Byte-Aligned Pattern Matching in Encoded Genomic Sequences

Petr Procházka[1] **and Jan Holub**[2]

1 **Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic**
`Petr.Prochazka@fit.cvut.cz`

2 **Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic**

Abstract

In this article, we propose a novel pattern matching algorithm, called `BAPM`, that performs searching in the encoded genomic sequences. The algorithm works at the level of single bytes and it achieves sublinear performance on average. The preprocessing phase of the algorithm is linear with respect to the size of the searched pattern m. A simple $\mathcal{O}(m)$-space data structure is used to store all factors (with a defined length) of the searched pattern. These factors are later searched during the searching phase which ensures sublinear time on average. Our algorithm significantly overcomes the state-of-the-art pattern matching algorithms in the locate time on middle and long patterns. Furthermore, it is able to cooperate very easily with the block q-gram inverted index. The block q-gram inverted index together with our pattern matching algorithm achieve superior results in terms of locate time to the current index data structures for less frequent patterns. We present experimental results using real genomic data. These results prove efficiency of our algorithm.

1998 ACM Subject Classification F.2.2 Pattern Matching

Keywords and phrases genomic sequences, pattern matching, q-gram inverted index

Digital Object Identifier 10.4230/LIPIcs.WABI.2017.20

1 Introduction

DNA sequencing is nowadays the integral part of several disciplines like personalized medicine, biology, biotechnology, or forensic biology. The demand for cheap sequencing induced the evolution of High-Throughput Sequencing (HTS) technologies that can sequence large stretches of DNA in a massively parallel fashion and that produce millions of DNA sequences simultaneously. The public sources report the necessary time per one run in the order of hours and the cost per one million bases lower than 0.02 USD[1]. General availability of the sequencing causes producing large volumes of genomic data that needs to be stored effectively in the form allowing extremely fast searching.

DNA molecule can be mapped one-to-one to a sequence of letters which implies that it can be processed as a text string. The string matching problem is crucial task since early beginnings of the text processing. The task is very simple – to find all occurrences of a given pattern P in a large text T. However, this task is performed very frequently and over large volumes of data (text T) which implies that very fast algorithms are necessary. To accelerate

[1] `https://www.genome.gov/sequencingcostsdata/`

the string matching, the algorithm can preprocess either the pattern or the text or both. The pattern preprocessing is relevant only for the given pattern and therefore it is included in the search process. The text preprocessing (which includes especially the indexing methods) is universal for all possible patterns, however it usually requires some extra space to store an auxiliary data structure.

Knuth-Morris-Pratt (KMP) [11] is one of the most famous pattern matching algorithms and the first one ensuring the worst-case time linear with the length of the text T. Boyer-Moore (BM) [3] family algorithms represent backward pattern matching approach. BM algorithm allows skipping of some characters which leads to lower than linear average time. There exist also other variations of this algorithm given by Horspool [10] or Sunday [20]. Suffix automaton (often called DAWG – Deterministic Acyclic Word Graph) is the essence of another algorithm achieving sublinear average time BDM (Backward DAWG Match) [4]. The suffix automaton of the reversed pattern performs backward searching for the pattern. The byproduct of the search is always the longest prefix of the pattern occurring at that position in the text which ensures safe shifting for BDM. Another approach is to use non-deterministic instead of deterministic automata for searching in the text. So-called *bit-parallelism* [5, 2] proved to be a very simple way how to simulate the non-deterministic automaton. It exploits the parallelism provided by bitwise operations in terms of one computer word. It can accelerate the operations up to a factor w, where w is the number of bits in the computer word. Bit-parallelism is particularly efficient for the patterns with size lower than the size of the computer word $m \leq w$. Navarro et al. applied the bit-parallelism to simulate the suffix automaton and they proposed BNDM algorithm [17] that achieved 20%-25% improvement in search time in comparison to its deterministic version BDM. Later, Durian et al. [22] proposed an efficiency improvement of BNDM and Shift-Or algorithm residing in processing q-grams of the input symbols. BSDM [7] is relatively recent algorithm using suffix automaton searching for a factor with no repetitions of a condensed pattern. BSDM proved to be very fast especially for middle-sized and longer patterns. Very recently, the algorithms (e.g., [6, 21]) exploiting SIMD (Single Instruction, Multiple Data) instructions of modern CPUs appeared. EPSM [6] tabulates all the factors (of a given length) of the searched pattern. The factors are easy to access using hash table whereas the hash function is provided by CRC SIMD specialized instruction. The algorithm performs searching for any of the factors (in the filtering phase) and any of the hits must be confirmed by direct check at the corresponding position in the text.

We propose a novel pattern matching algorithm, called `BAPM` (Byte-Aligned Pattern Matching). Our algorithm is optimized for searching in the encoded genomic sequences only. It exploits the low alphabet size of the genomic sequences which implies a possibility to tabulate all factors (of a given length) of the pattern achieving reasonable memory consumption. Furthermore, a simple encoding scheme for genomic sequences allows to process the factors as a sequence of one or more bytes. The searching phase of `BAPM` resides in shifting over the encoded DNA sequence, reading a sequence of one or more bytes and comparing its value with the tabulated factors of the searched pattern. When a factor is found the potential occurrence of the searched pattern must be still confirmed by direct comparison of the text to the pattern. `BAPM` works at the level of bytes all the time. Only the very last step confirming a potential occurrence applies bitwise operations. This leads to high efficiency in searching for middle-sized and long patterns.

Preprocessing of the input text is another way how to speed up searching in the text. Suffix trees [23] are one of the fundamental index structures. Suffix array [14] is another basic index structure that significantly improves demanding space requirements of the suffix trees.

