Dictionary Matching in Elastic-Degenerate Texts with Applications in Searching VCF Files On-line

Solon P. Pissis
Department of Informatics, King's College London, London, UK
solon.pissis@kcl.ac.uk

Ahmad Retha[1]
Department of Informatics, King's College London, London, UK
ahmad.retha@kcl.ac.uk

Abstract

An *elastic-degenerate string* is a sequence of n sets of strings of total length N. It has been introduced to represent multiple sequence alignments of closely-related sequences in a compact form. For a standard pattern of length m, pattern matching in an elastic-degenerate text can be solved on-line in time $\mathcal{O}(nm^2+N)$ with pre-processing time and space $\mathcal{O}(m)$ (Grossi et al., CPM 2017). A fast bit-vector algorithm requiring time $\mathcal{O}(N\cdot\lceil\frac{m}{w}\rceil)$ with pre-processing time and space $\mathcal{O}(m\cdot\lceil\frac{m}{w}\rceil)$, where w is the size of the computer word, was also presented. In this paper we consider the same problem for a set of patterns of total length M. A straightforward generalization of the existing bit-vector algorithm would require time $\mathcal{O}(N\cdot\lceil\frac{M}{w}\rceil)$ with pre-processing time and space $\mathcal{O}(M\cdot\lceil\frac{M}{w}\rceil)$, which is prohibitive in practice. We present a new on-line $\mathcal{O}(N\cdot\lceil\frac{M}{w}\rceil)$-time algorithm with pre-processing time and space $\mathcal{O}(M)$. We present experimental results using both synthetic and real data demonstrating the performance of the algorithm. We further demonstrate a real application of our algorithm in a pipeline for discovery and verification of *minimal absent words* (MAWs) in the human genome showing that a significant number of previously discovered MAWs are in fact false-positives when a population's variants are considered.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases on-line algorithms, algorithms on strings, dictionary matching, elastic-degenerate string, Variant Call Format

Digital Object Identifier 10.4230/LIPIcs.SEA.2018.16

Acknowledgements We warmly thank Panagiotis Charalampopoulos (King's College London), Roberto Grossi (University of Pisa) and Nadia Pisanti (University of Pisa) for their insightful comments.

1 Introduction

A set of closely-related sequences can be represented in different ways to reduce its size and improve search performance. DNA sequences of the same species or closely-related species can be combined into a *pan-genome* [17, 24, 13, 20], the result of a *multiple sequence alignment* (MSA) of these sequences. Most regions in the DNA sequences are in consensus but they exhibit differences at some positions consisting of letter substitutions, insertions or deletions. Various data structures have been proposed for storing pan-genomes [8, 2, 22, 24] – many

[1] AR is supported by the Graduate Teaching Assistant scheme of the Department of Informatics at King's College London

designs are realized from observing the result of aligning the related sequences in an MSA fashion. Consider the following example.

```
ATGCAACGGGTA--TTTTA
ATGCAACGGGTATATTTTA
ATGCACCTGG----TTTTA
```

The first five columns of the MSA all match, so when this is compacted, it creates a deterministic segment with a single string: `ATGCA`. The next letter in the MSA is `A` for the first and second sequence but `C` for the third, making it the site of a variant. The second variant in the example is similarly a single-base substitution variant. But the third variant site consists of insertions and deletions. These are represented differently in state-of-the-art data structures for the purpose of storage and indexing for on-line pattern searches.

Some researchers choose to represent the variants in the combined sequence in the form of *De Bruijn* graphs [24] or specialized implementations of the data structure – *Variation Graphs* [22]. Other researchers use *Trie*-based data structures such as the *Bloom Filter Trie* in [8] or compressed *Suffix Tree* data structures [2]. All of these are indexes constructed mainly for fast searching of patterns; and thus much effort has gone into solving this *off-line* version of the problem through pre-processing the set of similar sequences [9, 14, 22, 15]. Less research to-date has gone into the *on-line* version of this problem [19, 12, 7, 3, 18].

A text-based, on-line searchable representation for a set of similar sequences was suggested in [12], namely, the notion of *elastic-degenerate string* (ED string). Specifically, aligned sequences can be compacted into one sequence made up of *deterministic* and *non-deterministic* (or *degenerate*) segments. Deterministic segments contain letters that are in perfect conformity among the different sequences, meaning all the letters match, while degenerate segments mark polymorphic sites containing substitutions, insertions or deletions.

The MSA above can be converted into an ED string by representing the first deterministic segment with the single string `ATGCA`, and representing the next segment, which happens to be degenerate, with the set `{A,C}`. An empty string marker ε is used to represent a deletion, as is done for the third degenerate site in the MSA, consisting of the set `{TA,TATA,`ε`}`. The resulting ED string of the MSA above example is as follows.

$$\tilde{T} = \{\ \texttt{ATGCA}\ \} \cdot \left\{ \begin{matrix} \texttt{A} \\ \texttt{C} \end{matrix} \right\} \cdot \{\ \texttt{C}\ \} \cdot \left\{ \begin{matrix} \texttt{G} \\ \texttt{T} \end{matrix} \right\} \cdot \{\ \texttt{GG}\ \} \cdot \left\{ \begin{matrix} \texttt{TA} \\ \texttt{TATA} \\ \varepsilon \end{matrix} \right\} \cdot \{\ \texttt{TTTTA}\ \}$$

The motivation for solving the on-line version of the problem is to remove the burden of building disk-based indexes or rebuilding them with every update in the sequences. Indexes are often cumbersome, take a lot of time and space to build, and require lots of disk space to be stored. Their usage carries the assumption that the data is static or changes very infrequently. Solutions to the on-line version can be beneficial for a number of reasons: (a) efficient on-line solutions can be used in combination with partial indexes as practical trade-offs; (b) efficient on-line solutions for exact pattern matching can be applied for fast average-case approximate pattern matching, similar to standard strings; (c) on-line solutions can be useful when one wants to search for a few patterns in many ED texts.

