On-line Searching in IUPAC Nucleotide Sequences

Petr Procházka and Jan Holub

Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University, Thákurova 2700/9, Prague 6, Czech Republic

Keywords: Consensus Nucleotide Sequences, Genomic Sequences, Degenerate Pattern Matching, q-gram Inverted Index.

Abstract: We propose a novel pattern matching algorithm for consensus nucleotide sequences over IUPAC alphabet, called `BADPM` (Byte-Aligned Degenerate Pattern Matching). The consensus nucleotide sequences represent a consensus obtained by sequencing a population of the same species and they are considered as so-called degenerate strings. `BADPM` works at the level of single bytes and it achieves sublinear search time on average. The algorithm is based on tabulating all possible factors of the searched pattern. It needs $O(m+m\alpha^2\log m)$-space data structure and $O(m\alpha^2)$ time for preprocessing where m is a length of the pattern and α represents a maximum number of variants implied from a 4-gram over IUPAC alphabet. The worst-case locate time is bounded by $O(nm^2\alpha^4)$ for `BADPM` where n is the length of the input text. However, the experiments performed on real genomic data proved the sublinear search time. `BADPM` can easily cooperate with the block q-gram inverted index and so achieve still better locate time. We implemented two other pattern matching algorithms for IUPAC nucleotide sequences as a baseline: Boyer-Moore-Horspool (`BMH`) and Parallel Naive Search (`PNS`). Especially `PNS` proves its efficiency insensitive to the length of the searched pattern m. `BADPM` proved its strong superiority for searching middle and long patterns.

1 INTRODUCTION

DNA sequencing is nowadays the integral part of several disciplines like personalized medicine, human genetics and oncology, forensic biology, microbiology and virology. 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 0.012 USD[1] which is 1 121 USD per human genome. Thanks to the general availability of the sequencing, we face the challenge of analyzing rapidly increasing number of genomic sequences, which includes their effective storage in the form allowing extremely fast searching.

The accessibility of sequencing methods has also enabled the development of projects focused on sequencing the population of many individuals of the same species. These projects include the 1000 Genomes Projects (Consortium, 2011) and the UK10K project (Consortium, 2015). Each sequenced population uses its 'reference genome', which forms the basis for studying genetic variations for other individuals. Pan-genomics (Marschall, 2018) shifts the reference genome to a representation of all genomic content in a certain species or a phylogenetic clade. A consensus sequence drawn from an entire population is one of the forms that the reference genome can take. The consensus sequence can be expressed as a degenerate string over a degenerate alphabet. We give a simple example of a consensus sequence expressed in IUPAC alphabet[2] in Figure 1. The consensus represents an alignment of seven different organisms. The different bases are emphasized with gray color. The consensus base is given in blue color and it means that more than one solid bases are valid (e.g. for R both A or G symbols are valid).

Knuth-Morris-Pratt (KMP) (Knuth et al., 1977) 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 $\mathcal{T}$. Boyer-Moore (BM) (Boyer and Moore, 1977) family algorithms represent backward pattern matching approach. BM algorithm allows skipping of some characters which leads to lower than linear average time. There exist

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

[2]https://iupac.org

Procházka, P. and Holub, J.
On-line Searching in IUPAC Nucleotide Sequences.
DOI: 10.5220/0007382900660077
In *Proceedings of the 12th International Joint Conference on Biomedical Engineering Systems and Technologies (BIOSTEC 2019)*, pages 66-77
ISBN: 978-989-758-353-7

```
homo sapiens:           T C T A G C A C T T A C T C T A T G C C T G C
pan paniscus:           T C T A G C A C T T A C T C T A T G C C T G C
chlorocebus sabaeus:    T C C A G C A C T T A C T C T G T G C C C G C
macaca fascicularis:    T C C A G C A C T T A C T C T G T G C C C A C
macaca mulatta:         T C C A G C A C T T A C T C T G T G C C C A C
papio anubis:           T C C A G C A C T T A C T C T G T G C C C G C
callithrix jacchus:     T C C A G C G C T T A C T C T A T A C C T A A
-----------------------------------------------------------------------
CONSENSUS:              T C Y A G C R C T T A C T C T R T R C C Y R M
```

Figure 1: Consensus sequence over IUPAC alphabet for different species (chromosome 7: 55 187 593 - 55 187 615).

also other variations of this algorithm given by Horspool (Horspool, 1980) or Sunday (Sunday, 1990). Suffix automaton (often called DAWG – Deterministic Acyclic Word Graph) is the essence of another algorithm achieving sublinear average time BDM (Backward DAWG Match) (Crochemore and Rytter, 1994). 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* (Dömölki, 1964; Baeza-yates, 1992) proved to be a very simple way 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 (Navarro and Raffinot, 1998) that achieved 20%-25% improvement in search time in comparison to its deterministic version BDM.

The standard pattern matching was naturally extended to the problem of pattern matching in degenerate strings, i.e. the strings where each degenerate symbol can represent a subset of solid symbols. The very first practical algorithm addressing the pattern matching in degenerate strings was ***agrep*** utility (Wu and Manber, 1992). Next, Navarro et al. (Navarro and Raffinot, 1998) introduced an extension of BNDM ***NR-grep*** (Navarro and Raffinot, 2002) allowing classes of characters at each position of the searched pattern. Holub et al. (Holub et al., 2008) implemented a version of Boyer-Moore-Sunday algorithm (Sunday, 1990) for degenerate strings and experimentally proved its superiority over BNDM for standard pattern lengths. Iliopoulos et al. (Iliopoulos et al., 2008) presented their algorithm based on transformation into the ‘restricted pattern matching problem’. Their implementation runs in $O(n)$ time where n is a size of the input text for sufficiently small patterns and DNA/RNA alphabet. The algorithm based on Landau and Vishkin’s algorithm was proposed by Crochemore (Crochemore et al., 2015). They achieved interesting upper bound $O(kn)$, where k is maximum number of degenerate positions in the text and n is the length of the text. However, the experimentally evaluated implementation of the algorithm was not provided. Finally, the concept of ‘Elastic Degenerate Strings’ was introduced in (Iliopoulos et al., 2017). Elastic Degenerate Strings provide another alternative to represent the consensus/pan-genomic sequences. Later, the practical implementation of the pattern matching algorithm in Elastic Degenerate Strings was given in (Grossi et al., 2017) and a version of algorithm allowing errors was presented in (Bernardini et al., 2017). Recently, Cisłak et al. (Cisłak et al., 2018) presented SOPanG algorithm solving elastic degenerate string matching problem in the same upper bound $O(N\lceil \frac{m}{w} \rceil)$, however, achieving an order of magnitude better time than (Grossi et al., 2017) in practical experiments.