Other indexes like FM-index [8] or CSA (Compressed Suffix Array) [9] further improved the space requirements of the index data structure to be the same or lower than the size of the input text. A separate branch of research focused on indexing text files with natural language content. In this field, so-called inverted index [15] is considered as de-facto standard. However, the inverted index proved its efficiency also when performed on other kind of data [18]. We supplemented our `BAPM` with a simple block q-gram inverted index and experimentally compared its locate speed with state-of-the-art index data structures.

The rest of the paper is organized as follows. We give definitions of some basic notions in Section 2. The Section 3 is dedicated to definition and detailed description of `BAPM` algorithm and necessary data structures. The Section 4 summarizes experimental results performed on real genomic data. We give the conclusion and some ideas for future work in Section 5.

2 Basic Notions

Let $x = x_1x_2..x_n$ be a *string* composed of single symbols x_i of a finite ordered alphabet Σ. The length of the string x is $n = |x|$. The size of the alphabet Σ is $\sigma = |\Sigma| = \mathcal{O}(1)$. The start position i and the length j define so-called *factor* (or *substring*) denoted by $x_{i,j} = x_i..x_{i+j-1}$. A factor with $i = 0$ is called *prefix* and a factor with $i + j - 1 = n$ is called *suffix* of the string x. We denote by ε so-called empty string of length 0. The problem of string pattern matching is to find all occurrences of a pattern $P = p_1p_2..p_m$ in a text $T = t_1t_2..t_n$ where both strings are composed of symbols from the same alphabet Σ and $m \ll n$. Particularly, we can distinguish two tasks: (i) *count* when number of occurrences of P in T is reported and (ii) *locate* when exact positions of the occurrences of P in T are reported.

Pattern substitution method [13] is a compression method when q-grams of symbols of the input text T (i.e., Σ^q) are substituted with an assigned byte value b where $b \in \{0, 1, \ldots, 255\}$. The pattern matching on the compressed (encoded) text means to find all occurrences of the compressed pattern P_C in the compressed text T_C (both defined over the alphabet of byte values $b \in \{0, 1, \ldots, 255\}$).

Traditional *inverted index* consists of two major components: a vocabulary storing all distinct words occurring in the text T and a set of *posting lists* storing positions of all occurrences of a given word in the text T. The vocabulary of a q-gram inverted index [18] is composed of all possible q-grams of the alphabet Σ, i.e., Σ^q. For the purpose of *block indexing* we split the indexed text into single blocks of a defined fixed size. The posting lists of a *block inverted index* then store addresses of the blocks covering the exact positions of occurrences. The exact positions are determined in the next step when a standard pattern matching method is performed in terms of the preselected blocks.

In later description of the algorithm, we use C-like syntax for bitwise operations. Particularly, we use $|$ for bitwise-or, & for bitwise-and, $\ll$ for shift-left operation and $\gg$ for shift-right operation.

3 Byte-Aligned Pattern Matching

Byte-Aligned Pattern Matching algorithm (`BAPM`) is optimized for searching in the encoded genomic sequences. It assumes the input alphabet $\Sigma = \{A, C, G, T\}$ and a simple substitution encoding defined as $f : \Sigma^4 \mapsto B$ where $B = \{0, 1, \ldots, 255\}$ and $b \in B$ represents a byte value that is composed as a concatenation of bit couples given by the single symbols of the 4-gram $s \in \Sigma^4$ ($A \to 00$, $C \to 01$, $G \to 10$, $T \to 11$). The algorithm detects single occurrences within two steps. In the first step, the algorithm performs searching for all factors of a defined fixed

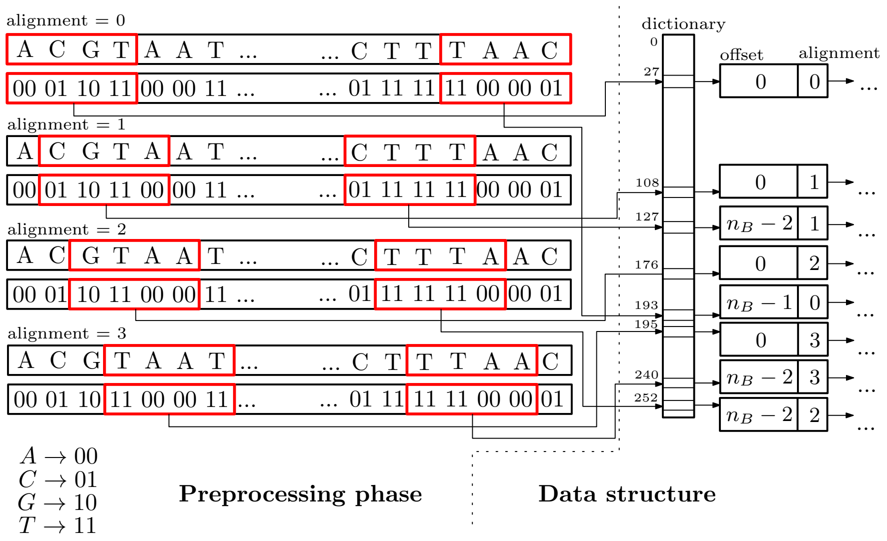

Figure 1 BAPM: Preprocessing phase. The length of the encoded pattern P_C is n_B bytes.

length that must be a multiple of 4 (in terms of the input alphabet Σ). This ensures that each encoded factor is represented as a sequence of one or more bytes. BAPM tabulates all possible factors of the encoded pattern during the preprocessing phase. It is reasonable to require the set of all factors to fit into the memory cache. For this reason, the acceptable lengths of the factors are 4 and 8 bases/symbols (in terms of the input alphabet Σ) which implies the length of one or two bytes, respectively for the encoded factors. The first step of the searching is only the filtering of possible occurrences. A potential occurrence must be always confirmed using direct comparison of the encoded pattern P_C with the encoded text T_C at a given position i. We have implemented two versions of BAMP tabulating 4-gram factors (BAPM4) and 8-gram factors (BAPM8), respectively. We explain all the principles of the algorithm using the version with 4-gram factors. However, the same principles are valid for the version with 8-gram factors as well. From now on, BAPM reports to 4-gram version of the algorithm if not stated other way.

