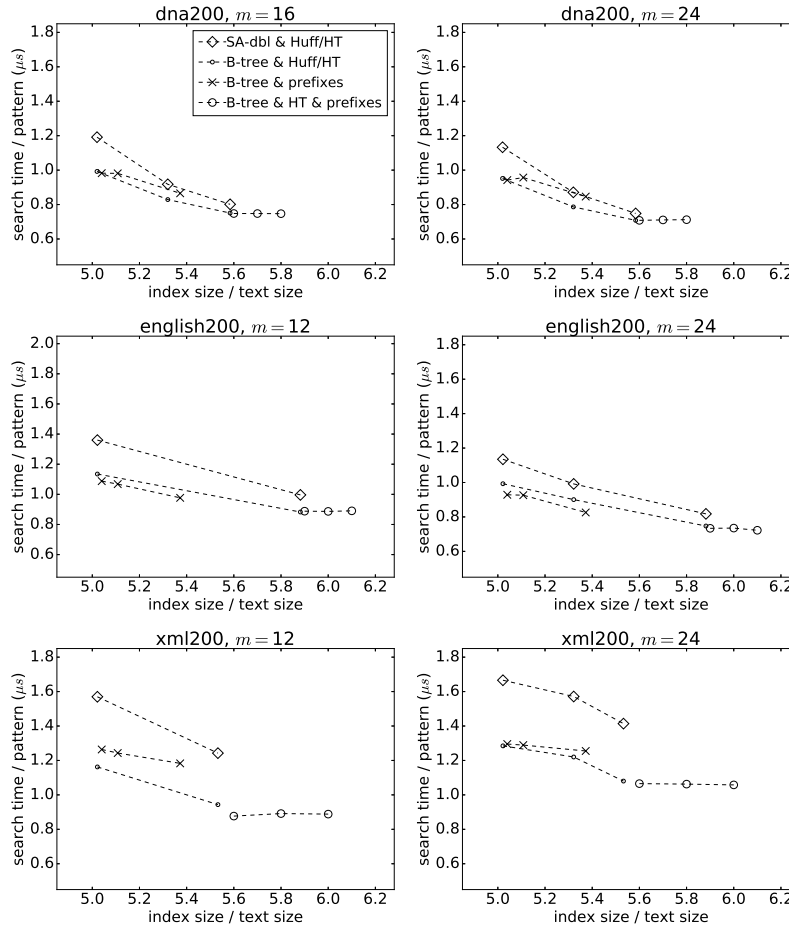

Figure 10. Index sizes and count times for several SA variants with different layouts and extra data. Successive points in the series are obtained by changing the LUT or hash table component and/or using the prefix copies on varying number of levels in the tree.

the same space, with (sometimes) queries faster by more than 10 % for the `xml200` dataset. On the `proteins200` dataset, however, using k_1 and k_2 instead of a single value of k makes the queries slower by more than 10 %.

4 CONCLUSION

Algorithm engineering not once revitalizes old ideas and data structures. In this work, we attempted to improve the performance of the classic full-text index, the suffix array. Our work focused on the navigation over the index, changes to its layout

