A Bloom filter based semi-index on $q$-grams

Szymon Grabowski*,†, Robert Susik and Marcin Raniszewski

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

SUMMARY

We present a simple $q$-gram based semi-index, which allows to look for a pattern typically only in a small fraction of text blocks. Several space-time tradeoffs are presented. Experiments on Pizza & Chili datasets show that our solution is up to three orders of magnitude faster than the Claude et al. (Journal of Discrete Algorithms 2012; 11:37) semi-index at a comparable space usage. Moreover, the construction of our data structure is fast and easily parallelizable. Copyright © 2016 John Wiley & Sons, Ltd.

Received 15 February 2016; Revised 4 July 2016; Accepted 9 July 2016

KEY WORDS: text indexing; $q$-grams; Bloom filter; minimizers

1. INTRODUCTION

String matching algorithms have traditionally been divided into on-line and off-line ones. This distinction is not always satisfactory. On-line solutions often lack in performance, if run over large texts, while off-line (index) schemes are complicated and/or resource-hungry. In [1], an intermediate approach was called a semi-index. A semi-index is a data structure built on top of a given text, which is easy and fast to construct, uses a reasonable amount of memory (both during construction and its lifetime) and allows to search for a pattern faster than using an on-line scan (albeit typically not as fast as with a ‘real’ index).

The semi-index of Claude et al. [1] replaces the original text with a shorter one, namely such that some symbols of the alphabet are omitted. The same symbols are also removed from the pattern before the search and potential matches have to be verified. Clearly, the search speed is usually improved if a bigger part of the alphabet is sampled out, yet the problem is that in an extreme case the whole pattern may be ‘erased’. Nonetheless, for long patterns, of length 100, a speedup in online search by factor about 5 while using 14% extra space was reported. For moderately long patterns, of length 20, the speedup was less than twofold. Several others algorithms from the literature can also be classified as semi-indexes, in particular $q$-gram based inverted files [2].

In this paper, we present a simple semi-index on $q$-grams, being faster than the Claude et al. solution by up to three orders of magnitude at a comparable space usage. Although standard compact full-text indexes (usually based on suffix sorting) are even faster in the search, they have a more costly construction stage. On the other hand, the block-based nature of our data structure allows for an efficient and easily parallelizable construction. (We note in passing that block indexing is an old approach in information retrieval, in particular used in word-based indexes [3, 4].) As our algorithm involves the Bloom filter, in the next paragraph we briefly recall this classical data structure.

A Bloom filter (BF) [5] is a succinct probabilistic membership data structure, in which false positive matches are possible, but false negatives never occur. The structure is merely a bit array, initially zeroed, to which elements are added by writing bits 1 into $u$ locations pointed by $u$ different hash functions. An element is considered to occur in the BF (which may be untrue) if all $u$ hash
functions applied to it point to set bits in the array. Bloom filters work best if the number of elements to add is known beforehand. In such case, for the chosen array size, we can set the parameter \( u \) in a way minimizing the expected error rate.

We use the standard notation throughout the paper. The pattern \( P[0 \ldots m - 1] \) is sought over the text \( T[0 \ldots n - 1] \). Both strings are composed of symbols from a common integer alphabet \( \Sigma = \{0, 1, \ldots, \sigma - 1\} \).

2. RELATED WORK

2.1. On using \( q \)-grams and Bloom filters for string matching

The notion of \( q \)-grams is widely used in string matching algorithms for more than two decades. In 1992, Ukkonen [6] introduced the \( q \)-gram distance between two strings and used it for (online) approximate string matching. Takaoka [7] presented an approximate matching filter based on sampling \( q \)-grams from the text. This technique was refined by Sutinen and Tarhio [8] with using ordered \( q \)-grams, based on a simple observation that the preserved \( q \)-grams must be approximately at the same locations both in the pattern and its approximate match in the text. Burkhardt and Kärkkäinen [9] advocated for gapped \( q \)-grams, proving their superior filtering capabilities. The concept of \( q \)-gram indexing, useful for approximate string matching, was initiated by Navarro et al. [10].

On the other hand, Bloom filters were also frequently chosen as a component in string matchers and indexes, including anti-virus scanners [11] intrusion-detection systems [12], distributed storage [13] and processing [14].

Perhaps the most important application area for string matching algorithms is bioinformatics. Indexes on \( q \)-grams have been applied for searching biological sequence databases in several solutions. In particular, QUASAR [15] and Swift [16] basically divide the text into small blocks and only the blocks having at least a specified fraction of \( q \)-grams in common with the query sequence are processed carefully, for example, with BLAST, to report alignments. In some solutions, for example, BLAT [17], the index contains only the information about the non-overlapping \( q \)-grams, resulting in a reduction in the index size but a loss in sensitivity. Both Bloom filters and \( q \)-grams were used by Chikhi and Risk [18] for a succinct representation of de Bruijn graphs.