Figure 1 depicts a simple data structure used to store the tabulated encoded factors of the pattern and it demonstrates also BAPM preprocessing phase when this data structure is filled. The dictionary data structure is depicted in the right part of the figure and its main part is an array with 256 entries (corresponding to 256 different byte values). Every entry can contain a pointer to a list which stores all occurrences of the factor (corresponding to the entry) in terms of the pattern. Each element of the list is a couple (offset, alignment). The offset o represents a byte position of the factor in the encoded pattern and it is easy deducible from its starting position i in the raw pattern $o = \lfloor \frac{i-1}{4} \rfloor$. The alignment a represents a position of the factor in terms of the byte and it can be computed as $a = (i-1) \mod 4$. Suppose constant size of the computer word. Then, it is obvious that the dictionary requires $\mathcal{O}(m)$ space where m is a size of the raw pattern. The space of the array is constant and the lists contain together $m-3$ elements, each of them consuming $\mathcal{O}(1)$ space. The offset o requires $\mathcal{O}(\log \frac{m}{4})$, however, we suppose it can be encoded within a single computer word in all practical cases.

The left part of Figure 1 describes single steps of the preprocessing phase of the algorithm. For every possible alignment $a \in \{0,1,2,3\}$, consecutive byte values of the shifted encoded pattern P_C are processed. The value of the byte determines its position in the dictionary.

For every byte, its offset and alignment are stored to the corresponding list pointed from the dictionary. The remaining bases at the end of the shifted pattern that do not compose a complete byte are omitted (e.g., the suffix AAC for alignment = 1 in Figure 1).

Suppose the length of the raw pattern $m = 128$ bases which implies 32 bytes for the encoded pattern P_C. `BAPM4` needs to store $128 - 3 = 125$ factors (elements of the lists). Suppose a simple byte code used to store the offset o and the alignment a for every factor. Only two bytes are consumed for every pair (o, a) and still all the information is encoded at the level of bytes. Thus, for `BAPM4` the total space is $(128 - 3) \times 2 + 256 = 506$ bytes, plus some overhead needed to implement the lists. Still, the data structure easily fits into 2 kiB of memory and it can be kept in the top level of the computer cache. For `BAPM8`, we can estimate the needed space as $(128 - 7) \times 2 + 256 \times 256 = 65\,778$ bytes, plus the overhead for the lists. This is still acceptable space ensuring storing the data structure in the fast levels of the computer memory.

Algorithm 1 describes preprocessing and searching phase of `BAPM4`. The function ENCODE is called in the preprocessing phase. The function performs the simple substitution encoding described above. Its parameters are: the text to be encoded; the starting index for encoding; and the number of bases/symbols that need to be encoded. The function returns desired encoded factor of the text. The function BUILDDICTIONARY is responsible for constituting dictionary D and storing the shifted versions of the pattern in the array B. The **`while`** cycle (line 4) iterates over all possible alignments $a \in \{0, 1, 2, 3\}$. For every alignment, the number of bases/symbols that constitute the longest byte sequence starting at i is computed (line 5) and the corresponding encoded pattern is obtained (line 6). The encoded pattern is stored for the given alignment (line 7) and later is used for direct comparison of bytes (the encoded text with the encoded pattern). Next **`while`** cycle (line 9) iterates over all bytes of the encoded aligned pattern and it ensures storing the couples (offset, alignment) to their corresponding lists (line 12).

The function BUILDMASK is another part of the preprocessing. It generates all necessary masks possibly needed in the last step of the comparison (a prefix and/or a suffix of the encoded pattern with the corresponding part of the encoded text). Since the prefix and the suffix are smaller than one byte the masks are necessary to minimize the bitwise operations and therefore also the needed time. The function stores the masks in single variables. The variable *pref* stores a prefix of the encoded pattern (smaller than one byte) for all possible alignments $a \in \{0, 1, 2, 3\}$. The variable *suf* stores a suffix of the encoded pattern (smaller than one byte) for all possible alignments $a \in \{0, 1, 2, 3\}$. The examples of the stored prefixes and suffixes can be seen in Figure 1 as the symbols preceding/following the red rectangles. Similarly, the variables *prefMask* and *sufMask* store the masks (used for bitwise-and operation with the corresponding byte in the encoded text) needed to compare a prefix or the suffix, respectively of the encoded pattern. The **`while`** cycle (line 22) iterates over only three possible alignments. The *pref* value is stored for the alignments $a \in \{1, 2, 3\}$ (starting from the value 3). The prefix for the alignment $a = 0$ is an empty string ε and therefore it is not stored. The pointer to the array of the suffix values *suf* is shifted by the value *la* and it starts from the position $(la + i) \mod 4$. In every step of the cycle, the algorithm: (i) stores the corresponding prefix value to *pref* array and the corresponding prefix mask to *prefMask* array; (ii) stores the corresponding suffix value to *suf* array and the corresponding suffix mask to *sufMask* array; (iii) shifts auxiliary variables *prefM* and *prefB* two bits right; (iv) shifts auxiliary variables *sufM* and *sufB* two bits left.