Variant Call Format (VCF) is a file format that has become the standard way of storing variants for pan-genomes and in next-generation sequencing. These specially-formatted, often compressed text files, in combination with a reference genome, are able to document all insertions, deletions and substitutions that occur in a population. While it is possible – for the purpose of searching for patterns – to recreate the genome of all individuals (samples) in the pan-genome as deterministic strings, it is very impractical and requires a lot of

processing power and disk space. It also defeats the purpose of storing the information in the VCF format in the first place. We were motivated to make it possible to do on-line searching of one or more patterns in a pan-genome without extracting sample sequences. Our solution takes the position of variants in the VCF file and encodes them as degenerate segments of an ED text. In this way, we are able to search a pan-genome on-line given the reference sequence and the associated VCF files. We created a tool EDSO (available at `https://github.com/webmasterar/edso`) for creating ED files which are directly searchable.

Our Contributions. Our focus in this paper is extending existing solutions for exact on-line pattern matching in ED texts, specifically, the algorithm of [7] through adding the ability to search for *multiple* patterns simultaneously. Grossi et al. [7] presented an algorithm requiring time $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(m \cdot \lceil \frac{m}{w} \rceil)$, where m is the length of the *single* pattern and w is the size of the computer word. A straightforward generalization of the existing bit-vector algorithm for a set of patterns of total length M would require time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(M \cdot \lceil \frac{M}{w} \rceil)$, which is prohibitive in practice. In this paper we present a new algorithm requiring time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ with pre-processing time and space $\mathcal{O}(M)$. We present experimental results using both synthetic and real data demonstrating the performance of our algorithm. Finally, we present a real application of our algorithm's use as part of identifying and verifying *minimal absent words* (MAWs) in the *Homo sapiens* pan-genome with data in the VCF taken from the 1000 Genomes Project [23]. Specifically, we show that a significant number of previously discovered MAWs are in fact false-positives when a population of genomes is considered.

2 Definitions and Notation

2.1 Strings

We begin with a few definitions from [4]. An *alphabet* Σ is a non-empty finite set of letters of size $\sigma = |\Sigma|$. A *(deterministic) string* on a given alphabet Σ is a finite sequence of letters of Σ. For this work, we assume that the alphabet is fixed, i.e. $\sigma = \mathcal{O}(1)$. The *length* of a string x is denoted by $|x|$. For two positions i and j on x, we denote by $x[i\,..\,j] = x[i]\,..\,x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j (it is empty if $j < i$), and by ε we denote the *empty string*. The set of all strings on an alphabet Σ (including the empty string ε) is denoted by Σ^*. For any string $y = uxv$, where u and v are strings, if $u = \varepsilon$ then x is a *prefix* of y. Similarly, if $v = \varepsilon$ then x is a *suffix* of y. If u and v are non-empty strings, we call x an *infix* of y. We say that x is a *proper factor* of y if x is a factor (resp. prefix/suffix) of y distinct from y.

We say that the string x is an *absent word* of string y if x does not occur in y. We consider absent words of length at least 2 only. An absent word x of length m, $m \geq 2$, of y is *minimal* if and only if all its proper factors occur in y. This is equivalent to saying that a minimal absent word (MAW) of y is of the form aub, $a, b \in \Sigma, u \in \Sigma^*$, such that au and ub are factors of y but aub is not.

▶ **Example 1.** Let $y =$ `ABAACA`. Its factors of lengths 1 and 2 are `A`, `B`, `C`, `AA`, `AB`, `AC`, `BA`, and `CA`. The set of MAWs of y is obtained by combining the aforementioned factors: {`BB`, `BC`, `CB`, `CC`, `AAA`, `AAB`, `BAB`, `BAC`, `CAA`, `CAB`, `CAC`}.

2.2 Elastic-Degenerate Strings

An *elastic-degenerate string* (ED string) $\tilde{X} = \tilde{X}[0]\tilde{X}[1]\dots\tilde{X}[n-1]$, of *length* n, on an alphabet Σ, is a finite sequence of n *degenerate letters*. Every *degenerate letter* $\tilde{X}[i]$, for all $0 \leq i < n$, is a non-empty set of strings $\tilde{X}[i][j]$, with $0 \leq j < |\tilde{X}[i]|$, where each $\tilde{X}[i][j]$ is a deterministic string on Σ. The *total size* of $\tilde{X}$ is defined as

$$N = \sum_{i=0}^{n-1} \sum_{j=0}^{|\tilde{X}[i]|-1} |\tilde{X}[i][j]|.$$

Only for the purpose of computing N, $|\varepsilon| = 1$. We remark that, for an ED string $\tilde{X}$, the size and the length are two distinct concepts.

We say that a string y *matches* an ED string $\tilde{X} = \tilde{X}[0]\dots\tilde{X}[m'-1]$ of length $m' > 1$, denoted by $y \approx \tilde{X}$, if and only if string y can be decomposed into $y_0 \dots y_{m'-1}$, $y_i \in \Sigma^*$, such that:

1. there exists a string $s \in \tilde{X}[0]$ such that a suffix of s is $y_0 \neq \varepsilon$;
2. if $m' > 2$, there exists $s \in \tilde{X}[i]$, for all $1 \leq i \leq m'-2$, such that $s = y_i$;
3. there exists a string $s \in \tilde{X}[m'-1]$ such that a prefix of s is $y_{m'-1} \neq \varepsilon$.

Note that, in the above definition, we require that both y_0 and $y_{m'-1}$ are non-empty to avoid spurious matches at the beginning or end of an occurrence. A string y is said to have an *occurrence* ending at position j in an ED string $\tilde{T}$ if there exist $i < j$ such that $\tilde{T}[i]\dots\tilde{T}[j] \approx y$, or, if there exists $s \in \tilde{T}[j]$ such that y occurs in s.