The nucleotide consensus sequences generated from ***vcf*** (Variant Call Format) files have the following properties. They have very limited number of distinct symbols in the alphabet (16 symbols of IUPAC alphabet of which only 4 symbols are solid). At the same time, the sequences prove a very low rate of degenerate symbols (between two and three percent for all chromosomes). The two aforementioned properties led us to the idea to store the consensus sequences as a simply encoded strings (over the alphabet of only solid symbols) together with marking the relatively rare degenerate positions. Our algorithm `BADPM` is a natural extension (for degenerate strings) of the `BAPM` algorithm presented in (Procházka and Holub, 2017). It is based on tabulating all possible factors of the searched pattern. The algorithm performs searching for all pattern factors in its basic loop. After every factor occurrence, the occurrence of the whole pattern must be confirmed.

`BADPM` works at the level of bytes which ensures its high speed. This implies that the length of the tabulated factors is expected to be a multiple of 4. Our experiments proved that the factor length 8 symbols is ideal trade-off between factor occurrence frequency on one side and memory consumption and the length of the shift on the other side. However, this approach ensures sublinear time complexity on average since frequent shifting in the text can be applied. This holds especially for longer patterns since the shift is implied from the pattern length. The space for a simple data structure (storing the encoded consensus sequence) is upper bounded by $O(n\alpha + n \log n)$ where n is a size of the text and α is a maximum possible variants provided by 4-gram of IUPAC degenerate symbols. In practice, the needed space is close to $O(n)$ since the rate of degenerate symbols in the consensus sequences is low and the number of possible variants provided by the occurring 4-grams is low as well. The ensured worst-case search time is $O(nm^2\alpha^4)$ where m is a size of the searched pattern. However, our experiments proved that the factor hit rate is very low. Thus, the real time complexity is linear for short patterns ($16 \leq m \leq 32$) and sublinear for longer patterns ($m > 32$).

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 BADPM algorithm. We define PNS (Parallel Naive Search) algorithm as a baseline for experimental comparison with BADPM in Section 4. The Section 5 summarizes experimental results performed on real genomic data. We give the conclusion and some ideas for future work in Section 6.

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 $\sigma = |\Sigma| = O(1)$. 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 $\mathcal{P} = p_1p_2..p_m$ in a text $\mathcal{T} = t_1t_2..t_n$ 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 $\mathcal{P}$ in $\mathcal{T}$ is reported and (ii) *locate* when exact positions of the occurrences of $\mathcal{P}$ in $\mathcal{T}$ are reported.

Pattern substitution method (Manber, 1997) is a compression method when q-grams of symbols of the input text $\mathcal{T}$ (i.e., Σ^q) are substituted with an assigned byte value b where $b \in \{0, 1, \ldots, 255\}$. The pattern matching in the compressed (encoded) text means to find all occurrences of the compressed pattern $\mathcal{P}_C$ in the compressed text $\mathcal{T}_C$ (both defined over the alphabet of byte values $b \in \{0, 1, \ldots, 255\}$).

A symbol that represents only a single value is called *solid symbol*. For DNA nucleotide sequences, we consider the following alphabet of solid symbols $\Sigma = \{A, C, G, T\}$. *Degenerate symbol* is called a symbol that represents a subset of solid symbols. The number of possible degenerate symbols is limited by $2^{|\Sigma|} - 1$. The degenerate symbol can be represented as a union of covered solid symbols (e.g. $[AC]$) or as a specific single symbol (e.g. M). For DNA consensus sequences, we consider the following alphabet of degenerate symbols $\Sigma_{IUPAC} = \{A, C, G, T, R, Y, S, W, K, M, B, D, H, V, N\}$. The mapping IUPAC symbols to the solid symbols is given in Table 1.

Table 1: Mapping IUPAC degenerate symbols into the subsets of solid symbols.

IUPAC symbol	Subset	Bit coding
A	$\{A\}$	$\langle 0001 \rangle$
C	$\{C\}$	$\langle 0010 \rangle$
G	$\{G\}$	$\langle 0100 \rangle$
T	$\{T\}$	$\langle 1000 \rangle$
R	$\{A, G\}$	$\langle 0101 \rangle$
Y	$\{C, T\}$	$\langle 1010 \rangle$
S	$\{C, G\}$	$\langle 0110 \rangle$
W	$\{A, T\}$	$\langle 1001 \rangle$
K	$\{G, T\}$	$\langle 1100 \rangle$
M	$\{A, C\}$	$\langle 0011 \rangle$
B	$\{C, G, T\}$	$\langle 1110 \rangle$
D	$\{A, G, T\}$	$\langle 1101 \rangle$
H	$\{A, C, T\}$	$\langle 1011 \rangle$
V	$\{A, C, G\}$	$\langle 0111 \rangle$
N	$\{A, C, G, T\}$	$\langle 1111 \rangle$

For the degenerate text and the degenerate pattern we can distinguish the following three pattern matching problems (Iliopoulos et al., 2008). We design our algorithm to solve the last problem which implies that it can solve also the previous two problems.

Problem 1. *Given a degenerate text $\mathcal{T}$ and a pattern $\mathcal{P}$. The problem is to find all the occurrences of $\mathcal{P}$ in $\mathcal{T}$ i.e. to find all i such that for all j in $[1, m]$, $\mathcal{P}_j \in \mathcal{T}_{i+j-1}$.*

Problem 2. *Given a text $\mathcal{T}$ and a degenerate pattern $\mathcal{P}$. The problem is to find all the occurrences of $\mathcal{P}$ in $\mathcal{T}$ i.e. to find all i such that for all j in $[1, m]$, $\mathcal{T}_{i+j-1} \in \mathcal{P}_j$.*

Problem 3. *Given a degenerate text $\mathcal{T}$ and a degenerate pattern $\mathcal{P}$. The problem is to find all the occurrences of $\mathcal{P}$ in $\mathcal{T}$ i.e. to find all i such that for all j in $[1, m]$, $\mathcal{T}_{i+j-1} \cap \mathcal{P}_j \neq \emptyset$.*

Traditional *inverted index* consists of two major components: a vocabulary storing all distinct words occurring in the text $\mathcal{T}$ and a set of *posting lists* storing positions of all occurrences of a given word in the text $\mathcal{T}$. The vocabulary of a q-gram inverted index (Puglisi et al., 2006) 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 DEGENERATE PATTERN MATCHING

BADPM (Byte-Aligned Degenerate Pattern Matching) algorithm is focused on searching in consensus nucleotide sequences. It exploits their two basic properties: (i) very low size of the alphabet, (ii) very low frequency of degenerate symbols in the text. Considering the alphabet expressing the values of solid bases, the alphabet has only four symbols. Considering even the degenerate bases, we still end up with fifteen symbols at most. Furthermore, we experimentally tested ***vcf*** files from the 1000 Genomes Projects (Consortium, 2011) and we found that the frequency of the degenerate symbols varies from 2 to 3 percent for different chromosomes.