2.2. Alphabet sampling

The main idea of the Claude et al. [1] approach, mentioned in Section 1, is to choose a subset of the alphabet to be the sampled alphabet and then to build a subsequence of the text by omitting all characters not in the sampled alphabet. In the search phase an analogous sampling is applied to the pattern. Additionally, at regular intervals of the sampled text the corresponding positions in the original text are stored. This mapping is used in the verification phase, which follows the main search phase.

For example, if the text is \( T = abaaacabdaa \) and the sampled alphabet is \( \Sigma_X = \{b, c, d\} \), then the sampled text is \( T_X = t_1t_4t_6t_7 = cbcd \). The pattern \( P = acab \) is correspondingly translated to \( P_X = p_1p_3 = cb \). The algorithm searches for \( P_X \) in \( T_X \) (the popular Boyer–Moore–Horspool algorithm [19] was chosen for that purpose) and uses the extra mapping data to find the positions of \( P \) in \( T \).

Two mapping ideas are considered in the Claude et al. solution. The first one uses an array \( M \) storing the positions in the original text of symbols which belong to \( \Sigma_X \) where every \( d \)-th sampled character is mapped to its position in the original text. The other mapping uses a bitmap \( B \) of length \( n \) where \( B[j] = 1 \) iff \( T[j] \in \Sigma_X \) for every \( j \). The bitmap is used in two variants; in the more compact one, called succinct alphabet mapping, two texts are stored: \( T_X \) and \( T_Y \), containing the characters in \( \Sigma_X \) and \( \Sigma_Y = \Sigma - \Sigma_X \), respectively, together with \( B \) mapping, what allows for efficient representation of \( T \).

Apart from the data structure representation issues, the authors proposed two variants of their algorithm. The variant Basic always uses the sampled text for the search. The variant Estimated Best Text estimates the cost of searching using the sampled text or the original text and chooses the text with smaller estimated cost.
3. OUR ALGORITHM

In this section, we propose Bloom Filter based Semi-Index (BFSI), an algorithm combining highly selective filtering of text blocks before the actual search (text scan) with simplicity, both on the conceptual and implementational level.

The text $T$ is partitioned into $n/b$ fixed-size blocks of $b$ symbols$^2$, and successive blocks are grouped into superblocks of size $r$ blocks. The overlapping $q$-grams of each $i$th superblock are added to a Bloom filter [5], represented as one bit table $B_i$ of size $cbr$ bits, where $c$ is the chosen number of bits per item in a BF, trading its size for accuracy. Let us introduce a set of $u$ baseline hash functions, $h_k(S): \sigma^q \rightarrow \{0, 1, \ldots, cb-1\}$, for $k \in \{0, \ldots, u-1\}$, where $S$ is a string of length $q$. The actual hash functions applied to elements from block $ir + j$, for any $0 \leq i \leq (n/(br))$ and $0 \leq j < r$, are however of the form $h'_k(S) = h_k(S) \cdot r + j$. Note that the hashes for $q$-grams from $j$th block within $i$th superblock may affect only such $B_i[\ell]$ cells that $\ell \mod r = j$. In other words, within a superblock the $q$-grams for each block are as if stored in a separate BF, but these $r$ substructures are interwoven; the motivation for choosing such a layout will be given later.

The search idea for the pattern $P[0 \ldots m - 1]$, where $m \geq q$, is very simple; if we cannot exclude that all its $q$-grams occur (in any location) in a given block, then the block is skipped using some “off-the-shelf” exact string matching algorithm. In the opposite case, when we are sure that at least one $q$-gram of $P$ has no occurrence in the block, the block is skipped. To this end, the search starts with computing $u$ hash values for each of the $(m - q + 1)$ overlapping $q$-grams of $P$. To simplify the exposition, let us present the following procedure on a single $q$-gram of the pattern, for example, the first one, $P[0 \ldots q - 1]$. Its computed hashes are of the form $h_k(P[0 \ldots q - 1])$, for $k \in \{0, \ldots, u-1\}$. We traverse over successive superblocks, and in each superblock we need to check in which of the $r$ blocks the $q$-gram occurs. To this end, we calculate $h'_k(P[0 \ldots q - 1]) = h_k(P[0 \ldots q - 1]) \cdot r + j$, for all valid $k$ and $j$, and the set bits in the found positions denote the $q$-grams containing the current $q$-gram. If a given block contains a set bit for all the $q$-grams from the pattern, we have to scan the block for occurrences of $P$ (as the actual pattern matching algorithm for it, we chose FAOSO [20, 21]). If not, we proceed to the next block. Yet for the next block the accessed cells of $B_i$ are simply successors of the corresponding cells accessed in the previous block (because we replace term $j$ with $j + 1$). In this way, we reduce the number of needed memory accesses, which justifies our data layout in $B_i$. We set $r$ to 64 (which is the machine word size in our test platform). Figure 1 illustrates.