▶ **Example 2.** Suppose we have a pattern $p = \texttt{ACACA}$ of length $m = 5$ and an ED string $\tilde{T}$ of length $n = 6$ and total size $N = 18$; the first occurrence of p starts at position 1 and ends at position 2 of $\tilde{T}$; and the second one starts at position 2 and ends at position 4.

$$\tilde{T} = \{\texttt{C}\} \cdot \{\texttt{A}, \texttt{C}\} \cdot \{\texttt{AC}, \texttt{ACC}, \texttt{CACA}\} \cdot \{\texttt{C}, \varepsilon\} \cdot \{\texttt{A}, \texttt{AC}\} \cdot \{\texttt{C}\}$$

We are now in a position to formally define the main problem of this paper.

Multiple Elastic-Degenerate String Matching (*MEDSM*) **Input:** A set P of strings of total length M and an ED string $\tilde{T}$ of length n and total size N. **Output:** All pairs (p, j): an occurrence of string $p \in P$ ends at position j in $\tilde{T}$.

3 Algorithmic Toolbox

3.1 Suffix Tree

Let x be a string of length $n > 0$. The *suffix tree* $\mathcal{ST}_x$ of string x is a compacted trie representing all suffixes of x. The nodes of the trie which become nodes of the suffix tree are called *explicit* nodes, while the other nodes are called *implicit*. Each edge of the suffix tree can be viewed as an upward maximal path of implicit nodes starting with an explicit node. Moreover, each node belongs to a unique path of that kind. Thus, each node of the trie can be represented in the suffix tree by the edge it belongs to and an index within the corresponding path. The label of an edge is its first letter. We let $\mathcal{L}(v)$ denote the *path-label* of a node v, i.e. the concatenation of the edge labels along the path from the root to v. We say that v is

path-labelled $\mathcal{L}(v)$. Additionally, $\mathcal{D}(v) = |\mathcal{L}(v)|$ is used to denote the *string-depth* of node v. Node v is a *terminal* node if its path-label is a suffix of x, that is, $\mathcal{L}(v) = x[i\,..\,n-1]$ for some $0 \leq i < n$; here v is also labelled with index i. It should be clear that each factor of x is uniquely represented by either an explicit or an implicit node of $\mathcal{ST}_x$. Once $\mathcal{ST}_x$ is constructed, it can be traversed in a depth-first manner to compute $\mathcal{D}(v)$ for each node v.

▶ **Fact 3** ([5, 4]). *Given a string x of length n, $\mathcal{ST}_x$ can be constructed in time and space $\mathcal{O}(n)$. Finding all Occ_p occurrences of a string p of length m in x can be performed in time $\mathcal{O}(m + Occ_p)$ using $\mathcal{ST}_x$.*

3.2 The Shift-And Algorithm

The *Shift-And* algorithm is an exact pattern matching algorithm that takes advantage of the parallelism of bitwise operations performed on a computer word [16]. It works by simulating a *Nondeterministic Finite Automaton* (NFA) and uses bit-level operations to simultaneously update the states of the NFA in a single CPU cycle. This offers speed-ups bounded by the number of bits in a computer word w, where, typically, on modern computer architectures, we have that $w = 64$. For short patterns, where $m = \mathcal{O}(w)$, searching a text of length n runs in $\mathcal{O}(n)$ time but for longer patterns, the search takes $\mathcal{O}(n \cdot \lceil \frac{m}{w} \rceil)$ time. The pre-processing time of the algorithm is $\mathcal{O}(\sigma \cdot \lceil \frac{m}{w} \rceil + m) = \mathcal{O}(m)$ thus making it suitable for small-sized alphabets and short patterns. The *Shift-And* algorithm can be easily generalized for a set of patterns; it is then known as the *Multiple Shift-And* algorithm [16].

▶ **Fact 4** ([16]). *Given a set P of strings of total length M, a string x of length N, and a computer word of size w, finding all occurrences of the patterns in P in x takes time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$ after pre-processing time $\mathcal{O}(M)$.*

In addition to using the *Shift-And* algorithm for searching whole patterns, we take advantage of its ability to compute suffix/prefix overlaps between string $s \in \tilde{T}[i]$, where $\tilde{T}[i]$ is the ith set arriving on-line, and the set P of patterns we are searching for. Given s, we can find all the prefixes of a pattern $p \in P$ of length m by searching $s[|s| - m + 1\,..\,|s| - 1]$ if $|s| \geq m$ or $s[0\,..\,|s| - 1]$, otherwise. This updates the NFA to mark any prefixes of the patterns occurring as suffixes of s. We store the states in a bit vector which we bitwise-OR to itself for each string $s \in \tilde{T}[i]$. The resulting state bit vector memorizes all the prefixes ending at position i in $\tilde{T}$, and we then use *Shift-And* in the searching stage of the algorithm to search the $(i+1)$th set to find a suffix, that either completes the match for some pattern in P, or further extends some prefix of a pattern, in which case the algorithm updates the search state. We summarize the above description in the following fact.

▶ **Fact 5.** *Given a set S of strings of total length $N = \sum_{s \in S} |s|$ and a set P of strings of total length $M = \sum_{p \in P} |p|$, computing the suffix/prefix overlaps of S and P can be done in time $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$.*

4 The Multi-EDSM Algorithm

4.1 Our Data Structure

We define the following auxiliary problem of independent interest.

Occurrences Vector Data Structure (*OccVec*)
Input: A string x of length n.
Query: Given a string α on-line, return a pointer to a bit vector B, with $B[i] = 1$ if and only if α occurs at starting position i of x, and otherwise $B[i] = 0$.

In what follows, we show the following lemma.

▶ **Lemma 6.** *Given a parameter $1 \leq \tau \leq \lceil n/w \rceil$, a data structure of size $\mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$ can be constructed in time and space $\mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$ answering OccVec queries in time $\mathcal{O}(|\alpha| + \tau)$.*

Let us denote the data structure of Lemma 6 over string x by Occ-Vector$_x$. Observe that, if we set $\tau = 1$, we essentially have the $\mathcal{O}(n \cdot \lceil n/w \rceil)$-sized data structure proposed by Grossi et al in [7], which is constructible in time and space $\mathcal{O}(n \cdot \lceil n/w \rceil)$.