The idea behind BADPM is to consider the consensus (degenerate) sequence as a solid string and to encode it using a simple substitution encoding (Manber, 1997). The encoding method is 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 $s \in \Sigma^4$ ($A \rightarrow 00$, $C \rightarrow 01$, $G \rightarrow 10$, $T \rightarrow 11$). The degenerate symbols in the string are explicitly marked using *variantPos* array. Two other auxiliary arrays are needed to correctly evaluate the degenerate symbols in the string. Array *variantNum* stores the number of byte variants (implied from all degenerate symbols of the byte). Array *variants* stores the byte variants/values implied from the degenerate symbols occurring in the given byte. The solid symbols of the byte are kept as a left and/or right context in terms of the byte. This approach implies a space waste because of storing the solid symbols in the byte as the contexts, however, it accelerates the search since the byte operation level is kept. The aforementioned auxiliary arrays are very easy to serialize. Together with the base sequence, between 30 and 35 percent of the original size is used to store the BADPM encoded text. On the other hand, the encoded IUPAC sequence (using a simple substitution encoding - 2 IUPAC symbols encoded with one byte) requires exactly 50 percent of the original size.

The BADPM encoded text is depicted in Figure 3. Let α denotes a maximum number of variants implied from a 4-gram over IUPAC alphabet and n denotes the length of the input text. The 4-gram over IUPAC alphabet providing most variants is clearly $\langle NNNN \rangle$ that provides $\alpha = 4^4 = 256$ variants. We state the following space complexity of the encoded sequence in bits. Array *variantNum* storing the number of byte variants implied from the corresponding degenerate position clearly requires $O(n \log \alpha)$ space at most. At most $O(n \log n)$ space is required for *variantPos* array. At most $O(n\alpha)$ space is needed for *variants* array storing all byte variants. The encoded base sequence intuitively needs $O(n)$ space. Thus, the total space complexity is limited by $O(n\alpha + n \log n)$. However, the real space needed for encoded consensus sequences fits between 30 and 35 percent of their original/unencoded size.

The following algorithms and their complexities correspond to BADPM variant optimized for Problem 3 if not stated others. BADPM needs a simple preprocessing of the pattern that includes tabulating of all pattern factors of a given length. The optimal length of the factors is 8 symbols which means two-byte-long factors of the encoded pattern. Figure 2 depicts a simple data structure used to store the tabulated encoded factors of the pattern and it demonstrates also BADPM 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 (denoted as *dictionary*) with 65 536 entries (corresponding to 65 536 different two-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/unencoded 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 $a = (i-1) \mod 4$. We state the following space complexity of the dictionary (preprocessed pattern) in bits. The offset o requires $O(\log \frac{m}{4})$ bits, however, the alignment a needs only a constant number of bits $O(1)$. Thus, one element of the list requires $O(\log m)$ space. The space of the *dictionary* array is $O(\alpha^2)$ and every list can possibly contains up to $m - 7$ elements. This implies the total space complexity $O(m\alpha^2 \log m)$.

In practice, the algorithm uses byte coding for simpler and faster memory operations. Suppose the length of the raw pattern $m = 128$ bases which implies 32 bytes for the encoded pattern P_C. Further-

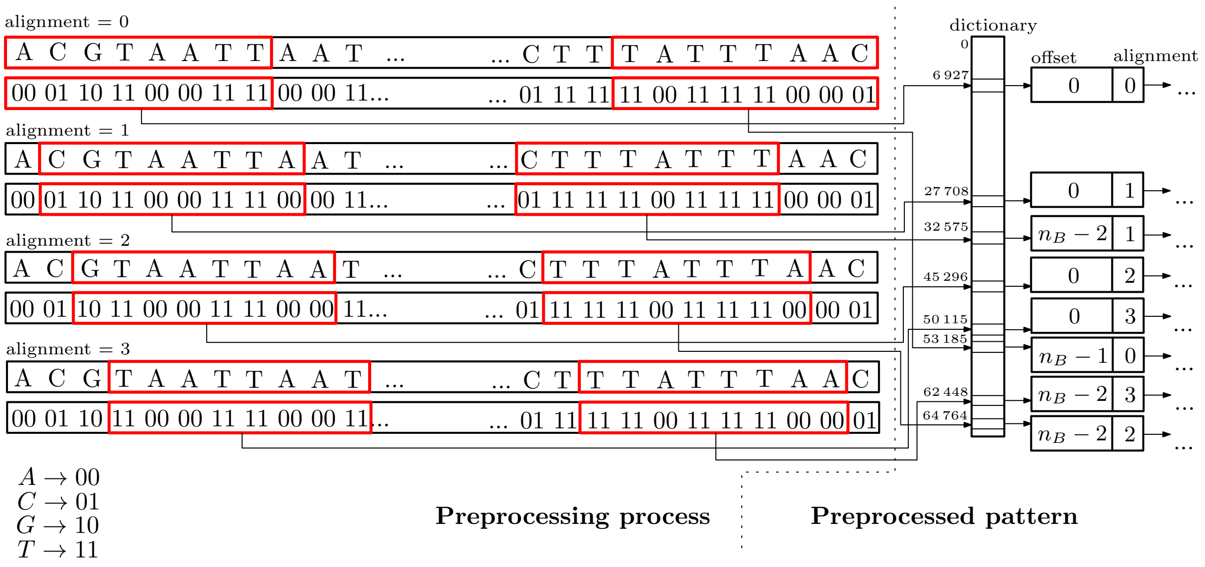


Figure 2: BADPM preprocessing phase for different byte alignments.

more, suppose every 16th symbol is degenerate implying 3 byte variants on average. BADPM needs to store $128-7+\frac{128}{16}\times 2 = 137$ 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 BADPM the total space is $137\times 2+65536 = 65810$ bytes, plus an overhead needed to implement the lists. Still, the data structure easily fits into 65 kiB of memory and it can be kept in a fast level of the computer memory.