The function SEARCH represents the main function of `BAPM`. After the preprocessing of the searched pattern (lines 31 and 32) the algorithm states a safe shift as the number of

Algorithm 1 `BAPM4` preprocessing and searching phase

```
 1: function buildDictionary($P$, $m$)
 2:     $D \leftarrow \emptyset$; $B \leftarrow \emptyset$;
 3:     $i \leftarrow 0$;
 4:     while $i \leq 3$ do
 5:         $b \leftarrow \lfloor (m-i)/4 \rfloor \times 4$;
 6:         $E \leftarrow$ encode($P, i+1, b$);
 7:         $B_i \leftarrow E$;
 8:         $j \leftarrow 1$;
 9:         while $j \leq \lfloor (m-i)/4 \rfloor$ do
10:             if $D_{E_j} = \emptyset$ then
11:                 $D_{E_j} \leftarrow$ create a new list storing offsets and alignments;
12:             add a couple of offset and alignment $(j-1, i)$ to the list $D_{E_j}$;
13:             $j \leftarrow j+1$;
14:         $i \leftarrow i+1$;

15: function buildMask($P$, $m$)
16:     $\mathit{pref} \leftarrow \emptyset$; $\mathit{prefMask} \leftarrow \emptyset$; $\mathit{prefM} \leftarrow$ 0x3f;
17:     $\mathit{suf} \leftarrow \emptyset$; $\mathit{sufMask} \leftarrow \emptyset$; $\mathit{sufM} \leftarrow$ 0xfc;
18:     $\mathit{prefB} \leftarrow$ encode($P, 1, 4$) $\gg 2$;
19:     $\mathit{sufB} \leftarrow$ encode($P, m-4, 4$) $\ll 2$;
20:     $\mathit{la} \leftarrow m \bmod 4$;
21:     $i \leftarrow 1$;
22:     while $i \leq 3$ do
23:         $\mathit{prefMask}_{4-i} \leftarrow \mathit{prefM}$;
24:         $\mathit{sufMask}_{(\mathit{la}+i) \bmod 4} \leftarrow \mathit{sufM}$;
25:         $\mathit{pref}_{4-i} \leftarrow \mathit{prefB}$;
26:         $\mathit{suf}_{(\mathit{la}+i) \bmod 4} \leftarrow \mathit{sufB}$;
27:         $\mathit{prefM} \leftarrow \mathit{prefM} \gg 2$; $\mathit{prefB} \leftarrow \mathit{prefB} \gg 2$;
28:         $\mathit{sufM} \leftarrow \mathit{sufM} \ll 2$; $\mathit{sufB} \leftarrow \mathit{sufB} \ll 2$;
29:         $i \leftarrow i+1$;

30: function search($T$, $n$, $P$, $m$)
31:     buildDictionary($P, m$);
32:     buildMask($P, m$);
33:     $\mathit{shift} \leftarrow \lfloor m/4 \rfloor - 1$;
34:     $i \leftarrow \mathit{shift}$;
35:     while $i \leq n$ do
36:         if $D_{T_i} \neq \emptyset$ then
37:             for each couple of offset and alignment $(o, a) \in D_{T_i}$ do
38:                 $r \leftarrow$ compare all bytes starting from $T_{i-o}$ with $B_a$;
39:                 if $r = 0$ & $a \neq 0$ then
40:                     $r \leftarrow$ compare $(T_{i-o-1}$ & $\mathit{prefMask}_a)$ with $\mathit{pref}_a$;
41:                 if $r = 0$ & $a \neq \mathit{la}$ then
42:                     $r \leftarrow$ compare $(T_{i-o+\mathit{shift}}$ & $\mathit{sufMask}_a)$ with $\mathit{suf}_a$;
43:                 if $r = 0$ then
44:                     report an occurrence at position $4 \times (i-o) - a + 1$;
45:         $i \leftarrow i + \mathit{shift}$;
```

whole bytes of the encoded pattern minus one (line 33). The `while` cycle (line 35) traverses the encoded text T of length n. It always reads a byte value T_i and the corresponding entry in the dictionary D_{T_i} is checked (line 36). If the dictionary entry D_{T_i} is empty the algorithm shifts (line 45) and it continues at the next position. Otherwise, the algorithm has to traverse over all couples (o, a) stored in the corresponding list and perform three-level comparison for every couple. The first level is comparison of the bytes in the encoded text (starting at the position given by the offset o) with the bytes of the encoded pattern B_a according to the shift/alignment a (see line 38). The second level (see line 40) is comparison of the prefix and it is applied only if the first level was successful. The third level (see line 42) is