Construction. We start by constructing the suffix tree $\mathcal{ST}_x$ of x. By Fact 3, this can be done in time and space $\mathcal{O}(n)$.

We next convert $\mathcal{ST}_x$ to a binary tree using a standard procedure (see [6], for instance). We process each explicit node of $\mathcal{ST}_x$ with out-degree $k > 2$ as follows. Let v be a node with children $u_1, \ldots, u_k$, and $k \geq 3$. We replace v with $k-1$ new nodes $v_1, \ldots, v_{k-1}$; make u_1 and u_2 be the left and right children of v_1, respectively; and for each $\ell = 2, \ldots, k-1$, we make $v_{\ell-1}$ and $u_{\ell+1}$ be the left and right children of v_ℓ, respectively. If v is not the root of $\mathcal{ST}_x$ then we set the parent of v_{k-1} to be the parent of v; otherwise, v_{k-1} is the root. This procedure can at most double the size of $\mathcal{ST}_x$ so still is in $\mathcal{O}(n)$. For clarity of presentation, in what follows, we use $\mathcal{ST}_x$ to refer to the resulting binary tree. (Note that we can keep a copy of the original $\mathcal{ST}_x$ and a pointer for each node from the original to its binary version).

We next rely on the classic notion of *micro-macro* tree decomposition [1]. We apply this decomposition on (the binary version of) $\mathcal{ST}_x$. Let τ be some input parameter, $1 \leq \tau \leq \lceil n/w \rceil$. We decompose $\mathcal{ST}_x$ in $\mathcal{O}(n/\tau)$ disjoint subgraphs called *micro trees*. Each micro tree is of size *at most* τ and contains *at most two* boundary nodes that are adjacent to nodes in other micro trees. The topmost of these boundary nodes is the *root* of the whole micro tree, and the other one is called the *bottom* boundary node. Such a decomposition is always possible and can be found in time $\mathcal{O}(n)$ (see [1] for more details).

For each boundary node v of a micro tree, we store a bit vector b_v, where $b_v[i] = 1$, if the terminal node representing the ith suffix of x is a descendant of v, and otherwise $b_v[i] = 0$. For the bottom boundary node v of micro tree t, b_v can be computed by merging the bit vectors from the roots of the micro trees that are adjacent to v and then add manually the terminal nodes within t for the root boundary node of t. By the above description and the fact that we have $\mathcal{O}(n/\tau)$ micro trees, the total size of Occ-Vector$_x$, and therefore the time to construct it, are bounded by $\mathcal{O}(n + \lceil n/\tau \rceil \cdot \lceil n/w \rceil) = \mathcal{O}(\lceil n/\tau \rceil \cdot \lceil n/w \rceil)$, for $1 \leq \tau \leq \lceil n/w \rceil$. Occ-Vector$_x$ also includes a linked-list $\mathcal{L}$ of integers from $[0, n-1]$ used to maintain the bit vectors when a new query arrives. This completes the construction.

Querying. Given a pattern α, we spell the pattern from the root of $\mathcal{ST}_x$ until we reach the last explicit node v. This takes time $O(|\alpha|)$ for constant-sized alphabets. There are then two cases to consider:

- If v is a boundary node of some micro tree, we simply return a pointer to b_v; this takes constant time.
- If v is not a boundary node, we first need to obtain the starting positions (labels) of all terminal nodes of the micro tree in the subtree rooted at v, and set the corresponding

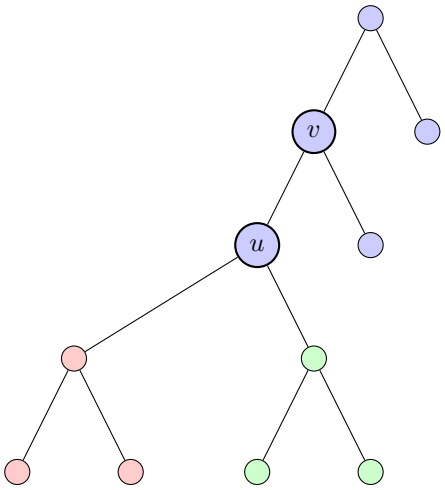


Figure 1 Three micro trees: the topmost in light blue and the bottommost ones in light red and light green. If the query reaches node v, then the terminal node in the subtree rooted at v in the topmost micro tree must be combined with the bit vector stored in the bottom boundary node u.

bits on in the bit vector b_u, where u is the bottom boundary node of the micro tree. We then return a pointer to the updated b_u. (If no such node u exists, we simply set these bits on in an empty bit vector.) The whole process takes time $\mathcal{O}(\tau)$: traverse the micro tree and set the bits on. We also need to store these starting positions in $\mathcal{L}$ for the next query. In the beginning of the next query we will need to set the bits at these indices off from b_u and empty $\mathcal{L}$. This maintenance cost requires time $\mathcal{O}(\tau)$ due to the size of the micro trees and so we charge it to this query. Inspect Figure 1 in this regard.

Hence any query requires time $\mathcal{O}(|\alpha| + \tau)$. By the above description we ultimately arrive at Lemma 6.

4.2 Pre-Processing Stage

The pre-processing stage of our algorithm consists in pre-processing the set P of our patterns. We view the set P as the concatenation of its elements to form a new string y of length M.

The first step is the pre-processing of the pattern set P of combined length M in the *Mutiple Shift-And* algorithm [16]. We create σ bit vectors of size $\lceil \frac{M}{w} \rceil$ and for each letter a in Σ we set $I_a[i] = 1$ if $y[i] = a$. Therefore this first step requires time and extra space $\mathcal{O}(M + \sigma \cdot \lceil \frac{M}{w} \rceil) = \mathcal{O}(M)$, for constant-sized alphabets.