Algorithm 1 describes preprocessing and searching phase of BADPM. 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 byte couples of the encoded aligned pattern and all corresponding variants (line 10) and it ensures storing the couples (offset, alignment) to their corresponding lists (line 13). $E_{j,2}$ represents a variant (composed of encoded solid symbols) provided by two-byte substring of the encoded pattern E starting at position j.

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 2 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 23) 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

Algorithm 1: `BADPM` preprocessing and searching phase.

```
 1: function BUILDDICTIONARY($\mathcal{P}$, $m$)
 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($\mathcal{P}, i+1, b$);
 7:         $B_i \leftarrow E$;
 8:         $j \leftarrow 1$;
 9:         while $j < \lfloor (m-i)/4 \rfloor$ do
10:             for each variant $E_{j,2}$ do
11:                 if $D_{E_{j,2}} = \emptyset$ then
12:                     $D_{E_{j,2}} \leftarrow$ create a new list storing offsets and alignments;
13:                 add a couple of offset and alignment $(j-1, i)$ to the list $D_{E_{j,2}}$;
14:                 $j \leftarrow j+1$;
15:         $i \leftarrow i+1$;

16: function BUILDMASK($\mathcal{P}$, $m$)
17:     $pref \leftarrow \emptyset$; $prefMask \leftarrow \emptyset$; $prefM \leftarrow \langle 00111111 \rangle$;
18:     $suf \leftarrow \emptyset$; $sufMask \leftarrow \emptyset$; $sufM \leftarrow \langle 11111100 \rangle$;
19:     $prefB \leftarrow$ ENCODE($\mathcal{P}, 1, 4$) $\gg 2$;
20:     $sufB \leftarrow$ ENCODE($\mathcal{P}, m-4, 4$) $\ll 2$;
21:     $la \leftarrow m \bmod 4$;
22:     $i \leftarrow 1$;
23:     while $i \leq 3$ do
24:         $prefMask_{4-i} \leftarrow prefM$;
25:         $sufMask_{(la+i) \bmod 4} \leftarrow sufM$;
26:         $pref_{4-i} \leftarrow prefB$;
27:         $suf_{(la+i) \bmod 4} \leftarrow sufB$;
28:         $prefM \leftarrow prefM \gg 2$; $prefB \leftarrow prefB \gg 2$;
29:         $sufM \leftarrow sufM \ll 2$; $sufB \leftarrow sufB \ll 2$;
30:         $i \leftarrow i+1$;

31: function SEARCH($\mathcal{T}$, $n$, $\mathcal{P}$, $m$)
32:     BUILDDICTIONARY($\mathcal{P}$,$m$);
33:     BUILDMASK($\mathcal{P}$,$m$);
34:     $shift \leftarrow \lfloor m/4 \rfloor - 2$;
35:     $i \leftarrow shift$;
36:     while $i < n$ do
37:         SYNCVARIANTPOINTERS($i$);
38:         for each variant $\mathcal{T}_{i,2}$ do
39:             if $D_{\mathcal{T}_{i,2}} \neq \emptyset$ then
40:                 for each couple of offset and alignment $(o,a) \in D_{\mathcal{T}_i}$ do
41:                     SYNCVARIANTPOINTERS($i-o$);
42:                     $r \leftarrow$ compare sequentially all bytes starting from $\mathcal{T}_{i-o}$ with $B_a$ considering all byte variants;
43:                     if $r = 0$ & $a \neq 0$ then
44:                         SYNCVARIANTPOINTERS($i-o-1$);
45:                         $r \leftarrow$ compare ($\mathcal{T}_{i-o-1}$ & $prefMask_a$) with $pref_a$ considering all variants $\mathcal{T}_{i-o-1}$;
46:                     if $r = 0$ & $a \neq la$ then
47:                         SYNCVARIANTPOINTERS($i-o+shift$);
48:                         $r \leftarrow$ compare ($\mathcal{T}_{i-o+shift}$ & $sufMask_a$) with $suf_a$ considering all variants $\mathcal{T}_{i-o+shift}$;
49:                     if $r = 0$ then
50:                         report an occurrence at position $4 \times (i-o) - a + 1$;
51:         $i \leftarrow i + shift$;
```

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 `BADPM`. After the preprocessing of the searched pattern (lines 32 and 33) the algorithm states a safe shift as the number of whole bytes of the encoded pat-

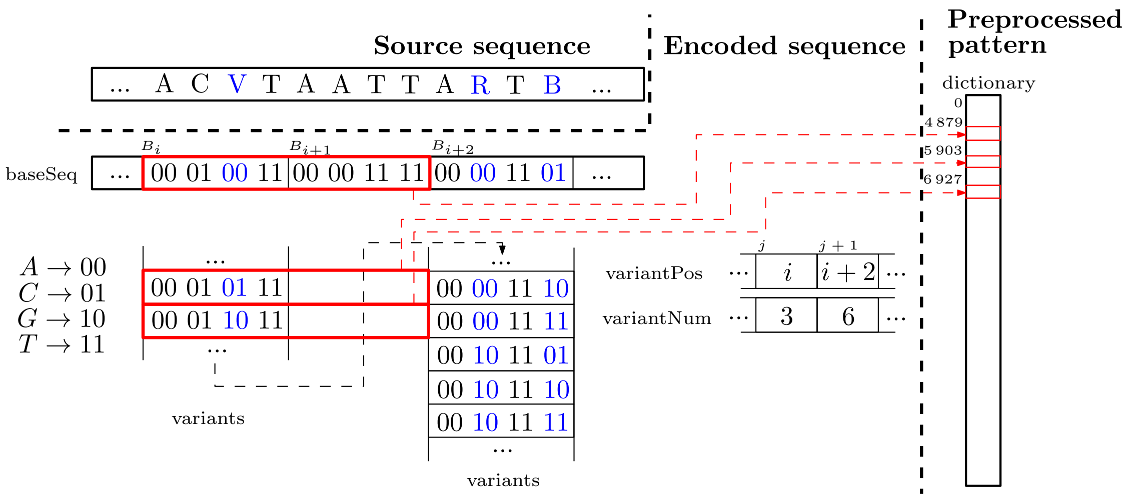


Figure 3: BADPM searching phase. Checking different factors implied from different byte variants.

tern minus two (line 34). The `while` cycle (line 36) traverses the encoded text $\mathcal{T}$ of length n. It always reads a byte couple value $\mathcal{T}_{i,2}$ and all its variants and the corresponding entries in the dictionary $D_{\mathcal{T}_{i,2}}$ are checked (line 39). The pointers for the array containing byte variants need to be synchronized for a given position $i-o$ (line 41). If the dictionary entry $D_{\mathcal{T}_{i,2}}$ is empty the algorithm shifts (line 51) 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 a 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 42). The second level (see line 45) is a comparison of the prefix and it is applied only if the first level was successful. The variant pointers need to be synchronized again (line 44). The third level (see line 48) is a 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 50).