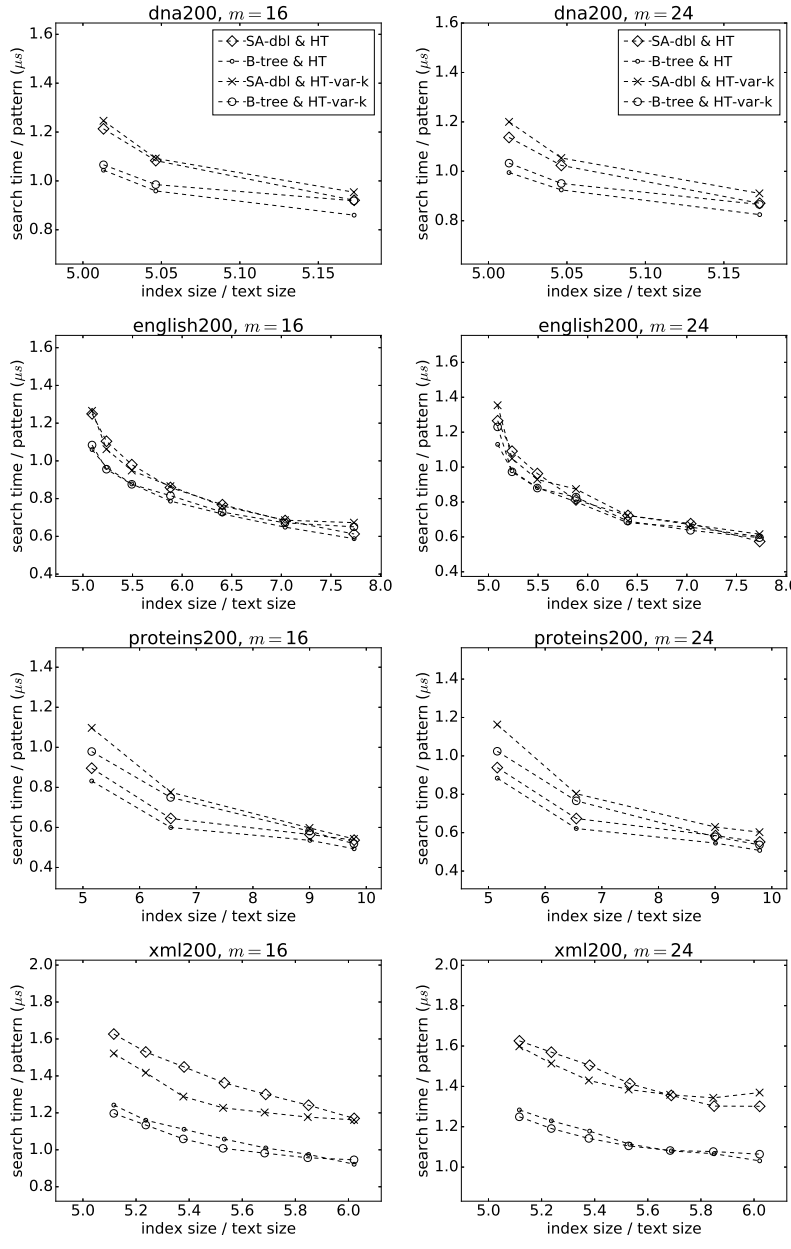

Figure 11. Index sizes and count times for the SA-hash index with two hashed prefix lengths (k_1 and k_2)

Dataset	k	k_1	k_2	r	Total Size	Filter Size	avg $\log_2$(ival_width)
dna200	12	–	–	–	5.584	–	5.256 ± 2.333
	–	10	16	964	5.584	0.006	7.601 ± 2.528
english200	8	–	–	–	5.882	–	7.045 ± 3.797
		6	12	3726	5.882	0.030	7.115 ± 3.695
proteins200	6	–	–	–	6.549	–	4.056 ± 2.801
		4	10	2989	6.549	0.001	7.802 ± 4.277
xml200	8	–	–	–	5.532	–	12.102 ± 5.942
		6	12	867	5.528	0.032	10.925 ± 5.539

Table 2. Index sizes for four 200 MB Pizza & Chili datasets. The used parameters are: k for the standard SA-hash index and k_1, k_2 for the SA-hash index with variable prefix lengths. The parameter r, the interval width threshold, was set in a way to have the two index sizes possibly close to each other. The rightmost column presents the average binary logarithms (with their standard deviations) of the search interval widths.

and augmenting the index with extra data. The experiments show that generally the best option is to combine a B-tree layout of the suffix with a lookup table or a hash table (reducing the interval of suffixes for further binary or k-ary search), with a speedup over the standard SA configuration by a factor usually exceeding 3, for a relatively small penalty in the space.

Acknowledgement

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

REFERENCES

[1] Manber, U.—Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing, Vol. 22, 1993, No. 5, pp. 935–948, doi: 10.1137/0222058.

[2] Ferragina, P.—González, R.—Navarro, G.—Venturini, R.: Compressed Text Indexes: From Theory to Practice. Journal of Experimental Algorithmics, Vol. 13, 2009, Art. No. 12, doi: 10.1145/1412228.1455268.

[3] Weiner, P.: Linear Pattern Matching Algorithms. Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory (swat 1973), Washington, DC, 1973, pp. 1–11, doi: 10.1109/swat.1973.13.

[4] Farach, M.: Optimal Suffix Tree Construction with Large Alphabets. Proceedings of the 38th IEEE Annual Symposium on Foundations of Computer Science (FOCS '97), 1997, pp. 137–143, doi: 10.1109/sfcs.1997.646102.

[5] Grimsmo, N.: On Performance and Cache Effects in Substring Indexes. Technical Report IDI-TR-2007-04, Norwegian University of Science and Technology, Trondheim, Norway, 2007.