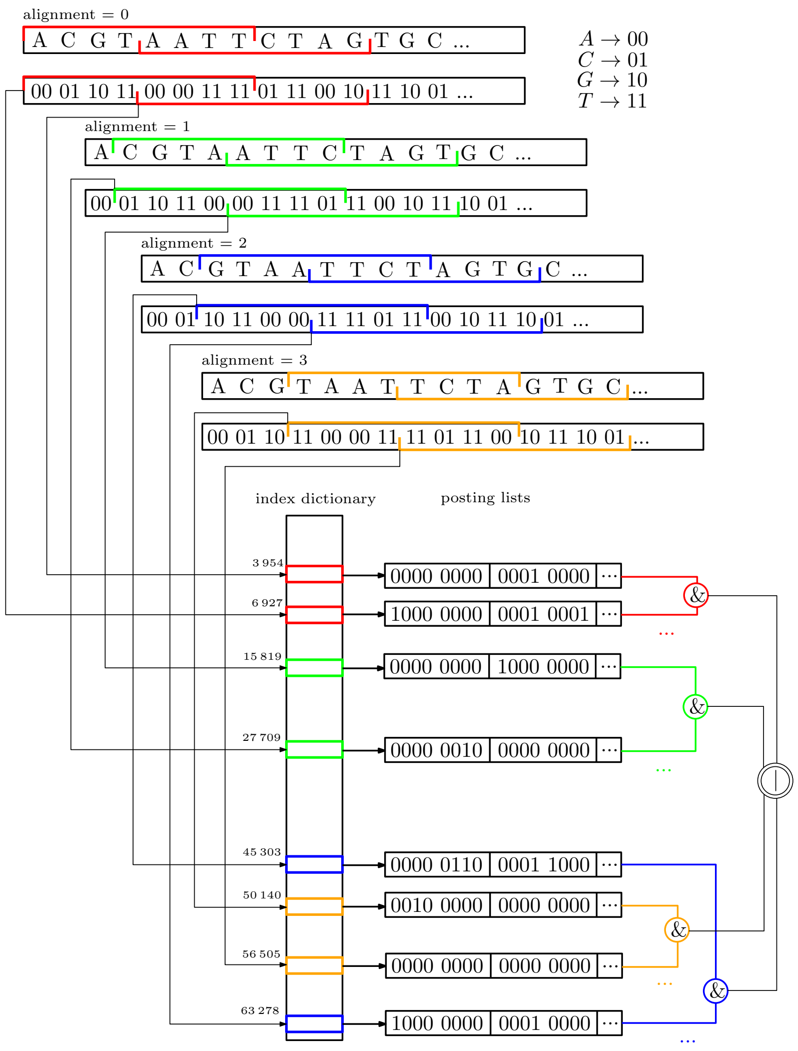

Figure 2 Block q-gram inverted index. Single colors (red, green, blue, yellow) represent the alignments $a \in \{0, 1, 2, 3\}$ of the pattern.

comparison of the suffix and it is applied only if the second level was successful. If all levels of the comparison are successful the algorithm reports a new occurrence at the corresponding position $4 \times (i - o) - a + 1$ in the raw text (line 44).

The preprocessing phase of the algorithm needs clearly $\mathcal{O}(m)$ time at most. In the function BUILDDICTIONARY, the algorithm consumes $\mathcal{O}(m)$ time to perform encoding (line 6) and $\mathcal{O}(\frac{m}{4})$ time to perform the `while` cycle (line 9). Other steps of the function are performed in the constant time. The function BUILDMASK contains all steps that are performed in the constant time. The worst-case time complexity for the searching phase of the algorithm is given by the `while` cycle (line 35) traversing the text and the `for each` cycle traversing the list of couples (line 37). Thus, the upper bound is $\mathcal{O}(nm)$. However, the average time is lower than linear for real genomic data. According to our tests on real data, the most of the factors (especially for `BAPM8`) occur only once in the pattern and therefore majority of the lists pointed from the dictionary D contain only one element. Furthermore, especially for longer patterns where the size of the pattern is significantly greater than the size of the tabulated factors, the algorithm jumps over majority of the processed text and so achieves lower than linear time. We can conclude that the worst-case time of the algorithm is $\mathcal{O}(nm)$, however the average expected time is lower than linear $\mathcal{O}(n)$.

The next logical step in improving efficiency of the searching is to add an index data structure. Navarro et al. [16] proved the efficiency of the block inverted index in combination with sequential scanning of the encoded text. `BAPM` works with the encoded q-grams ($q \in \{4, 8\}$) so we decided to supply it with the block q-gram inverted index. The $q = 8$ proved to be optimal in our experiments. The 8-gram factors are encoded into two-byte long values (short data type in C) which means that they are easily addressed and manipulated. Figure 2 gives a brief description of generating and applying the index. The index dictionary stores all encoded factors of all pattern alignments $a \in \{0, 1, 2, 3\}$. Every engaged dictionary entry points to a posting list implemented as a bitmap. Single bits correspond to blocks in the encoded text and are set to one when the q-gram occurs in the block.

Searching using the inverted index has the following steps. The pattern has to be encoded. For each alignment $a \in \{0, 1, 2, 3\}$ (represented by different colors in Figure 2) all two-byte long values are retrieved. The posting lists of all retrieved values (of a given alignment) are processed and bitwise-and operation is applied. Next, bitwise-or operation is applied among intermediate results of single alignments. Finally, the blocks of the encoded text corresponding to the set bits contain a possible occurrences of the pattern and need to be processed using `BAPM` to confirm the occurrences and report the exact positions in the text.

4 Experiments

We present experimental results that give a detailed comparison of our newly presented algorithms `BAPM4` and `BAPM8` with the state-of-the-art best algorithms. We considered all known algorithms focused especially on searching middle-sized and long patterns. The essence of `BAPM` (its byte orientation and its principle of tabulating all factors) predetermines this algorithm to search for patterns with length $m \geq 8$ bases. In particular, we compared `BAPM4` and `BAPM8` with the following algorithms (all of them from SMART library[2]):

- Exact Packed String Matching (EPSM) [7],
- Shift-Or algorithm (SO) [1],
- Backward-SRN-DAWG-Matching (BSDM4 and BSDM8) [6],
- Simplified BNDM with q-grams (SBNDMQ4 and SBNDMQ8) [22].

All the tested algorithms were implemented in C programming language[3]. We carried out our tests on Intel® Core$^{\text{TM}}$ i7-4702MQ 2.20 GHz, 8 GB RAM. We used compiler gcc version 5.4.0 with compiler optimization -O3. The tested patterns were chosen randomly from the input text and their length m was ranging from 12 to 256. All experiments were run in loop 1 000 times and we report the mean of the running time in milliseconds. All reported times represent measured `user` time + `sys` time and they always include any necessary preprocessing. For evaluating the algorithms, we used `Ecoli.txt` file from the Canterbury corpus[4] and `human100MB.txt` file that contains the sequence of human chromosome 15 from the project Ensembl[5].