Longer $q$-grams should be more selective than shorter ones, yet choosing too large $q$ prevents searching for short patterns. Another observation is that we may trade the BFSI space for filtering selectivity with building the BF with a sample of $q$-grams only. To this end, we consider the following variants:

- BFSI-STD, the standard version of BFSI, with all $q$-grams from the block used,
- BFSI-SAM (sampled $q$-grams), which inserts every $s$th, $s \geq 1$, sampled $q$-gram to a BF; note that $s = 1$ corresponds to the BFSI-STD variant,
- BFSI-MSAM (minimizer-based sampled $q$-grams), which is similar to BFSI-SAM, but samples the $q$-grams in a non-regular way, using the idea of minimizers [22]. More concretely, we slide a window of size $w$ over a block and for each window, we find the lexicographically smallest substring of length $p$ starting in the window. Note that successive text windows are likely to share the same minimizer. The starting symbols of all distinct minimizers in a block are also the starting positions of the sampled $q$-grams. It is easy to notice that the gap between two successive sampled $q$-grams cannot exceed $w$.

Let us estimate the probability that a given block will be checked in the BFSI-STD variant, assuming no error introduced by the Bloom filter. A pattern contains $m - q + 1$ (overlapping) $q$-grams. We assume that $m = o(b)$, where $b$ is the block length (in characters). The probability that

$^2$Actually, we force the blocks to have an overlap of 128 symbols. Each occurrence of a pattern of length up to 128 may then be found twice, yet those redundant occurrences are easily filtered out in the postprocessing. Patterns longer than 128 are truncated to 128 symbols before the actual search and their tentative locations in the text are verified.
Figure 1. Searching of a pattern q-gram in a Bloom filter (BF) in two variants: (a) standard searching in each BF block causes a lot of memory accesses, (b) searching in a larger BF structure created by interwoven BF blocks, which (c) reduces the number of memory accesses thanks to the changed data layout. Note that r is set to 4 in this example, while in practice its value should be related to the machine word size.

a given q-gram occurs in the given block is $O(b/\sigma^q)$. Let us consider a q-gram $P[i \ldots i + q - 1]$, where $i + q < m$. The probability that a given q-gram $P[i + j \ldots i + j + q - 1]$, for some $0 < j < m - i - q$, occurs in the block, given that $P[i \ldots i + q - 1]$ occurs in the block, is the sum of the two probabilities: the probability that $P[i + j \ldots i + j + q - 1]$ overlaps $P[i \ldots i + q - 1]$ in the block, and the probability that $P[i + j \ldots i + j + q - 1]$ does not overlap $P[i \ldots i + q - 1]$. The first case can be split into two subcases: $P[i + j \ldots i + j + q - 1]$ is located on the right of the occurrence of $P[i \ldots i + q - 1]$ in the block, and the opposite subcase. The latter subcase has a negligible probability. Let us focus on the subcase where a proper prefix of $P[i \ldots i + j + q - 1]$ is a suffix of $P[i \ldots i + q - 1]$ in the block. The sought probability is $O \left(1/\sigma + 1/\sigma^2 + \ldots + 1/\sigma^{q-1}\right) = O(1/(\sigma - 1))$. The second case, in which $P[i + j \ldots i + j + q - 1]$ does not overlap $P[i \ldots i + q - 1]$, has a probability of $O(b/\sigma^q)$, remembering that $q < m = o(b)$. As any of the two terms, $b/\sigma^q$ or $1/(\sigma - 1)$, can be asymptotically larger, we bound the sought (conditional) probability with $O \left(b/\sigma^q + 1/(\sigma - 1)\right)$. 

Copyright © 2016 John Wiley & Sons, Ltd.