Example 3.1. *We state an example to present the way* BADPM *processes the degenerate symbols. Suppose the situation depicted in Figure 3. Current position in the encoded text is i. The algorithm checks the couple of bytes B_i and B_{i+1}. The variant pointers are synchronized (line* 37 *in Algorithm 1) and the algorithm checks if B_i or/and B_{i+1} contain a degenerate symbol. The check is evaluated according to the next degenerate symbol position in variatPos array. The element variantPos$_j$ points to the position i which implies that B_i byte has more than one variant. The number of variants is variantNum$_j$ =* 3*. All byte variants are traversed (line* 38 *in Algorithm 1 and the red rectangles in Figure 3) and the corresponding entries in the dictionary are evaluated (lines* 39 - 50*). The evaluation consists of byte-by-byte comparing $\mathcal{T}_{i-o}$ with the encoded pattern (with corresponding alignment) B_a (line* 42*). The prefix and suffix comparison must be performed as well (line* 45 *and line* 48*). All byte variants are considered in the aforementioned comparisons.*

4 PARALLEL NAIVE SEARCH

We prepared two simple baseline algorithms for our experiments: Boyer-Moore-Horspool (BMH) algorithm and Parallel Naive Search (PNS). BMH represents a version of well-known algorithm (Horspool, 1980) adapted for degenerate strings. However, PNS (thanks to its engaging results proved in the experiments) deserves a little bit deeper description. The idea of PNS is based on intrinsic parallelism of computer words and on a bit representation of the alphabet symbols. All symbols of IUPAC alphabet are represented using code words with the size four bits (left or right nibble of a byte). The bits of a code word represent four solid symbols of the alphabet $\Sigma_{solid} = \{A,C,G,T\}$ starting from the least significant bit to the most significant bit. The bits of the code word are set when the represented symbol includes the given solid symbol (see the bit coding in Table 1). Both the processed text $\mathcal{T}$ and the searched pattern $\mathcal{P}$ need to be encoded in this fashion.

PNS performs standard forward pattern matching process. However, $\frac{w}{4}$ symbols are processed at once in parallel fashion where w is a size of the computer

Algorithm 2: PNS searching phase.

```
 1: function COMPAREWORDS($a$, $b$)
 2:     $sub \leftarrow \langle 0001\,0000 \rangle^{w/8}$;
 3:     $mask \leftarrow \langle 0000\,1111 \rangle^{w/8}$;
 4:     $c \leftarrow a \,\&\, b$;
 5:     $r_1 \leftarrow sub - (c \,\&\, mask)$;
 6:     $r_2 \leftarrow sub - ((c \gg 4) \,\&\, mask)$;
 7:     $r \leftarrow ((r_1 \,\&\, sub) \ll 1) \,\|\, (r_2 \,\&\, sub)$

 8: function SEARCH($\mathcal{T}$, $n$, $\mathcal{P}$, $m$)
 9:     $pMask \leftarrow$ BUILDPATTERNMASK($\mathcal{P}$,$m$);
10:     $rMask \leftarrow \langle 1100\ 1111 \rangle^{w/8}$;
11:     $i \leftarrow 1$;
12:     while $i \leq n - m$ do
13:         $j \leftarrow 1$;
14:         $r \leftarrow \langle 0000\ 0000 \rangle^{w/8}$;
15:         while $j \leq m$ do
16:             $r \leftarrow r \,\|$ COMPAREWORDS($\mathcal{T}_{i+j,\frac{w}{8}}$,$pMask_j$);
17:             if $(r \,\|\, rMask) = \langle 11111111 \rangle^{w/8}$ then
18:                 go to shift;
19:             $j \leftarrow j+1$
20:         $j \leftarrow 1$;
21:         while $j \leq m$ do
22:             $r \leftarrow r \,\|$ COMPAREWORDS($\mathcal{T}_{i+j,\frac{w}{8}} \ll 4$,$pMask_j$);
23:             if $(r \,\|\, rMask) = \langle 11111111 \rangle^{w/8}$ then
24:                 go to shift;
25:             $j \leftarrow j+1$
26:         report all positions/nibbles corresponding to unset bits in $r \,\|\, rMask$;
27: shift:
28:         $i \leftarrow i + \frac{w}{8}$;
```

word. SEARCH function described in Algorithm 2 represents a basic loop of the algorithm. The parameters of the SEARCH function are: an encoded text $\mathcal{T}$, a length of the encoded text n, an encoded pattern $\mathcal{P}$, a length of the encoded pattern m (the length of the raw/unencoded pattern is consequently $2m$). In the first step (line 9), the searched pattern $\mathcal{P}$ is preprocessed into the form of array *pMask*. The situation is depicted in the right part of Figure 4. This array consists of $2m$ computer words of length w. Every element of the array contains a mask composed of $\frac{w}{4}$ code words of the corresponding symbol of pattern $\mathcal{P}$. The first m elements of the array represent the odd symbols and the last m elements represent the even symbols of the searched pattern. The variable *rMask* represents a register necessary to store the intermediate results. The register is a computer word composed of $\frac{w}{8}$ bytes. The third and fourth most significant bits of j-th byte $rMask_j$ represent the activity of the left and right nibble of $(i+j)$-th byte in text $\mathcal{T}_{i+j}$. If the bit is zero then the corresponding position in the text is still a candidate for a match. The main loop of SEARCH function traverses over the encoded text $\mathcal{T}$ (line 12). In every step, a parallel comparison of corresponding factor of the text $\mathcal{T}_{i+j,\frac{w}{8}}$ is performed. The odd symbols of the pattern are compared in the first block (lines 15–19). The even symbols of the pattern are compared in the second block (lines 21–25). This is a technical optimization implying only one shifting of the compared text factor $\mathcal{T}_{i+j,\frac{w}{8}}$. Whenever no active position remains (no zero bit in *rMask*, lines 17 and 23), the comparing sequence stops and the text is shifted (line 28).