[2] `http://www.dmi.unict.it/~faro/smart/`
[3] `BAPM4` and `BAPM8` implementation is available at `http://www.stringology.org/bapm/bapm.zip`
[4] `http://corpus.canterbury.ac.nz/descriptions/large/E.coli.html`
[5] `http://www.ensembl.org/info/data/ftp/index.html`

Table 1 `Ecoli.txt`: Locate times in milliseconds. The best results are bolded.

m	BAPM4	BAPM8	EPSM	SO	BSDM4	BSDM8	SBNDMQ4	SBNDMQ8
12	1.032	1.188	**1.031**	4.015	**1.031**	1.984	1.156	2.953
16	0.813	**0.609**	1.468	4.000	0.843	1.125	0.906	1.625
20	0.672	**0.407**	1.515	4.015	0.734	0.812	0.765	1.125
24	0.609	**0.328**	1.312	4.015	0.687	0.640	0.671	0.875
28	0.562	**0.265**	1.328	4.015	0.656	0.531	0.625	0.703
32	0.515	**0.219**	1.265	4.000	0.625	0.468	0.578	0.593
36	0.546	**0.203**	1.265	3.062	0.609	0.406	0.546	0.625
40	0.515	**0.187**	1.250	3.093	0.593	0.375	0.546	0.609
44	0.500	**0.172**	1.250	3.078	0.593	0.359	0.546	0.609
48	0.484	**0.156**	1.234	3.000	0.563	0.328	0.546	0.625
52	0.469	**0.141**	1.234	3.046	0.562	0.312	0.546	0.625
56	0.454	**0.140**	1.218	3.078	0.562	0.296	0.546	0.625
60	0.454	**0.125**	1.218	3.093	0.562	0.296	0.546	0.625
64	0.453	**0.124**	1.203	3.015	0.546	0.281	0.546	0.609
96	0.422	**0.102**	1.203	3.062	0.531	0.250	0.546	0.609
128	0.391	**0.094**	1.172	3.015	0.531	0.234	0.546	0.609
160	0.375	**0.092**	1.171	3.046	0.515	0.218	0.546	0.625
192	0.344	**0.092**	1.155	3.125	0.515	0.218	0.546	0.625
224	0.328	**0.095**	1.155	3.015	0.515	0.218	0.562	0.609
256	0.312	**0.098**	–	–	–	–	–	–

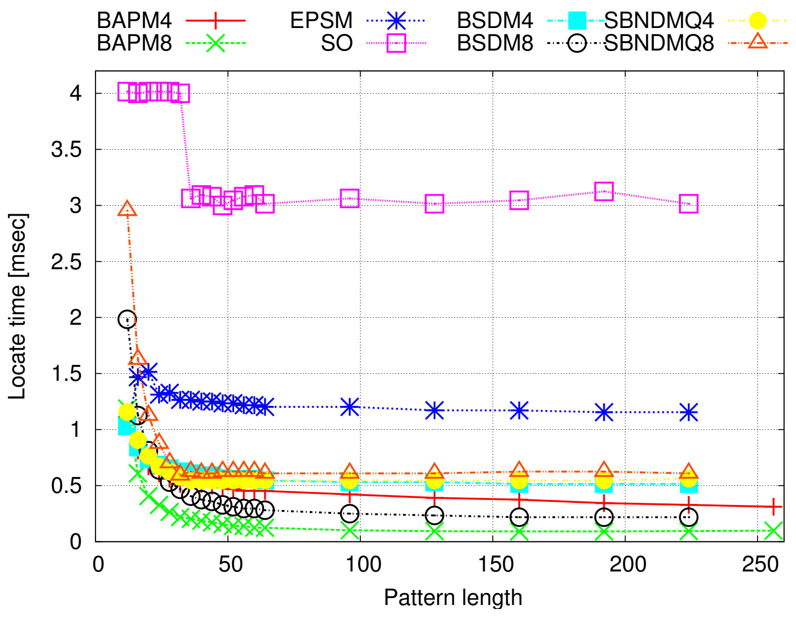

Figure 3 `Ecoli.txt`: Locate time depending on the length of the searched pattern m.

Table 1 and Figure 3 report the results of single algorithms when performed on the file `Ecoli.txt`. Our `BAPM8` algorithm achieved the best locate time for almost all pattern lengths. For the shortest pattern $m = 12$, the algorithms EPSM and BSDM4 overcame all the other competitors. The safe shift distance is limited for `BAPM`. It is given as the number of complete bytes of the encoded pattern minus one. In practise, it means shifting by $\lfloor \frac{12}{4} \rfloor - 1 = 2$ bytes for a pattern of length $m = 12$ and thus omitting only 50 % of the input text.

At the same time, `BAPM4` achieved better result than `BAPM8` for $m = 12$. The tabulated encoded factors of length one byte are sufficient for efficient filtration in the first step of searching. The higher memory consumption of `BAPM8` is not balanced by significantly more

Table 2 `human100MB.txt`: Locate times in milliseconds. The best results are bolded.