Softw. Pract. Exper. 2017; 47:799–811
DOI: 10.1002/spe
By the same reasoning applied to all the \( q \)-grams, we obtain the final probability that all \( q \)-grams from the pattern occur in a given block is

\[
O \left( \left( \frac{b}{\sigma^q} \right) \left( \frac{b}{\sigma^q} + 1/\left( \sigma - 1 \right) \right)^m \right).
\]

Note that the upper bound of \( b/\sigma^q \) on the probability of the occurrence of a given \( q \)-gram in the block may be vastly imprecise (e.g., for a small alphabet), hence we can replace this term with \( 1 - \left( \frac{\sigma - 1}{\sigma^q} \right)^{b-q+1} \). The final formula then becomes

\[
O \left( \left( 1 - \frac{\sigma - 1}{\sigma^q} \right)^{b-q+1} \right) \left( 1 - \left( \frac{\sigma - 1}{\sigma^q} \right)^{b-q+1} + 1/\left( \sigma - 1 \right)^m \right).
\]

4. EXPERIMENTAL RESULTS

In order to evaluate the performance of BFSI, we ran quite extensive experiments. As the competitor, we took the algorithm from Claude et al. [1], whose source codes were received from the authors. The test machine was equipped with an Intel i7-4930K 3.4 GHz CPU and 64 GB of DDR3 RAM, running Ubuntu 15.10 LTS 64-bit. The RAM modules were 8 \( \times \) 8 GB DDR3-1600 with the timings 11-11-11 (Kingston KVR16N11D4K4/64). The CPU cache sizes were: 6 \( \times \) 32 KB (data) and 6 \( \times \) 32 KB (instructions) in the L1 level, 6 \( \times \) 256 KB in L2 and 12 MB in L3. One CPU core was used for the computations. All codes were written in C++ and compiled with g++ 5.2.1 and -O3 flag. Our software is available at https://github.com/rsusik/bfsi/. As the datasets, we took five 200 MB texts from the widely used Pizza & Chili corpus (http://pizzachili.dcc.uchile.cl/). In most test cases, we search 10,000 random patterns and present the average timings. (The exceptions are the Claude et al. [1] semi-index variants, for which we used 100 random patterns, otherwise the complete tests would last too long.) BFSI tests were ran with varying multiple parameters. We can distinguish between parameters common for all variants (BFSI-STD, BFSI-SAM, and BFSI-MSAM), parameters specific for particular variants (BFSI-SAM and BFSI-MSAM) and parameters of the chosen search algorithm (which was FAOSO in all tests). The standard variant, BFSI-STD, uses only the
common parameters: $q$-gram size ($q$), block size ($b$) and Bloom filter density ($c$), which affects the number of hash functions and the expected false positive rate. For the hash functions, we used the algorithm Murmur3 (https://github.com/aappleby/smhasher) and generate all $u$ hash functions using a linear combination of only two hash functions. According to the finding of Kirsch and Mitzenmacher [23], this allows for an implementation of a Bloom filter without a loss in the asymptotic false positive probability, given two pairwise-independent universal hash functions for the input.

The parameter specific to BFSI-SAM is $s$, which is the $q$-gram sampling rate (i.e., every $s$-th $q$-gram is selected). The BFSI-MSAM variant makes use of two specific parameters: window size ($w$) and minimizer length ($p$).

We set the necessary requirements: $m \geq q + s - 1$ for the BFSI-SAM variant and $m \geq w + \max(q, p) - 1$ for the BFSI-MSAM variant. As said earlier, in all variants we use the FAOSO exact pattern matching to scan the selected text blocks. FAOSO depends on two parameters which were fixed ($U = 4$ and $k = 2$). It might be possible to achieve better performance results by tuning the FAOSO parameters for particular datasets or use another pattern matcher instead.

Figure 2 presents a comparison of our solution and the Claude et al. semi-index [1, Sect. 2.2 and Sect. 3.1] (more information on the particular variants can be found in the cited work) on english.
A BLOOM FILTER BASED SEMI-INDEX ON \( q \)-GRAMS

Figure 4. Search speed (left) and the relative amount of the scanned text (right) by BFSI-STD for varying block size \( b \) and fixed \( c = 6, q = 8 \) and \( m = 32 \). For each series, the points read from right to left correspond to the block sizes \( b \) of 8, 16, 32, 64, 128 (KiB).

Figure 5. Search speed (left) and the relative amount of the scanned text (right) by BFSI-SAM for varying \( s \) and fixed \( b = 8 \) (KiB), \( c = 6, q = 8 \) and \( m = 32 \). For each series, the points read from right to left correspond to the sampling parameter \( s \) of 1, 2, 3, 4, 5, 6, 7, and 8.

and dna datasets and patterns of length \( m \in \{16, 32\} \). Specifically, we show the results of the following variants of the Claude et al. index:

- Basic, map., \( d = 32 \) (using an array; every 32nd sampled character is mapped to its position in the original text)\(^6\);
- Est., map., \( d = 32 \) (Estimated Best Text, using an array; every 32nd sampled character is mapped to its position in the original text),
- Est., succinct (Estimated Best Text, using a bit-vector and a succinct representation of \( T \)).