The second step is a simple application of Lemma 6 over string y of length M with the additional steps of filtering out non-infix positions and subtracting 1 from the index positions. This makes it possible to maintain infix extensions during the search stage. We build the Occ-Vector$_P$ data structure by setting $\tau = \lceil \frac{M}{w} \rceil$, thus restricting its size and construction time to $\mathcal{O}(M)$, resulting also in $\mathcal{O}(|\alpha| + \lceil \frac{M}{w} \rceil)$ query time for the search stage.

The total time and space for the pre-processing stage are thus in $\mathcal{O}(M)$.

4.3 On-line Searching Stage

After the pre-processing stage, every degenerate letter S of text $\tilde{T}$ can be searched one after the other in an on-line manner by passing them to the Search function (see Algorithm 1). We maintain the state of the search in bit vector B in between searches and use temporary

Algorithm 1 Multiple Elastic-Degenerate String Matching search function.

```
1:  procedure Search(S)
2:      if isFirstSegment(S) then
3:          for s ∈ S do
4:              if |s| ≥ m_min and s ≠ ε then
5:                  B_3 ← 0
6:                  Multi-Shift-And-Search(s, B_3)
7:                  Report any matches found
8:          B ← Overlaps(S)
9:      else
10:         B_1 ← Overlaps(S)
11:         if ε ∈ S then
12:             B_1 ← B_1 | B
13:         for s ∈ S and s ≠ ε do
14:             ▷ Pattern suffix completion / full pattern searching
15:             B_2 ← B
16:             Multi-Shift-And-Search(s, B_2)
17:             Report any matches found
18:             ▷ Maintain valid infix positions
19:             if |s| ≤ m_max − 2 then
20:                 B_3 ← B & Occ-Vector_P(s)
21:                 B_1 ← B_1 | Left-Shift(B_3, |s|)
22:         B ← B_1
```

bit vectors B_1, B_2 and B_3 to update the state during processing. By $m_{\min}$ and $m_{\max}$ we denote the length of the shortest and longest patterns in pattern set P, respectively.

In the first degenerate letter, for every string $s \in S$ of length $|s| \geq m_{\min}$, we call the Multi-Shift-And-Search function with a fresh state to find any patterns that occur in s. In any case, the Overlaps function is called for computing the suffix/prefix *overlaps* between every string $s \in S$ and the set P of patterns we are searching for. The function essentially memorises the prefixes starting in the current segment using B. By Facts 4 and 5, lines 2-8 require $\mathcal{O}(||S|| \cdot \lceil \frac{M}{w} \rceil)$ time. For subsequent degenerate letters, we use the state of B from the previously searched letter to continue the search with Multi-Shift-And-Search. This time the function is called regardless of the length of s because it is used to find whole patterns as well as prefixes of patterns that began in the previously searched letters and whose suffixes end in the current letter. By Facts 4 and 5, this requires $\mathcal{O}(||S|| \cdot \lceil \frac{M}{w} \rceil)$ time.

Then we consider how to handle infixes (see line 19). We only need to process strings short enough to be considered as infixes of a pattern and we query them with the Occ-Vector$_P$ data structure to mark the positions where each infix starts. Querying Occ-Vector$_P$ for a string s requires $\mathcal{O}(|s| + \tau) = \mathcal{O}(|s| + \lceil \frac{M}{w} \rceil)$ time by Lemma 6. We bitwise-AND the bit vector it returns with B to maintain only the states started or continued from the previous letter. On the next line, we use the Left-Shift function to update the position of the bits to reflect the state of the search while ensuring that the bits are not shifted past the end of a pattern. What bits remain as 1s are bitwise-ORed with B_1 to update the state to maintain partial search states. This takes time $\mathcal{O}(\lceil \frac{M}{w} \rceil)$.

The final line of the algorithm saves the final search state of the segment to bit vector B, ready for the next on-line letter to be sent to the Search function. A full illustrative example of the search stage is provided below.

With the above description we arrive at the main result of this paper.

▶ **Theorem 7.** *Algorithm Multi-EDSM solves the MEDSM problem in an on-line manner in time* $\mathcal{O}(N \cdot \lceil \frac{M}{w} \rceil)$*. Algorithm Multi-EDSM requires pre-processing time and space* $\mathcal{O}(M)$*.*

Notably, our algorithm improves on the pre-processing time and space of [7] by a factor of $\mathcal{O}(\lceil \frac{m}{w} \rceil)$; namely, it improves on the algorithm for a single pattern (*EDSM* problem [7]).

▶ **Corollary 8.** *Algorithm Multi-EDSM solves the EDSM problem in an on-line manner in time* $\mathcal{O}(N \cdot \lceil \frac{m}{w} \rceil)$*. Algorithm Multi-EDSM requires pre-processing time and space* $\mathcal{O}(m)$*.*

▶ **Example 9.** Given an ED string $\tilde{T}$ as shown below, we wish to search for the patterns in set $P = \{\texttt{ATAT}, \texttt{TAGA}\}$ of total length $M = 8$. A collection I of σ bit vectors are created during the Shift-And pre-processing stage marking the positions of the letters in Σ of the concatenated patterns. We also build Occ-Vector$_P$. Note that the bit vectors are read from right to left and recall that Occ-Vector$_P$ subtracts 1 from the index positions.

$$\tilde{T} = \left\{ \begin{matrix} \texttt{AT} \\ \texttt{A} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{AT} \\ \texttt{TA} \end{matrix} \right\} \cdot \left\{ \begin{matrix} \texttt{TTTA} \\ \texttt{AGA} \end{matrix} \right\}$$

$I_{\texttt{A}}$	1010 0101
$I_{\texttt{C}}$	0000 0000
$I_{\texttt{G}}$	0100 0000
$I_{\texttt{T}}$	0001 1010

The algorithm starts with $\tilde{T}[0]$, skips the Shift-And search of the strings in the segment because they are too short, and computes bit vector $B = \texttt{0001 0011} = \textsc{Overlaps}(\tilde{T}[0])$ on line 8. The Overlaps function memorises the prefixes starting in the current segment using B.