m	BAPM4	BAPM8	EPSM	SO	BSDM4	BSDM8	SBNDMQ4	SBNDMQ8
12	**21.250**	22.030	22.031	79.531	21.375	38.578	24.796	59.234
16	16.560	**11.250**	31.562	79.547	17.890	22.766	19.531	33.656
20	14.370	**7.500**	32.031	79.546	16.171	16.375	16.688	23.906
24	12.650	**5.940**	30.171	79.547	15.047	12.828	15.141	18.312
28	11.880	**4.680**	30.421	79.844	14.266	10.891	14.031	15.343
32	11.100	**3.900**	29.876	79.562	13.969	9.875	13.250	12.828
36	11.720	**3.440**	29.938	59.421	13.641	9.296	12.796	13.359
40	10.940	**3.130**	29.734	59.437	13.343	8.828	12.812	13.359
44	10.780	**2.810**	29.672	59.453	13.141	8.671	12.828	13.375
48	10.620	**2.650**	29.922	59.422	13.000	8.562	12.750	13.375
52	10.470	**2.500**	29.875	59.438	12.859	8.516	12.766	13.343
56	10.310	**2.340**	30.296	59.453	12.672	8.468	12.781	13.360
60	9.850	**2.340**	30.250	59.438	12.546	8.391	12.750	13.359
64	9.840	**2.190**	30.219	59.453	12.500	8.375	12.796	13.360
96	9.060	**2.030**	29.734	59.453	12.093	8.343	12.750	13.328
128	8.590	**2.030**	29.078	59.438	11.860	8.375	12.812	13.359
160	7.960	**2.030**	28.266	59.453	11.781	8.390	12.781	13.313
192	7.500	**2.030**	27.703	59.422	11.656	8.391	12.781	13.343
224	7.030	**2.030**	26.938	59.453	11.578	8.406	12.828	13.328
256	6.560	**2.030**	–	–	–	–	–	–

Table 3 Block inverted index: Locate time per occurrence in milliseconds (`Ecoli.txt`, `human100MB`).

	`Ecoli.txt`			`human100MB`		
m	BAPM8	LZ77	RLCSA	BAPM8	LZ77	RLCSA
12	0.717	**0.077**	0.156	**0.087**	0.096	0.175
16	0.507	**0.099**	0.159	**0.072**	0.092	0.159
20	0.287	**0.127**	0.170	0.104	**0.092**	0.155
24	0.177	**0.158**	0.173	0.126	**0.080**	0.153
28	**0.115**	0.177	0.177	0.127	**0.074**	0.126
32	**0.087**	0.176	0.167	0.117	**0.078**	0.129
36	**0.056**	0.195	0.180	0.109	**0.101**	0.134
40	**0.044**	0.251	0.192	**0.088**	**0.088**	0.121
44	**0.043**	0.271	0.181	**0.070**	0.111	0.123
48	**0.029**	0.262	0.175	**0.074**	0.108	0.123
52	**0.030**	0.283	0.175	**0.055**	0.126	0.127
56	**0.031**	0.273	0.196	**0.058**	0.189	0.139
60	**0.044**	0.301	0.195	**0.048**	0.187	0.136
64	**0.044**	0.329	0.194	**0.044**	0.216	0.137
96	**0.044**	0.460	0.222	**0.035**	0.550	0.200
128	**0.043**	0.606	0.251	**0.042**	0.799	0.240
160	**0.044**	0.775	0.268	**0.047**	1.215	0.295
192	**0.060**	0.958	0.302	**0.053**	1.701	0.367
224	**0.060**	1.108	0.329	**0.060**	2.021	0.401
256	**0.059**	1.553	0.350	**0.057**	2.226	0.436

efficient filtration and the search speed of `BAPM8` is lower for $m = 12$. The more efficient filtration outweighs for the longer patterns where $m \geq 16$. Similar results were achieved also on the file `human100MB.txt` (see Table 2 and Figure 4). `BAPM4` achieved the best result for $m = 12$ and `BAPM8` proved to be superior for $m \geq 16$. For $m = 224$, `BAPM8` is more than four times faster than the second fastest algorithm BSDM8.

We present the comparison of different indexing methods in Figure 5. Our `BAPM8` works together with block q-gram inverted index. We performed the experiments with $q = 8$ and the size of the block 102 400 bytes. `BAPM8` (together with the inverted index) was compared with LZ77 self-index [12] and RLCSA [19]. The presented results prove that block q-gram

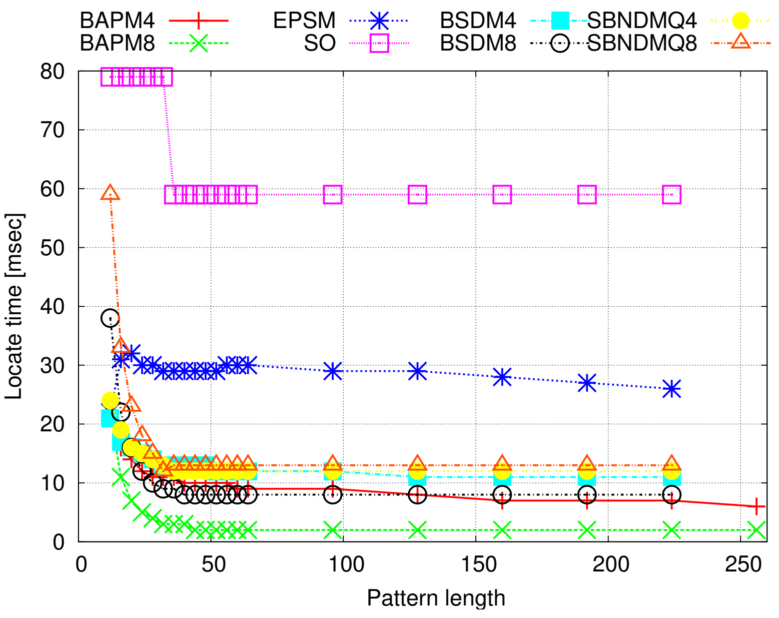

Figure 4 `human100MB.txt`: Locate time depending on the length of the searched pattern m.