We tested the listed versions with \( h \in \{1, 2, 4, 8, 12, 16, 20\} \) for english and \( h \in \{1, 2, 4\} \) for dna, on the same patterns and datasets as our solution, where \( h = |\Sigma - \Sigma_X| \), that is, the number of rejected symbols in the alphabet sampling. Actually, we tested also other variants of the Claude et al. solution, ‘Basic, bitmap’ and ‘Est., bitmap’, but they were not competitive due to using somewhat more space at comparable search speeds and thus are not presented in the figure.

In this comparison, we used all three BFSI variants, with the parameter values of \( q \in \{4, 6, 8, 10\} \), \( c = 6 \) and \( b = 8 \) (KiB) or \( b = 128 \) (KiB), the latter only for BFSI-STD. For BFSI-SAM, we used the parameter \( s = 4 \) and for BFSI-MSAM the parameters \( p = 2 \) and \( w = 4 \). We show the space used by the index (in addition to the text itself), as a fraction of the text size, and the search speed in GB/s. Our solution in the best case (for relatively large space of the semi-index) is about three orders of magnitude faster than the Claude et al. semi-index at the same space. When the space is limited, BFSI-STD is only a few times faster than the corresponding Claude et al. variants, yet the BFSI-SAM and BFSI-MSAM variants are much better choice then, up to two orders of magnitude. As

\(^6\)In the original paper, the sampling parameter was denoted with \( q \); we changed it to \( d \), to prevent confusing it with the \( q \)-gram length, the parameter frequently used in the experiments with our algorithms.
expected, choosing a larger $q$ for BFSI variants makes the search faster yet at the cost of requiring more space. Note that the ‘Est., succinct’ variant is represented by a single point, as its space use is indifferent to the sampling parameter $h$. We simply chose the best speed out of several results (choosing another $h$ may result in a speed drop of up to 10%–29%, depending on the dataset and the pattern length $m$).

Figure 3 presents performance comparison in function of the pattern length $m$. The plots show that speed generally grows with increasing $m$ and that this speed increase tends to be directly proportional to the reduction of the fraction of the text blocks to be scanned. In some cases (english, proteins), however, we can observe a slow-down at the longest patterns ($m = 64$). This can be explained by the fact that $q$-gram checks in the Bloom filters are not free and will eventually be dominating, as the number of scanned blocks already for $m = 32$ is very small.

Figure 4 presents the speed, the fraction of the text to scan and the space in function of the block size $b$. As expected, with growing blocks the index gets smaller and the speed decreases. One should be careful however with setting the block size parameter, because a too small block may also hamper the performance. This is the case of $b = 8$ for english and can be explained with relatively large costs of handling the Bloom filters. Note on the right figure that the fraction of the text to scan for the english dataset is very low generally, hence reducing it further with a change in the block size.
Table I. Speed [GB/s], time [ms] and average number of verified blocks in searching 100 patterns ($m = 8$ or $m = 32$) over a uniformly random text ($\sigma = 20$).

| text length | $m = 8$ |                       | $m = 32$ |                       |
|-------------|---------|-----------------------|----------|-----------------------|
|             | speed [GB/s] | verified blocks [%] | time [ms] | speed [GB/s] | verified blocks [%] | time [ms] |
| 10 MB       | 0.353   | 75.91 (0.30%)         | 29.77    | 5.675   | 1.00 (0.00391%)     | 1.85     |
| 20 MB       | 0.363   | 150.04 (0.59%)        | 57.80    | 8.914   | 1.00 (0.00391%)     | 2.35     |
| 50 MB       | 0.348   | 380.29 (1.49%)        | 150.88   | 13.449  | 1.01 (0.00395%)     | 3.90     |
| 100 MB      | 0.352   | 748.71 (2.92%)        | 297.81   | 18.832  | 1.01 (0.00395%)     | 5.56     |
| 150 MB      | 0.351   | 1128.05 (4.41%)       | 448.75   | 22.064  | 1.00 (0.00391%)     | 7.13     |
| 200 MB      | 0.341   | 1433.11 (5.60%)       | 614.41   | 23.318  | 1.00 (0.00391%)     | 8.99     |

Figure 7. Speed/space relation for fixed $b = 8$ (KiB), $c = 6$, $q = 8$, $p = 2$ and $m = 32$. Note the scale on the Y axis is logarithmic for dna and linear in the remaining cases.
Figure 8. Locate speeds for patterns of varying length.

from, for example, 16 KiB to 8 KiB, has a minor overall effect, yet doubling the number of blocks makes the number of checks in the Bloom filters doubled, which can yield a noticeable penalty.