Then, the next segment is considered; on line 10 we compute the bit vector $B_1 = \texttt{0011 0011} = \textsc{Overlaps}(\tilde{T}[1])$. The next step is to check each string $s \in \tilde{T}[1]$ and after doing Multi-Shift-And-Search(AT, B_2) with state B (from $\tilde{T}[0]$) it discovers a match for $P[0]$ and reports it. This time the Shift-And search function is called regardless of the length of s because it is used to find whole patterns as well as suffixes of patterns that began in the previous segments. The function completes the suffix by matching AT at positions $I_{\texttt{A}}[2]$ and $I_{\texttt{T}}[3]$ to spell out $P[0]$.

Then we consider how to handle infixes on line 19. We only need to process strings short enough to be considered as infixes of a pattern and we query them with the Occ-Vector$_P$ data structure to mark the positions where each infix starts. Calling Occ-Vector$_P$(AT) finds infix position 2 and returns 0000 0010. So $B_3 = \texttt{0000 0010} = B$ & 0000 0010, thus maintaining the active search state. The Left-Shift function does bitwise left-shift of B_3 by $|s|$ positions whilst ensuring no 1s end up at or beyond the ending position of each pattern in the set. What bits remain as 1s are bitwise-ORed with B_1 to update the state to maintain partial search states, but in this case, the 1 is shifted too far and B_1 remains unchanged.

In the next iteration, we do Multi-Shift-And-Search(TA, B_2) yielding no match. Then we call Occ-Vector$_P$(TA) which finds infix position 1 and returns 0000 0001 which we bitwise-AND with B to take $B_3 = \texttt{0000 0001}$. Then Left-Shift is performed on B_3 and bitwise-ORing its result with B_1 makes $B_1 = \texttt{0011 0111}$ because no boundaries are crossed. Having searched all the strings in the segment, we save the state of the search to B on line 22 and observe that we have thus far matched ATA of $P[0]$ spanning across $\tilde{T}[0]$ and $\tilde{T}[1]$.

Now we go ahead and search the final segment $\tilde{T}[2]$ of our example. We compute $B_1 = \texttt{0010 0001} = \textsc{Overlaps}(\tilde{T}[2])$ first and then by calling Multi-Shift-And-Search(TTTA, B_2) with the state $B_2 = B$ from the previous segment, we complete the partial match and report finding $P[0]$ in this segment. Calling this function again for the next string in the segment, Multi-Shift-And-Search(AGA, B_2) also completes the suffix for $P[1]$, and we report it.

On the very last line of the algorithm we save the final search state of the segment to bit vector B, ready for the next on-line letter to be sent to the SEARCH function.

5 Experiments

Multi-EDSM code was written in C++ and compiled with g++ version 5.4.0 at optimization level 3 (-O3) and scripts were written in Python 2.7. A simpler version of the OCC-VECTOR$_P$ data structure has been implemented in which the user can set a memory limit to be used by the program and then the analogous number of bit vectors are stored in the explicit nodes of the suffix tree that are closer to the root. This is because the vast majority of the variation strings to be queried in real datasets are rather short.

The following experiments were conducted on a desktop computer using one core of Intel® Core™ i7-2600S CPU at 2.8GHz and 8GB of RAM under 64-bit GNU/Linux. The Multi-EDSM application code and generated experimental datasets and scripts are licensed under the GNU General Public License (GPL-3.0); they are all freely available from `https://github.com/webmasterar/multi-edsm`.

5.1 Time Performance

To test the time performance of Multi-EDSM application, we devised two experiments. In both experiments we set the memory limit of Multi-EDSM to 4GB for the program to use as much memory as necessary.

In Figure 2a we measured the processing time when searching a randomly-generated fixed ED text of length $n = 1600000$ ($N = 5700610$) over the DNA alphabet with randomly-generated pattern sets doubling in length from length $M = 1600$ to $M = 102400$. The text searched contains 10% degenerate segments and within each degenerate segment there are 2 to 10 random strings of length 1 to 10 each. Similarly, in Figure 2b we measured the processing time when searching randomly generated ED texts doubling in size from $N = 100000$ to $N = 6400000$ with a fixed set of randomly-generated patterns of length $M = 3000$. The text had 10% degenerate positions representing single-base substitutions. (Uniform distribution has been used in all randomizations.)

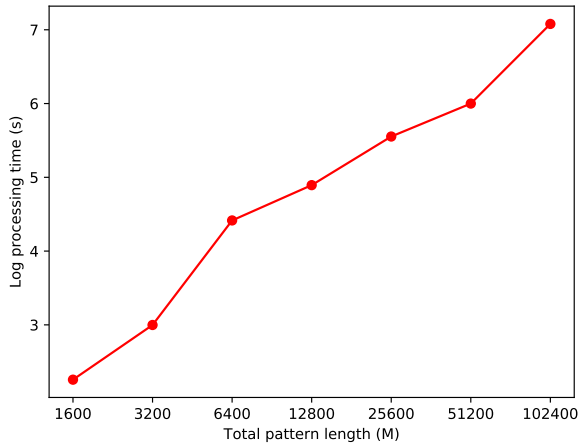
As can be seen from the charts, the change in performance, whether it be an increase in the patterns total length M or the total text size N, causes a linear increase in processing time, which conforms to our theoretical findings (Theorem 7).