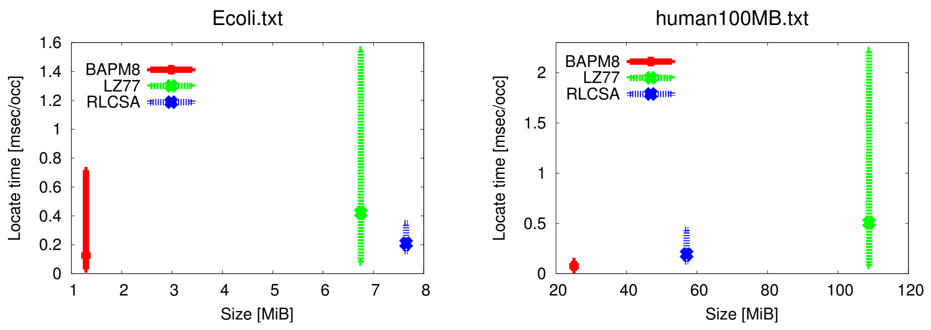

Figure 5 Comparison `BAPM8` supplemented with the block q-gram inverted index with other indexing methods. Minimum, average and maximum locate time per occurrence for different patterns (m ranging from 12 to 256) are reported. The horizontal axis presents the spaces consumption of single methods in MiB.

inverted index together with `BAPM8` represent a very good alternative to the other indexing methods, especially for sequences obtained using so-called *De Novo Sequencing* when LZ77 self-index and RLCSA cannot exploit their ability to compress highly similar sequences.

5 Conclusion and Future work

We presented a novel pattern matching algorithm, named `BAPM` (Byte-Aligned Pattern Matching), optimized for searching in encoded genomic sequences. The presented algorithm is based on two crucial properties: (i) processing at byte level of the input text; (ii) tabulating all factors of the pattern and searching for them in the filtration step. These two principles provide extraordinary efficiency of searching that was proved on real genomic data. We demonstrated that `BAPM` together with block q-gram inverted index can overcome other indexing methods and achieve locate time in the order of tens of microseconds per one occurrence.

In our future work, we aim to extend BAPM to be applicable for texts of larger alphabets (e.g., protein sequences, natural language texts). Furthermore, we intend to present a version of the algorithm for the degenerate strings (e.g., genomic sequences composed of symbols of IUPAC alphabet[6]) with its possible applications like searching for so-called Clustered-Clumps. The solution of this problem consists in an efficient data structure allowing the access to the sparse alphabet of the pattern factors in constant time.

References

1 Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, October 1992.

2 Ricardo A. Baeza-yates. Text retrieval: Theory and practice. In *In 12th IFIP World Computer Congress, volume I*, pages 465–476. Elsevier Science, 1992.

3 Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977.

4 Maxime Crochemore and Wojciech Rytter. *Text Algorithms.* Oxford University Press, Inc., New York, NY, USA, 1994.

5 B. Dömölki. An algorithm for syntactical analysis. *Computational Linguistics*, 3:29–46, 1964. Hungarian Academy of Science, Budapest.

6 Simone Faro and M. Oğuzhan Külekci. Fast and flexible packed string matching. *J. of Discrete Algorithms*, 28(C):61–72, September 2014.

7 Simone Faro and Thierry Lecroq. A fast suffix automata based algorithm for exact online string matching. In Nelma Moreira and Rogério Reis, editors, *Implementation and Application of Automata: 17th International Conference, CIAA 2012, Porto, Portugal, July 17-20, 2012. Proceedings*, pages 149–158. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

8 P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS'00, pages 390–398, Washington, DC, USA, 2000. IEEE Computer Society.

9 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. of the Thirty-second Annual ACM Symposium on Theory of Computing*, pages 397–406, New York, NY, USA, 2000.

10 R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

11 Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, March 1977.

12 Sebastian Kreft and Gonzalo Navarro. LZ77-like compression with fast random access. In *Proceedings of the 2010 Data Compression Conference*, DCC'10, pages 239–248, Washington, DC, USA, 2010. IEEE Computer Society.

13 Udi Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. Inf. Syst.*, 15(2):124–136, April 1997.

14 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Disc. Algorithms*, SODA'90, pages 319–327, Philadelphia, USA, 1990. Society for Industrial and Applied Mathematics.

15 Udi Manber and Sun Wu. Glimpse: A tool to search through entire file systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, page 4, Berkeley, CA, USA, 1994. USENIX Association.

[6] `https://iupac.org/`

16 Gonzalo Navarro, Edleno Silva de Moura, Marden S. Neubert, Nivio Ziviani, and Ricardo A. Baeza-Yates. Adding compression to block addressing inverted indexes. *Inf. Retr.*, 3(1):49–77, 2000. `doi:10.1023/A:1009934302807`.

17 Gonzalo Navarro and Mathieu Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, CPM'98, pages 14–33, London, UK, UK, 1998. Springer-Verlag.

18 Simon J. Puglisi, W. F. Smyth, and Andrew Turpin. Inverted files versus suffix arrays for locating patterns in primary memory. In Fabio Crestani, Paolo Ferragina, and Mark Sanderson, editors, *String Processing and Information Retrieval: 13th International Conference, SPIRE 2006, Glasgow, UK, October 11-13, 2006. Proceedings*, pages 122–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

19 J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. of the 15th International Symposium on String Processing and Information Retrieval*, SPIRE'08, pages 164–175, Berlin, Heidelberg, 2009. Springer-Verlag.

20 Daniel M. Sunday. A very fast substring search algorithm. *Commun. ACM*, 33(8):132–142, August 1990.

21 J. Tarhio, J. Holub, and E. Giaquinta. Technology beats algorithms (in exact string matching). *CoRR*, abs/1612.01506, 2016. URL: `http://arxiv.org/abs/1612.01506`.

22 Branislav Ďurian, Jan Holub, Hannu Peltola, and Jorma Tarhio. Tuning bndm with q-grams. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 29–37, Philadelphia, USA, 2009. Society for Industrial and Applied Mathematics.

23 P. Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Oct 1973.