Figure 5 shows the impact of the sampling parameter $s$ on the search speed, index size and the fraction of the text to scan for the BFSI-SAM variant. The chosen parameters are: $m = 32$, $c = 6$, $b = 8$ (KiB) and $q = 8$. Larger $s$ can significantly reduce the index size, but the performance is reduced as well (although not much when $s$ is not greater than 4 or 5).

In Figure 6, the varying parameter is the $q$-gram size. As expected, using a larger $q$ is generally helpful for speed, at the cost of larger space requirements. This space usage increase is more significant for BFSI-STD than for the variants with sampled $q$-grams. For the dblp.xml dataset, setting $q > 6$ is counterproductive, with the exception of the BFSI-SAM variant. On two datasets, english and proteins, we can observe that sampling the $q$-grams allows to obtain both higher searching speed and lower space usage than those offered by the baseline variant, BFSI-STD.

The text size is another factor that impacts the performance of our algorithms. To evaluate this impact, we use datasets of growing lengths (10 MB, 20 MB, 50 MB, 100 MB, 150 MB, and 200 MB), being prefixes of a 200 MB uniformly random text over an alphabet of size $\sigma = 20$. The results for BFSI-STD, averaged over 100 patterns chosen randomly from the used text, are presented in Table I. The fixed parameters were: $q = 8$, $c = 6$, $b = 8$ (KiB), while the pattern length $m$ was set either to 8, or to 32. We can notice that for short patterns the number of blocks that need to be scanned
grows rather linearly with the growing text size. On the other hand, long patterns (selected from the text) tend to occur only once, and a large value of $q$ usually prevents from yielding multiple 'false' blocks to verify. This implies that the speed in the former case is roughly constant while in the latter case it grows with the text size.

Figure 7 presents a performance comparison of the BFSI-SAM and BFSI-MSAM variants for varying $s$ (BFSI-SAM) and $w$ (BFSI-MSAM) parameters on multiple datasets. We executed BFSI-SAM with $s = \{2, 3, 4, 5, 6\}$ and then ran BFSI-MSAM with the $w$ parameter selected from $\{2, 3, 4, 5, 6, 7, 8, 9, 10, 12\}$ in a way to obtain the index size possibly close to the corresponding BFSI-SAM index size. The parameter $p$ for BFSI-MSAM was set to two in all cases. All the other parameters are common for both variants. Clearly, BFSI-SAM dominates on $\text{dblp.xml}$, $\text{dna}$ and $\text{sources}$, while the situation comes different for $\text{proteins}$, with mixed results on $\text{english}$. BFSI-MSAM's performance is often more vulnerable to shifting its parameter ($w$) than BFSI-SAM to shifting $s$.

Finally, we performed a preliminary comparison of BFSI and a few 'standard' full-text indexes: FM-index with rank V5 [24], that is, a variant trading space for more speed, sparse suffix array [25] and sampled suffix array on minimizers (SamSAMi) [26]. The latter two indexes contain a subset of the suffix offsets stored in the classic suffix array [27], and are generally less efficient for short patterns, where the extra verifications may be a bottleneck. The FM index code was taken from the well-known sdsl-lite library (https://github.com/simongog/sdsl-lite) while the other implementations are ours (https://github.com/mranisz/). The task was to report the locations of all the pattern occurrences, that is, the locate query. In Figure 8 each series contains the search speeds for patterns of length $m = \{8, 16, 32, 64\}$ and the corresponding space usages are given in Table II. Clearly, our

| dataset   | BFSI-STD | FM-uncompr. (V5) | Sparse SA | SamSAMi2 |
|-----------|----------|------------------|-----------|----------|
| dna       | 1.651    | 1.796            | 1.667     | 2.517    |
| english   | 1.690    | 1.670            | 1.667     | 2.117    |
| proteins  | 1.632    | 1.617            | 1.667     | 2.184    |
| sources   | 1.413    | 1.355            | 1.667     | 2.380    |
| dblp.xml  | 1.309    | 1.331            | 1.667     | 2.276    |