5.2 Comparison to EDSM-BV

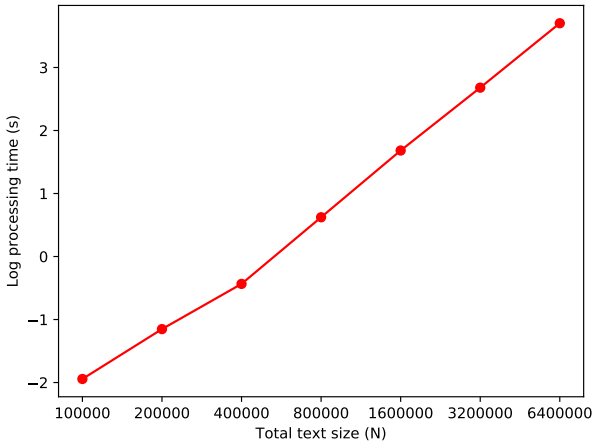
To test the performance of Multi-EDSM compared against EDSM-BV [7] we searched the same randomly-generated ED text of length $n = 1600000$ ($N = 5700610$) mentioned above against multiple sets of randomly-generated patterns of length 40 each. First we tested with a single pattern of length 40 and then the number of patterns in each set was incremented in steps of 10 from 10 to 100 patterns of length 40 each. EDSM-BV is only able to search one pattern at a time so we searched each pattern in a set individually and summed-up the total time spent. We see from the chart in Figure 3 that for a single pattern the EDSM-BV algorithm is fast, but it becomes immediately clear that for dictionary searching of even a handful of patterns, Multi-EDSM becomes orders of magnitude faster.

5.3 Real Application

We designed a three stage pipeline for determining the validity of MAWs discovered in the human genome. We obtained the GRCh37 chromosome sequences from *Ensembl* [10]



(a) Processing time with increasing patterns total length on a fixed ED text of length $n = 1600000$ ($N = 5700610$).



(b) Processing time with increasing ED total text size on a fixed set of patterns of total length $M = 3000$.

Figure 2 Time performance of Multi-EDSM.

and the associated phase 3 VCF files were obtained from the *1000 Genomes project* [23] on-line repositories. Phase 3 of the 1000 Genomes project contains 2504 individuals from 26 populations whose variants are encoded in VCF files. The first step in the pipeline was to use *emMAW* [11] to extract MAWs of length between 3 and 12 from a concatenated file of the 22 autosomes and two sex chromosomes. The filtered list of MAWs contained 161565 patterns with combined length $M = 1937789$. The second stage was to use the tool *EDSO* to combine the reference chromosome sequences and the variants in the VCF files into searchable ED string (EDS) format files. Then Multi-EDSM was used to search each of the EDS files against the MAW patterns to produce a list of tuples marking the position of the match and pattern id. The final stage was validation. A script was written to validate each match, verifying a MAW genuinely exists for an individual at the identified position in the chromosome. We have found that for each chromosome more than half the MAWs discovered

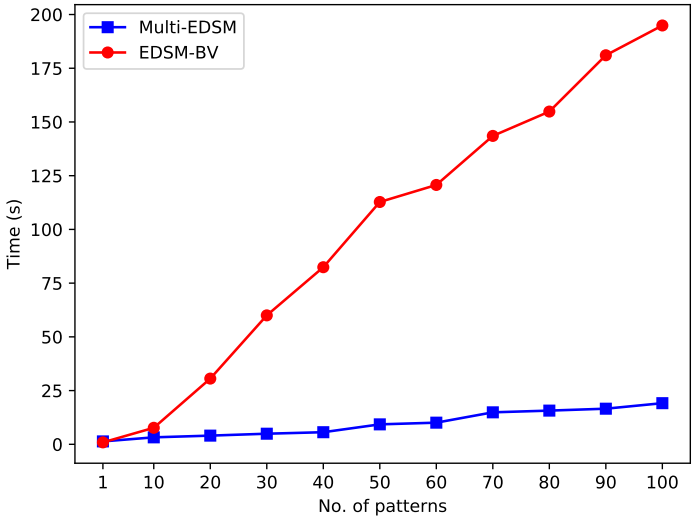


Figure 3 Elapsed-time comparison of Multi-EDSM and EDSM-BV with an ED text of total size $N = 5700610$ and sets of randomly-generated patterns of length 40 each.

Table 1 Ebola sequences *absent* from human reference genome but *present* in human pan-genome.

id	sequence	position	variant id	sample id	ethnicity
RAW1	`TTTCGCCCGACT`	6:93819539	rs569027564	NA18606	Han Chinese
RAW2	`TACGCCCTATCG`	1:74075482	rs578167440	HG02146	Peruvian
RAW3	`CCTACGCGCAAA`	15:71003880	rs564150197	HG03598	Bengali

using Multi-EDSM exist in one or more individuals and are subsequently disqualified, i.e. they are not really MAWs. Our compiled summary of the results show that 73% of MAWs were disqualified, leaving only 43722 of 161565 potential MAWs remaining.

We applied the results of this pipeline to validate the work of Silva et. al. in [21] to identify MAWs in Ebola virus genomes that are absent from the human genome. They identify three MAWs of length 12, called RAW1, RAW2 and RAW3, that are not present in the *reference* human genome sequence. These MAWs could be used to verify an Ebola infection in a patient. However, we discovered from our results that each of the three MAWS do in fact occur in one or more individuals in the 1000 Genomes dataset, although indeed they are not that common. This means that they cannot be used as perfect identifiers for Ebola virus infection and perhaps longer unique MAWs should be used instead. In Table 1 we list the position of the discovery of each MAW as well as information about the variant and the id of one individual they occur in.

6 Final Remarks

It would be relevant [9] to investigate the problem of dictionary matching in elastic-degenerate texts under the Hamming or edit distance models (see [3] for a single pattern).

References

1 Stephen Alstrup, Jens P. Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Inf. Process. Lett.*, 64(4):161–164, 1997. doi:10.1016/S0020-0190(97)00170-1.

2 Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the Burrows-Wheeler transform. *Bioinformatics*, 32(4):497–504, 2016.

3 Giulia Bernardini, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone. Pattern matching on elastic-degenerate text with errors. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Palermo, Italy, September 26-29, 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 2017. doi:10.1007/978-3-319-67428-5_7.

4 M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

5 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. URL: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5208, doi:10.1109/SFCS.1997.646102.

6 Travis Gagie, Danny Hermelin, Gad M. Landau, and Oren Weimann. Binary jumbled pattern matching on trees and tree-like structures. *Algorithmica*, 73(3):571–588, 2015. doi:10.1007/s00453-014-9957-6.