The parallel comparison of the text factor $\mathcal{T}_{i+j,\frac{w}{8}}$ with $pMask_j$ is done in function COMPAREWORDS. This function is performed in constant time and its usage is depicted in Figure 4. The bitwise AND operation between the text factor $\mathcal{T}_{i+j,\frac{w}{8}}$ and the corresponding $pMask_j$ is the first step (line 4). Next, the non-empty intersection in terms of single nibbles need to discovered. The right nibbles are checked at first. The intermediate results of bitwise AND operation is masked and then subtracted from *sub* register where only the fourth most significant bit is set (line 5). If the bit is set even after the subtraction it means that there is an empty intersection (corresponding symbols do not match) and the corresponding position in the text is excluded

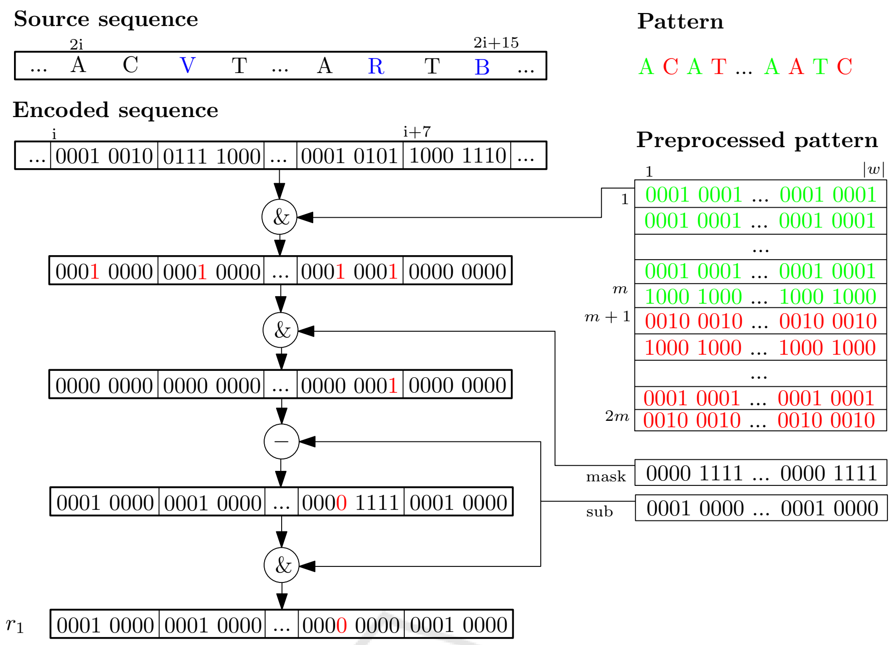


Figure 4: `PNS` searching phase. The first step of comparing the source sequence at position $2i$ with the preprocessed pattern.

from the potential candidates for a match. The same operation is performed for the left nibbles (line 6) and the intermediate results are merge into one register r (line 7). The searching phase of `PNS` clearly requires $O(\frac{n}{w}m)$ time at most where w is a size of the computer word. Very simple preprocessing of the pattern is required and it takes $O(m)$ time.

Example 4.1. *We demonstrate the function* COMPAREWORDS *in the following example (see Figure 4). Suppose text factor $\mathcal{T}_{2i,16}$ with a prefix "ACVT" and a suffix "ARTB". Furthermore, suppose a pattern $\mathcal{P}$ with a prefix "ACAT" and a suffix "AATC". After the bitwise* AND *operation between the encoded text factor and the corresponding mask pMask$_1$ only four of the depicted nibbles have non-empty intersection: the first symbol 'A', the third symbol 'V', the first symbol of the suffix 'A' and the second symbol of the suffix 'R'. The active bits are emphasize with red color. Next, the comparison of the right nibbles is performed and the corresponding bitwise* AND *operation with the register mask keeps active only the second nibble of the suffix (symbol 'R'). Subtraction is performed and the corresponding bit (fourth most significant bit in the first byte of the suffix) stays unset which implies that the corresponding position $2i+13$ in the text is still a candidate for a match. The processing of the left nibbles is skipped to simplify the example.*

5 EXPERIMENTS

We present experimental results that give a detailed comparison of our algorithm `BADPM` with the aforementioned baseline algorithms. Particularly, we performed the comparison in terms of locate time with Boyer-Moore-Horspool (`BMH`) algorithm (Horspool, 1980), with Parallel Naive Search (`PNS`) algorithm (see Section 4) and with Backward Nondeterministic DAWG Matching (`BNDM`) algorithm (Navarro and Raffinot, 1998) optimized for IUPAC alphabet. We planned also a comparison with the algorithm proposed by Iliopoulos in (Iliopoulos et al., 2008). However, the authors did not provide an implementation of their algorithm. The essence of `BADPM` (its byte orientation and its principle of tabulating all factors) predetermines this algorithm to search for patterns with length $m \geq 12$ bases. `BMH` and `BNDM` shall also profit from the longer searched patterns, while `PNS` is practically independent of the length of the searched pattern m. Theoretically, `PNS` locate time could worsen with the growing pattern length thanks to its time complexity $O(\frac{n}{w}m)$, however, the experiments did not prove this trend.

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

[3] `BADPM` implementation is available at http://www.stringology.org/badpm/badpm.zip

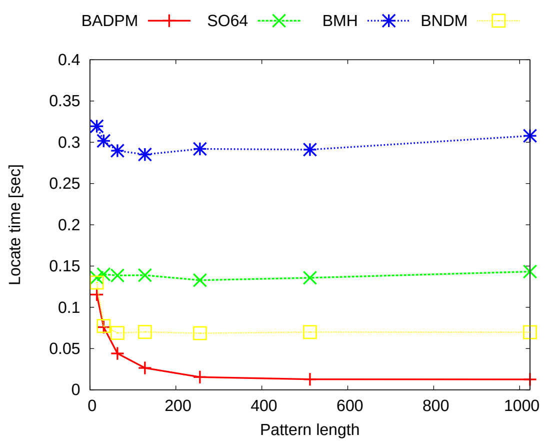


Figure 5: `Human chromosome 7`: Locate time depending on the length of the searched pattern *m*.

optimization -O3. The tested patterns were chosen randomly from the input text and their length m was ranging from 16 to 1 024. All experiments were run in loop 1 000 times and we report the mean of the running time in seconds. All reported times represent measured `user` time + `sys` time and they always include any necessary pattern preprocessing. For evaluating the algorithms, we used ***vcf*** files of different chromosomes downloaded from 1000 Genomes Projects (Consortium, 2011)[4]. The ***vcf*** files were transformed into the consensus text files over IUPAC alphabet using *bcftools consensus* utility by Sanger institute with parameter *-I* denoting the output given in IUPAC alphabet.