Table II. Index sizes, as multiples of the text size. The BFSI-STD parameters are just like the ones used in Figure 3. The parameters of the other algorithms was chosen in order to have similar space requirements. More precisely, the FM-index was run with the parameters $s_{SA} = 4$ and $s_{ISA} = 64$ for $\text{dna}$, $\text{english}$ and $\text{proteins}$, and $s_{SA} = 8$ and $s_{ISA} = 128$ for the remaining datasets. The sparse suffix array was run with the sampling step 6 and SamSAMi2 with $q = 6$ and $p = 1$. For the algorithms Sparse SA and SamSAMi2, it was not always possible to set the space very close to the one of BFSI-STD, as the minimal pattern length handled then would be higher than 8. Note that the space of the original text is included in the presented space figures, with the exception of the FM-index which is a self-index and replaces the text rather than builds on it.

| dataset    | 1    | 2    | 4    | 6    | 12   |
|------------|------|------|------|------|------|
| dna        | 6.4  | 12.5 | 18.1 | 25.4 | 36.8 |
| english    | 5.4  | 9.6  | 13.2 | 17.2 | 20.8 |
| proteins   | 5.4  | 9.5  | 12.9 | 16.7 | 19.8 |
| sources    | 6.1  | 11.2 | 15.9 | 21.5 | 28.1 |
| dblp.xml   | 6.4  | 11.9 | 17.0 | 23.4 | 31.3 |

Table III. Construction speed [MB/s] for the variants BFSI-STD and BFSI-SAM ($s = 4$) for an increasing number of threads. The other parameters, common for both variants: $b = 8$ (KiB), $c = 6$, $q = 6$. 

| dataset    | # of threads |
|------------|-------------|
| dna        | 6.4 12.5 18.1 25.4 36.8 |
| english    | 5.4 9.6 13.2 17.2 20.8 |
| proteins   | 5.4 9.5 12.9 16.7 19.8 |
| sources    | 6.1 11.2 15.9 21.5 28.1 |
| dblp.xml   | 6.4 11.9 17.0 23.4 31.3 |

Copyright © 2016 John Wiley & Sons, Ltd.  Softw. Pract. Exper. 2017; 47:799–811 DOI: 10.1002/spe
approach cannot be competitive to other indexes for long patterns, where the number of matches is typically very small, yet for \( m = 8 \) it is sometimes faster than the FM-index in a comparable space.

We also measured the construction time of our algorithm (BFSI-STD and BFSI-SAM variants), in a parallel implementation, using from 1 to 12 threads on our 6-core (with hyper-threading) machine. They are shown in Table III. As expected, the construction is significantly faster for BFSI-SAM (as fewer items are inserted into a Bloom filter) and may even reach above 100 MB/s, which, to our knowledge, far exceeds the performance of the best FM or suffix array construction algorithms [28–30] on a similar hardware.

5. CONCLUSIONS

The concept of a semi-index, a lightweight data structure combining high performance with conceptual and implementational simplicity, deserves more attention than it has so far received in the literature. We presented a semi-index working on a block basis and involving a Bloom filter applied to text \( q \)-grams. Experiments show that this simple concept allows to search for a pattern up to about three orders of magnitude faster than with an online scan for the cost of extra space less than 70% of the original text size. Moreover, the block nature of our data structure allows for its efficient (semi-)external and parallel construction, a feature which seems unavailable for solutions based on the suffix array or the FM-index.