7 Roberto Grossi, Costas S. Iliopoulos, Chang Liu, Nadia Pisanti, Solon P. Pissis, Ahmad Retha, Giovanna Rosone, Fatima Vayani, and Luca Versari. On-Line Pattern Matching on Similar Texts. In Juha Kärkkäinen, Jakub Radoszewski, and Wojciech Rytter, editors, *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, volume 78 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CPM.2017.9.

8 Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms for Molecular Biology*, 11:3, 2016.

9 Lin Huang, Victoria Popic, and Serafim Batzoglou. Short read alignment with populations of genomes. *Bioinformatics*, 29(13):361–370, 2013.

10 T. Hubbard, D. Barker, E. Birney, G. Cameron, Y. Chen, L. Clark, T. Cox, J. Cuff, V. Curwen, T. Down, R. Durbin, E. Eyras, J. Gilbert, M. Hammond, L. Huminiecki, A. Kasprzyk, H. Lehvaslaiho, P. Lijnzaad, C. Melsopp, E. Mongin, R. Pettett, M. Pocock, S. Potter, A. Rust, E. Schmidt, S. Searle, G. Slater, J. Smith, W. Spooner, A. Stabenau, J. Stalker, E. Stupka, A. Ureta-Vidal, I. Vastrik, and M. Clamp. The Ensembl genome database project. *Nucleic Acids Research*, 30(1):38–41, 2002. doi:10.1093/nar/30.1.38.

11 Alice Héliou, Solon P. Pissis, and Simon J. Puglisi. emMAW: computing minimal absent words in external memory. *Bioinformatics*, 33(17):2746–2749, 2017. doi:10.1093/bioinformatics/btx209.

12 Costas S. Iliopoulos, Ritu Kundu, and Solon P. Pissis. Efficient pattern matching in elastic-degenerate texts. In Frank Drewes, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, volume 10168 of *Lecture Notes in Computer Science*, pages 131–142, 2017. doi:10.1007/978-3-319-53733-7_9.

13 Paul Julian Kersey, James E. Allen, Irina Armean, Sanjay Boddu, Bruce J. Bolt, Denise Carvalho-Silva, Mikkel Christensen, Paul Davis, Lee J. Falin, Christoph Grabmueller,

Jay C. Humphrey, Arnaud Kerhornou, Julia Khobova, Naveen K. Aranganathan, Nicholas Langridge, Ernesto Lowy, Mark D. McDowall, Uma Maheswari, Michael Nuhn, Chuang Kee Ong, Bert Overduin, Michael Paulini, Helder Pedro, Emily Perry, Giulietta Spudich, Electra Tapanari, Brandon Walts, Gareth Williams, Marcela K. Tello-Ruiz, Joshua C. Stein, Sharon Wei, Doreen Ware, Daniel M. Bolser, Kevin L. Howe, Eugene Kulesha, Daniel Lawson, Gareth Maslen, and Daniel M. Staines. Ensembl genomes 2016: more genomes, more complexity. *Nucleic Acids Research*, 44(Database-Issue):574–580, 2016.

14 Sorina Maciuca, Carlos del Ojo Elias, Gil McVean, and Zamin Iqbal. A natural encoding of genetic variation in a Burrows-Wheeler transform to enable mapping and genome inference. In Martin C. Frith and Christian Nørgaard Storm Pedersen, editors, *Algorithms in Bioinformatics - 16th International Workshop, WABI 2016, Aarhus, Denmark, August 22-24, 2016. Proceedings*, volume 9838 of *Lecture Notes in Computer Science*, pages 222–233. Springer, 2016. doi:10.1007/978-3-319-43681-4_18.

15 Joong Chae Na, Hyunjoon Kim, Heejin Park, Thierry Lecroq, Martine Léonard, Laurent Mouchard, and Kunsoo Park. FM-index of alignment: A compressed index for similar strings. *Theor. Comput. Sci.*, 638:159–170, 2016.

16 Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences*. Cambridge University Press, 2002.

17 Ngan Nguyen, Glenn Hickey, Daniel R. Zerbino, Brian J. Raney, Dent Earl, Joel Armstrong, W. James Kent, David Haussler, and Benedict Paten. Building a pan-genome reference for a population. *Journal of Computational Biology*, 22(5):387–401, 2015.

18 Nadia Ben Nsira, Mourad Elloumi, and Thierry Lecroq. On-line string matching in highly similar DNA sequences. *Mathematics in Computer Science*, 11(2):113–126, 2017. doi:10.1007/s11786-016-0280-2.

19 Nadia Ben Nsira, Thierry Lecroq, and Mourad Elloumi. A fast Boyer-Moore type pattern matching algorithm for highly similar sequences. *IJDMB*, 13(3):266–288, 2015. doi:10.1504/IJDMB.2015.072101.

20 Siavash Sheikhizadeh, M. Eric Schranz, Mehmet Akdel, Dick de Ridder, and Sandra Smit. Pantools: representation, storage and exploration of pan-genomic data. *Bioinformatics*, 32(17):487–493, 2016.

21 Raquel M. Silva, Diogo Pratas, Luísa Castro, Armando J. Pinho, and Paulo J. S. G. Ferreira. Three minimal sequences found in Ebola virus genomes and absent from human DNA. *Bioinformatics*, 31(15):2421–2425, 2015. doi:10.1093/bioinformatics/btv189.

22 Jouni Sirén. Indexing variation graphs. In Sándor P. Fekete and Vijaya Ramachandran, editors, *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2017, Barcelona, Spain, Hotel Porta Fira, January 17-18, 2017.*, pages 13–27. SIAM, 2017. doi:10.1137/1.9781611974768.2.

23 The 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.

24 The Computational Pan-Genomics Consortium. Computational pan-genomics: status, promises and challenges. *Briefings in Bioinformatics*, pages 1–18, 2016.