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

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 $O(m + \alpha m^2 \log m)$-space data structure and $O(m \alpha^2)$ time for preprocessing where $m$ is a length of the pattern and $\alpha$ 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 $T$. 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
also other variations of this algorithm given by Hornspool (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 (Domíček, 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 $O(\alpha n)$, where $\alpha$ 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 $O(N[m/w])$, 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 positions. Our algorithm $\text{BADPM}$ is a natural extension (for degenerate strings) of the $\text{BAM}$ 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.

$\text{BADPM}$ 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 $\alpha$ is a maximum possible variants provided by 4-gram of IUPAC degenerate sym-
bols. 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 \( O(nm^22^3) \) 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 \( \text{BADPM} \) algorithm. We define \( \text{PN} \) (Parallel Naive Search) algorithm as a baseline for experimental comparison with \( \text{BADPM} \) 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 \( \Sigma \). The length of the string \( x \) is \( n = |x| \). The size of the alphabet \( \Sigma \) 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 \( \varepsilon \) 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 \( \Sigma \) 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 (Manber, 1997) is a compression method when \( q \)-grams of symbols of the input text \( T \) (i.e., \( \Sigma^q \)) 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 \( \mathcal{P} \) in the compressed text \( \mathcal{C} \) (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 \( 2^{\sigma^q} - 1 \). 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 \( \Sigma_{\text{IUPAC}} = \{A,C,G,T,R,Y,S,W,K,M,B,D,H,V,N\} \). The mapping IUPAC symbols to the solid symbols is given in Table 1.

<table>
<thead>
<tr>
<th>IUPAC symbol</th>
<th>Subset</th>
<th>Bit coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{A}</td>
<td>(0001)</td>
</tr>
<tr>
<td>C</td>
<td>{C}</td>
<td>(0010)</td>
</tr>
<tr>
<td>G</td>
<td>{G}</td>
<td>(0100)</td>
</tr>
<tr>
<td>T</td>
<td>{T}</td>
<td>(1000)</td>
</tr>
<tr>
<td>R</td>
<td>{A,G}</td>
<td>(0101)</td>
</tr>
<tr>
<td>Y</td>
<td>{C,T}</td>
<td>(1010)</td>
</tr>
<tr>
<td>S</td>
<td>{C,G}</td>
<td>(0110)</td>
</tr>
<tr>
<td>W</td>
<td>{A,T}</td>
<td>(1001)</td>
</tr>
<tr>
<td>K</td>
<td>{G,T}</td>
<td>(1100)</td>
</tr>
<tr>
<td>M</td>
<td>{A,C}</td>
<td>(0011)</td>
</tr>
<tr>
<td>B</td>
<td>{C,G,T}</td>
<td>(1110)</td>
</tr>
<tr>
<td>D</td>
<td>{A,G,T}</td>
<td>(1101)</td>
</tr>
<tr>
<td>H</td>
<td>{A,C,T}</td>
<td>(1011)</td>
</tr>
<tr>
<td>V</td>
<td>{A,C,G}</td>
<td>(0111)</td>
</tr>
<tr>
<td>N</td>
<td>{A,C,G,T}</td>
<td>(1111)</td>
</tr>
</tbody>
</table>

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 \( T \) and a pattern \( P \). The problem is to find all the occurrences of \( P \) in \( T \) i.e. to find all \( i \) such that for all \( j \) in \( [1,m] \), \( P_j \in T_{i+j-1} \).

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

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

Traditional inverted index consists of two major components: a vocabulary storing all distinct words occurring in the text \( T \) 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 (Puglisi et al., 2006) is composed of all possible \( q \)-grams of the alphabet \( \Sigma \), i.e., \( \Sigma^q \). 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, ≪ for shift-left operation and ≫ 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 ref 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: \( f : \Sigma^4 \rightarrow B \) 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 \( \alpha \) 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/uncoded 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 a array (denoted as dictionary) with 65536 entries (corresponding to 65536 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/encoded pattern \( o = i \lfloor \frac{m}{4} \rfloor \). 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 \( O(\alpha^2 \log m) \).

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-
more, suppose every 16th symbol is degenerate implying 3 byte variants on average. BADPM needs to store 128 \( - 7 + \frac{128 \times 2}{16} = 137 \) 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 index for encoding; and the number of bases/symbols parameters are: the text to be encoded; the starting index for encoding; and the number of bases/symbols.

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 \( a \in \{0, 1, 2, 3\} \). The prefix for the alignment \( a = 0 \) is an empty string \( \varepsilon \) and therefore it is not stored. The pointer to the array of the suffix values \( suf \) is shifted by the value \( la + i \) 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

![Figure 2: BADPM preprocessing phase for different byte alignments.](image-url)
Algorithm 1: BADPM preprocessing and searching phase.