ACKNOWLEDGEMENTS

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

REFERENCES

1. Claude F, Navarro G, Peltola H, Salmela L, Tarhio J. String matching with alphabet sampling. Journal of Discrete Algorithms 2012; 11:37–50.
2. Puglisi SJ, Smyth WF, Turpin A. Inverted files versus suffix arrays for locating patterns in primary memory. In String Processing and Information Retrieval (SPIRE), vol. 4209, Crestani F, Ferragina P, Sanderson M (eds.), LNCS. Springer: Glasgow, UK, 2006; 122–133.
3. Manber U, Wu S. GLIMPSE: A tool to search through entire file systems. In USENIX Winter 1994 Technical Conference, USENIX Association: San Francisco, California, 1994; 23–32.
4. Baeza-Yates R, Navarro G. Block-addressing indices for approximate text retrieval. Journal of the American Society for Information Science (JASIS) 2000; 51(1):69–82.
5. Bloom BH. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 1970; 13(7):422–426.
6. Ukkonen E. Approximate string-matching with \( q \)-grams and maximal matches. Theoretical Computer Science 1992; 92(1):191–211.
7. Takaoka T. Approximate pattern matching with samples. In Algorithms and Computation (ISAAC), vol. 834, Du D, Zhang X (eds.), LNCS. Springer: Beijing, China, 1994; 234–242.
8. Sutinen E, Tarhio J. On using \( q \)-gram locations in approximate string matching. In European Symposium on Algorithms (ESA), vol. 979, Spirakis P (ed.), LNCS. Springer: Corfu, Greece, 1995; 327–340.
9. Burkhardt S, Kärkkäinen J. Better filtering with gapped \( q \)-grams. In Combinatorial Pattern Matching (CPM), vol. 2089, Amir A, Landau GM (eds.), LNCS, 2001; 73–85.
10. Navarro G, Baeza-Yates R, Sutinen E, Tarhio J. Indexing methods for approximate string matching. IEEE Data Engineering Bulletin 2001; 24(4):19–27.
11. Erdogan O, Cao P. Hash-AV: fast virus signature scanning by cache-resident filters. International Journal of Security and Networks 2007; 2(1-2):50–59.
12. Kandhan R, Teletia N, Patel JM. Simmatch: Fast and scalable multi-pattern matching. PVLDB 2010; 3(1):1173–1184.
13. Wang Z, Luo C, Luo T, Chen X, Hou J. A Bloom filter-based index for distributed storage systems. In 12th International Conference Distributed Computing and Artificial Intelligence (DCAI 2015), Salamanca, Spain, 2015; 293–301.
14. Crainiceanu A. Bloom: a hierarchical Bloom filter index with applications to distributed data provenance. In 2nd International Workshop on Cloud Intelligence ( colocated with VLDB 2013), Cloud-I ’13: Riva del Garda, Trento, Italy, 2013; 4:1–4:8.
15. Burkhardt S, Crauser A, Ferragina P, Lenhof H, Rivals E, Vingron M. \( q \)-gram based database searching using a suffix array. In Research in Computational Molecular Biology (RECOMB); Lyon, France, 1999; 77–83.
16. Rasmussen KR, Stoye J, Myers EW. Efficient q-gram filters for finding all e-matches over a given length. Journal of Computational Biology 2006; 13(2):296–308.
17. Kent WJ. BLAT—the BLAST–like alignment tool. Genome Research 2002; 12(4):656–664.
18. Chikhi R, Rizk G. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. Algorithms for Molecular Biology 2013; 8(1):1–22.
19. Horspool RN. Practical fast searching in strings. Software: Practice and Experience 1980; 10(6):501–506.
20. Fredriksson K, Grabowski S. Practical and optimal string matching. In String Processing and Information Retrieval (SPIRE), vol. 3772, Consens MP, Navarro G (eds.), LNCS. Springer: Buenos Aires, Argentina, 2005; 376–387.
21. Fredriksson K, Grabowski S. Average-optimal string matching. Journal of Discrete Algorithms 2009; 7(4):579–594.
22. Roberts M, Hayes W, Hunt BR, Mount SM, Yorke JA. Reducing storage requirements for biological sequence comparison. Bioinformatics 2004; 20(18):3363–3369.
23. Kirsch A, Mitzenmacher M. Less hashing, same performance: Building a better bloom filter. Random Structures and Algorithms 2008; 33(2):187–218.
24. Gog S, Petri M. Optimized succinct data structures for massive data. Software–Practice and Experience 2014; 44(11):1287–1314. DOI: 10.1002/spe.2198.
25. Kärkkäinen J, Ukkonen E. Sparse suffix trees. In Computing and Combinatorics Conference (COCOON), vol. 1090, Cai J, Wong CK (eds.), LNCS: Hong Kong, 1996; 219–230.
26. Grabowski S, Raniszewski M. Sampling the suffix array with minimizers. In String Processing and Information Retrieval (SPIRE), vol. 9309, Iliopoulos CS, Puglisi SJ, Yilmaz E (eds.), LNCS. Springer: London, UK, 2015; 287–298.
27. Manber U, Myers G. Suffix arrays: a new method for on-line string searches. In Symposium on Discrete Algorithms (SODA). SIAM: San Francisco, California, 1990: 319–327.
28. Wang L, Baxter S, Owens JD. Fast parallel suffix array on the GPU. In Parallel and distributed computing (Euro-Par), vol. 9233, Träff JL, Hunold S, Versaci F (eds.), LNCS, Springer, 2015; 573–587.
29. Baier U. Linear-time suffix sorting. Master Thesis, Ulm University, Ulm, Germany, 2015.
30. Labeit J, Shun J, Blelloch GE. Parallel lightweight wavelet tree, suffix array and FM-index construction. In Data Compression Conference (DCC). IEEE Computer Society: Snowbird, Utah, 2016; 33–42.