The first experiment describes dependency of locate time on the length of the searched pattern for single algorithms. `BADPM` proves strongly improving locate time when the length of the pattern $\mathcal{P}$ grows up to the value 256 bases. This is expected result since the shift of `BADPM` basic loop directly depends on the length of the pattern. `BMH` also derives its shifting from the pattern length. However, its improvement is not so strong thanks to small alphabet size and the degenerate string domain which limits shifting potential of `BMH`. `BNDM` should also profit from the increasing length of the searched pattern, however, it is limited by the size of the computer word. Since the length of the pattern exceeds the size of the computer word `BNDM` shows no further improvement. `BADPM` proved its superiority in locate time for middle-sized and long patterns. `BADPM` is almost three-times faster than `BMH` for $m = 16$. `PNS` and `BNDM` achieve only a slightly worse locate time than `BADPM` for $m = 16$. However, `BADPM` strongly dominates for longer patterns. It substantially improves its locate time until the pattern length $m = 256$ when the practical lower bound is achieved.

We compared the locate time of single algorithms on real genomic data obtained from 1000 Genomes Projects (Consortium, 2011) in our second experiment (see Figure 6). We chose a middle size of the searched pattern ($m = 16$) which implies no significant shifting benefit for `BADPM` which dominates especially for longer patterns. Still, `BADPM` achieved the best locate time for all tested chromosomes. `PNS` thanks to its parallelism performs a relatively large shifts corresponding to the computer word size w and so it achieves also very good locate time over all tested files. `BNDM` proved also very good results since $m = 16$ is clearly lower than the size of the computer word. The second vertical axis in Figure 6 reports the size of the chromosome files. The experiment proves that the locate time of all tested algorithms directly corresponds to the size of the tested files.

We combined all tested algorithms with a simple q-gram block inverted index defined in Section 2 to achieve locate time that is able to compete with the times achieved by other index data structures (e.g. *self-indexes*). We tested all algorithms for different block sizes (varying from 12 800 to 102 400 bases) of the q-gram inverted index (see Figure 7). We can observe that for smaller block sizes (12 800 and 25 600 bases, see Figure 7(a) and Figure 7(b)) and for longer patterns ($m \geq 512$ bases) the algorithms with simpler preprocessing phase (`PNS` and `BMH`) dominate. The reason is that the inverted index performs very efficient filtration of the blocks for longer patterns and smaller blocks. Thus, the following search algorithm itself processes only a small portion of the file and so the time needed to preprocess the searched pattern dominates over the search time of the algorithm. The vertical axis in Figure 7 are given in logarithmic scale to achieve a better overview of the achieved results. The minimal locate times achieved generally for the longer patterns are less than one millisecond. This locate time is two orders of magnitude worse than the time needed for standard (not degenerate) pattern matching problem (see Experiments section in (Procházka and Holub, 2017)). However, the time around one millisecond is still competitive for many applications.

6 CONCLUSION AND FUTURE WORK

We proposed the algorithm `BADPM` optimized for searching in the degenerate text over an alphabet of a small size. Practically, `BADPM` is designed to search

[4]ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20130502/

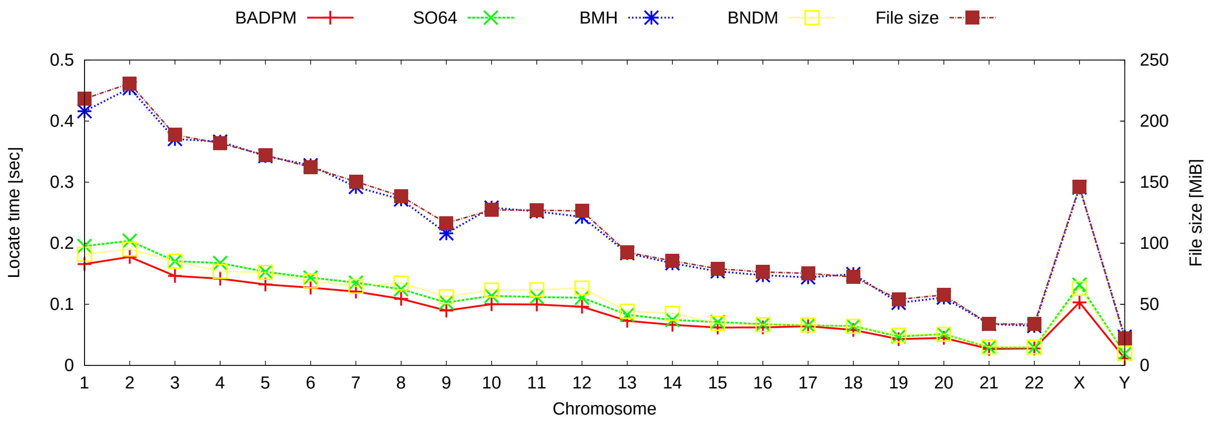
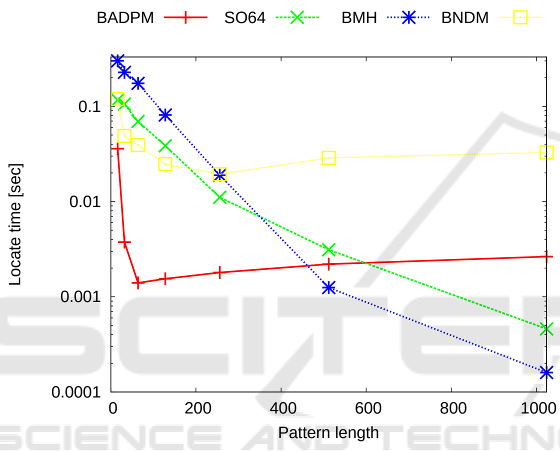
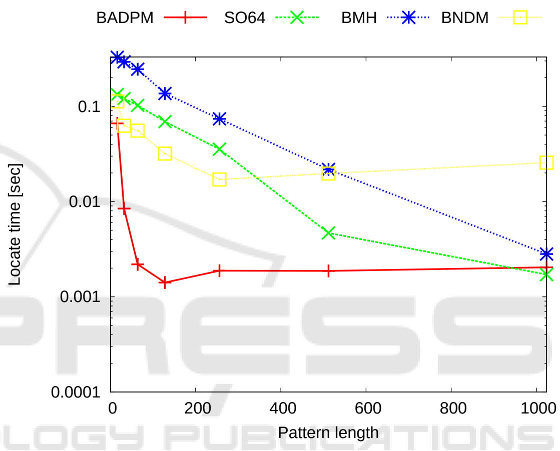