1: function BUILD_DICTIONARY(P, m)
2: D ← ∅; B ← ∅;
3: i ← 0;
4: while i ≤ 3 do
5:   b ← ⌊(m − i)/4⌋ × 4;
6:   E ← ENCODE(P[i + 1 + b]);
7:   Bb ← E;
8:   j ← 1;
9:   while j < ⌊(m − i)/4⌋ do
10:      for each variant E_j do
11:          if D_{j, i} = 0 then
12:              D_{j, i} ← create a new list storing offsets and alignments;
13:          add a couple of offset and alignment (j − 1, i) to the list D_{j, i};
14:          j ← j + 1;
15:      i ← i + 1;
16:      for each variant E_j do
17:          pref ← ∅; prefMask ← ∅; prefM ← (00111111);
18:          suf ← ∅; sufMask ← ∅; sufM ← (11111100);
19:          prefB ← ENCODE(P[m − 4, i]) ≫ 2;
20:          sufB ← ENCODE(P[m − 4, i]) ≪ 2;
21:          if i < m mod 4 then
22:              la ← m − i;
23:          while i ≤ 3 do
24:              prefMask_{a−1} ← prefM;
25:              sufMask_{a−1} ≡ a ← sufM;
26:              pref_{a−1} ← prefB;
27:              if a = 0 then
28:                  sufM ← prefM ≫ 2;
29:                  sufB ← prefB ≫ 2;
30:                  i ← i + 1;
31:         end for
32:         for each couple of offset and alignment (o, a) ∈ D_{j, i} do
33:             if D_{j, o} ≠ 0 then
34:                 for each variant E_j do
35:                     if D_{j, o} ≠ 0 then
36:                         if r = 0 & a ≠ la then
37:                             r ← compare (E_i, o−1, & prefMask_a) with pref_a considering all variants E_i, o−1;
38:                         else
39:                             r ← compare (E_i, o−1, & prefMask_a) with pref_a considering all variants E_i, o−1;
40:                     end if
41:                 end for
42:             end if
43:         end for
44: end for
45: end while
46: end function

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-
The while cycle (line 36) traverses the encoded text $T$ of length $n$. It always reads a byte couple value $T_{o-1}$ and all its variants and the corresponding entries in the dictionary $D_{T_{o-1}}$ 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 $D_{T_{o-1}}$ 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 $B_a$ and $B_{a+1}$. The variant pointers are synchronized (line 37 in Algorithm 1) and the algorithm checks if $B_a$ or/and $B_{a+1}$ contain a degenerate symbol. The check is evaluated according to the next degenerate symbol position in variatPos array. The element variantPos$_i$ points to the position $i$ which implies that $B_a$ byte has more than one variant. The number of variants is variantNum$_i = 3$. 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 $T_{i-o}$ 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 $T$ and the searched pattern $P$ need to be encoded in this fashion.

PNS performs standard forward pattern matching process. However, $w$ symbols are processed at once in parallel fashion where $w$ is a size of the computer word.
Algorithm 2: PNS searching phase.

1: function COMPARE WORDS (a, b)
2: sub ← ⟨00010000⟩ w/8;
3: mask ← ⟨00001111⟩ w/8;
4: i ← 0;
5: r1 ← sub − (c & mask);
6: r2 ← sub − ((c ≧ 4) & mask);
7: r ← ((r1 & sub) ≦ 1) || (r2 & sub);

8: function SEARCH(T, n, P, m)
9: pMask ← BUILD PATTERN MASK(P, m);
10: rMask ← ⟨1100 1111⟩ w/8;
11: i ← 1;
12: while i ≤ n − m do
13: j ← 1;
14: r ← ⟨0000 0000⟩ w/8;
15: while j ≤ m do
16: r ← r || COMPARE WORDS(T[i+j−1:4], pMask);
17: if (r || rMask) = ⟨1111 1111⟩ w/8 then
18: go to shift;
19: j ← j+1
20: j ← 1;
21: while j ≤ m do
22: r ← r || COMPARE WORDS(T[i+j−1:4], pMask);
23: if (r || rMask) = ⟨1111 1111⟩ w/8 then
24: go to shift;
25: j ← j+1
26: report all positions/nibbles corresponding to unset bits in r || rMask;
27: shift:
28: i ← i + w/4;

The parallel comparison of the text factor $T_{i+j−1:4}$ with $pMask_{j}$ is done in function COMPARE WORDS. This function is performed in constant time and its usage is depicted in Figure 4. The bitwise AND operation between the text factor $T_{i+j−1:4}$ and the corresponding $pMask_{j}$ is the first step (line 4). Next, the non-empty intersection in terms of single nibbles need to be 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.

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 $T_{i+j−1:4}$. 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).

On-line Searching in IUPAC Nucleotide Sequences
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 $O(\frac{w}{m})$ 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 $T_{216}$ with a prefix “ACVT” and a suffix “ARTB”. Furthermore, suppose a pattern $P$ 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 $O(\frac{w}{m})$, 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\textsuperscript{®} Core\textsuperscript{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
Locate time [sec] for searching in the degenerate text over an alphabet of a small size. Practically, BADPM is designed to search on-line Searching in IUPAC Nucleotide Sequences

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 1024. All experiments were run in loop 1000 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 ref files of different chromosomes downloaded from 1000 Genomes Projects (Consortium, 2011). The ref 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 \( P \) 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 12800 to 102400 bases) of the \( q \)-gram inverted index (see Figure 7). We can observe that for smaller block sizes (12800 and 25600 bases, see Figure 7(a) and Figure 7(b)) and for longer patterns \( (m \geq 512) \) 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.

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 millisec-
onds 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