[6] Cole, R.—Kopelowitz, T.—Lewenstein, M.: Suffix Trays and Suffix Trists: Structures for Faster Text Indexing. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (Eds.): Automata, Languages and Programming (ICALP 2006). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4051, 2006, pp. 358–369, doi: 10.1007/11786986_32.

[7] Fischer, J.—Gawrychowski, P.: Alphabet-Dependent String Searching with Wexponential Search Trees. In: Cicalese, F., Porat, E., Vaccaro, U. (Eds.): Combinatorial Pattern Matching (CPM 2015). Springer, Cham, Lecture Notes in Computer Science, Vol. 9133, 2015, pp. 160–171, doi: 10.1007/978-3-319-19929-0_14.

[8] Munro, J. I.—Navarro, G.—Nekrich, Y.: Fast Compressed Self-Indexes with Deterministic Linear-Time Construction. Proceedings of the 28th International Symposium on Algorithms and Computation (ISAAC 2017), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 92, 2017, Art. No. 57, 12 pp., doi: 10.4230/LIPIcs.ISAAC.2017.57.

[9] Bille, P.—Gørtz, I. L.—Skjoldjensen, F. R.: Deterministic Indexing for Packed Strings. Proceedings of the 28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 78, 2017, Art. No. 6, 11 pp., doi: 10.4230/LIPIcs.CPM.2017.6.

[10] Willard, D. E.: Searching Unindexed and Nonuniformly Generated Files in $\log \log N$ Time. SIAM Journal on Computing, Vol. 14, 1985, No. 4, pp. 1013–1029, doi: 10.1137/0214071.

[11] Zhou, J.–Ross, K. A.: Implementing Database Operations Using SIMD Instructions. Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02), ACM, 2002, pp. 145–156, doi: 10.1145/564691.564709.

[12] Schlegel, B.—Gemulla, R.—Lehner, W.: K-ary Search on Modern Processors. Proceedings of the Fifth International Workshop on Data Management on New Hardware (DaMoN '09), 2009, pp. 52–60, doi: 10.1145/1565694.1565705.

[13] Khuong, P.-V.—Morin, P.: Array Layouts for Comparison-Based Searching. Journal of Experimental Algorithmics, ACM, Vol. 22, 2017, No. 1, Art. No. 1.3, doi: 10.1145/3053370.

[14] Williams, J.: Algorithm 232: Heapsort. Communications of the ACM, Vol. 7, 1964, No. 6, pp. 347–348.

[15] Grabowski, S.—Raniszewski, M.: Two Simple Full-Text Indexes Based on the Suffix Array. In: Holub, J., Žďárek, J. (Eds.): Proceedings of the Prague Stringology Conference (PSC), 2014, pp. 179–191.

[16] Grabowski, S.—Raniszewski, M.: Compact and Hash Based Variants of the Suffix Array. Bulletin of the Polish Academy of Sciences – Technical Sciences, Vol. 65, 2017, No. 4, pp. 407–418, doi: 10.1515/bpasts-2017-0046.

Tomasz Marek Kowalski received his computer science M.Sc. and Ph.D. degrees from Łódź University of Technology in 2004 and 2008, respectively. His research interests involve object systems, computer languages and indexing techniques for various searching problems (string matching, bioinformatics, etc.). He has published over 60 papers in journals and conferences. He is currently Assistant Professor at the Institute of Applied Computer Science of the Łódź University of Technology.

Szymon Grabowski received his M.Sc. degree from the University of Łódź in 1996, Ph.D. degree from the AGH University of Science and Technology in Cracow in 2003, and the habilitation degree at the Systems Research Institute of the Polish Academy of Sciences in Warsaw in 2011. His former research, including Ph.D. dissertation, involved nearest neighbor classification methods in pattern recognition, also with applications in image processing. Currently, his main interests are focused on string matching and text indexing algorithms, and data compression. Some of his particular research topics include various approximate string matching problems, compressed text indexes, and XML compression. He has published over 120 papers in journals and conferences. He is currently Professor at the Institute of Applied Computer Science of Łódź University of Technology.

Kimmo Fredriksson received his computer science M.Sc. and Ph.D. degrees from University of Helsinki in 1997 and 2001, respectively. His research interests include wide variety of string matching problems as well as indexing techniques for searching in metric spaces. He has published about 60 papers on these topics in international conferences and journals. He has the position of Adjunct Professor at the University of Eastern Finland.