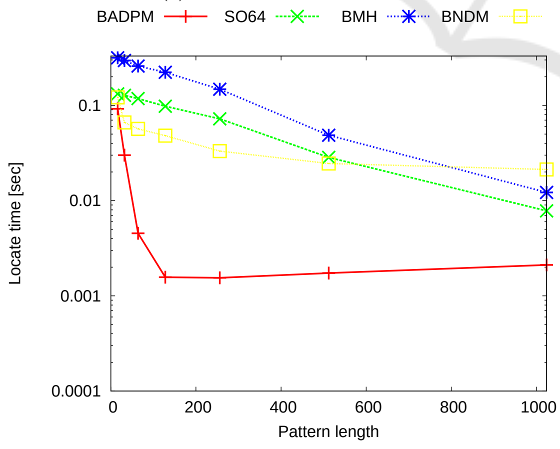
Figure 6: Locate time for different human chromosomes for $m = 16$. The second vertical axis represents the chromosome file size.



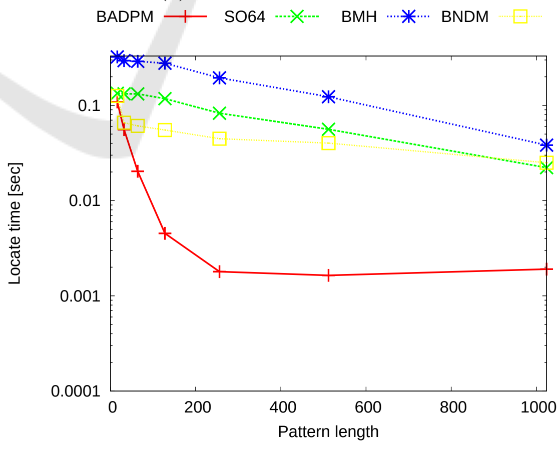
(a) Block size = 12 800



(b) Block size = 25 600



(c) Block size = 51 200



(d) Block size = 102 400

Figure 7: `Human chromosome 7`: Locate time depending on the length of the searched pattern m. The vertical axis is in logarithmic scale.

very efficiently in pangenomic data of a population of the same species, specifically ***vcf*** files. `BADPM` is based on the idea of `BAPM` (Procházka and Holub, 2017) and it inherits its basic properties: (i) processing at byte level of the input text; (ii) tabulating all possible factors of the pattern and searching for them in the filtration step. These two properties ensure very competitive locate time in the order of milliseconds and sublinear average time complexity. This was proven on real pangenomic data from 1000 Genomes Projects (Consortium, 2011).

We plan to design search algorithms for other forms of pangenomic data in our future work. This includes the algorithms optimized for so-called Elastic Degenerate Strings which is another form of representation genomic data for a population of the same species.

ACKNOWLEDGEMENTS

The research was partially supported by OP VVV project Research Center for Informatics no. CZ.02.1.01/0.0/0.0/16_019/0000765.

REFERENCES

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

Bernardini, G., Pisanti, N., Pissis, S. P., and Rosone, G. (2017). Pattern matching on elastic-degenerate text with errors. In Fici, G., Sciortino, M., and Venturini, R., editors, *String Processing and Information Retrieval*, pages 74–90, Cham. Springer International Publishing.

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

Cisłak, A., Grabowski, S., and Holub, J. (2018). Sopang: online text searching over a pan-genome. *Bioinformatics*, page bty506.

Consortium, T. . G. P. (2011). A map of human genome variation from population-scale sequencing. *Nature*, 473:544 EP –. Corrigendum.

Consortium, T. U. (2015). The uk10k project identifies rare variants in health and disease. *Nature*, 526:82 EP –.

Crochemore, M., Iliopoulos, C. S., Kundu, R., Mohamed, M., and Vayani, F. (2015). Linear algorithm for conservative degenerate pattern matching. *CoRR*, abs/1506.04559.

Crochemore, M. and Rytter, W. (1994). *Text Algorithms*. Oxford University Press, Inc., New York, NY, USA.

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

Grossi, R., Iliopoulos, C. S., Liu, C., Pisanti, N., Pissis, S. P., Retha, A., Rosone, G., Vayani, F., and Versari, L. (2017). On-line pattern matching on similar texts. In *28th Symposium on Combinatorial Pattern Matching, CPM 2017, July 4-6, 2017, Warsaw, Poland*, pages 9:1–9:14.

Holub, J., Smyth, W., and Wang, S. (2008). Fast pattern-matching on indeterminate strings. *Journal of Discrete Algorithms*, 6(1):37 – 50. Selected papers from AWOCA 2005.

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

Iliopoulos, C. S., Kundu, R., and Pissis, S. P. (2017). Efficient pattern matching in elastic-degenerate texts. In Drewes, F., Martín-Vide, C., and Truthe, B., editors, *Language and Automata Theory and Applications*, pages 131–142, Cham. Springer International Publishing.

Iliopoulos, C. S., Mouchard, L., and Rahman, M. S. (2008). A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching. *Mathematics in Computer Science*, 1(4):557–569.

Knuth, D. E., Morris, J. H., and Pratt, V. R. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350.

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

Marschall, T. (2018). Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, 19(1):118–135.

Navarro, G. and Raffinot, M. (1998). 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. Springer-Verlag.

Navarro, G. and Raffinot, M. (2002). *Frontmatter*, pages i–iv. Cambridge University Press.

Procházka, P. and Holub, J. (2017). Byte-aligned pattern matching in encoded genomic sequences. In *17th Int. Workshop on Algorithms in Bioinformatics, WABI 2017, August 21-23, 2017, Boston, MA, USA*, pages 20:1–20:13.

Puglisi, S. J., Smyth, W. F., and Turpin, A. (2006). *Inverted Files Versus Suffix Arrays for Locating Patterns in Primary Memory*, pages 122–133. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

Wu, S. and Manber, U. (1992). Agrep - a fast approximate pattern-matching tool. In *In Proc. of USENIX Technical Conference*, pages 153–162.