Unary Words Have the Smallest Levenshtein k-Neighbourhoods

Panagiotis Charalampopoulos
Department of Informatics, King's College London, UK
Institute of Informatics, University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Solon P. Pissis
CWI, Amsterdam, The Netherlands
Vrije Universiteit, Amsterdam, The Netherlands
ERABLE Team, Lyon, France
solon.pissis@cwi.nl

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Poland
Samsung R&D, Warsaw, Poland
jrad@mimuw.edu.pl

Tomasz Waleń
Institute of Informatics, University of Warsaw, Poland
walen@mimuw.edu.pl

Wiktor Zuba
Institute of Informatics, University of Warsaw, Poland
w.zuba@mimuw.edu.pl

Abstract

The edit distance (a.k.a. the Levenshtein distance) between two words is defined as the minimum number of insertions, deletions or substitutions of letters needed to transform one word into another. The Levenshtein k-neighbourhood of a word w is the set of words that are at edit distance at most k from w. This is perhaps the most important concept underlying BLAST, a widely-used tool for comparing biological sequences. A natural combinatorial question is to ask for upper and lower bounds on the size of this set. The answer to this question has important algorithmic implications as well. Myers notes that "such bounds would give a tighter characterisation of the running time of the algorithm" behind BLAST. We show that the size of the Levenshtein k-neighbourhood of any word of length n over an arbitrary alphabet is not smaller than the size of the Levenshtein k-neighbourhood of a unary word of length n, thus providing a tight lower bound on the size of the Levenshtein k-neighbourhood. We remark that this result was posed as a conjecture by Dufresne at WCTA 2019.

1   Introduction

BLAST (Basic Local Alignment Search Tool) is a widely-used tool for comparing biological sequences such as the amino-acid sequences of proteins or the nucleotides of DNA or RNA sequences. A BLAST search enables to compare a subject sequence, called a *query*, against a database of sequences to identify the ones that resemble the query sequence above a certain threshold. The paper describing BLAST [1] is one of the most highly cited papers in science.

According to Myers [8], the most important algorithmic idea underlying BLAST is that of searching for exact matches to words in the neighbourhood of fixed-length fragments selected from the query sequence. We call these fragments *words*. Let δ be a sequence comparison measure that given two words v and w returns a numeric measure $\delta(v, w)$ of the degree to which the two words differ. Given a word w, the k-neighbourhood of w with respect to δ is the set of all words whose best alignment with w under measure δ is no more than k. The most widely-used case is where δ is the *edit distance* (a.k.a. the *Levenshtein distance*), which is the minimum number of insertions, deletions or substitutions of letters needed to transform one word into another [6]. When δ is the Levenshtein distance, we call this neighbourhood the *Levenshtein k-neighbourhood* of w and we denote it by $N_{k,\Sigma}(w)$, where Σ is the considered alphabet. We provide an example below.

▶ **Example 1** (Levenshtein k-neighbourhood). Let $w = baab$, $k = 1$ and $\Sigma = \{a, b\}$. Then $N_{1,\Sigma}(baab)$ is:

$$\{bbab, bbaab, babb, bab, babab, baab, baabb, baaba, baa, baaa, baaab, abaab, aab, aaab\}.$$

From an algorithmic point of view, the most natural question is how we can generate the Levenshtein k-neighbourhood in time that is proportional to the size of the neighbourhood. In fact, this is the core computational task underlying BLAST. Myers described an algorithm for generating a condensed version of this neighbourhood efficiently (see [8] for more details). Another natural question is how we can compute the size of the Levenshtein k-neighbourhood. Touzet gave an algorithm for computing $|N_{k,\Sigma}(w)|$ for a word w of length n over an alphabet Σ that works in time linear in n but exponential in k [11]. This algorithm is based on a variant of the so-called *Universal Levenshtein Automaton* [7], which in turn is based on the *Levenshtein automaton* of w: the non-deterministic finite automaton recognising all words which are at Levenshtein distance at most k from w. For other related works, see [2, 3, 9, 10].

From a combinatorial point of view, the most natural question asks for upper and lower bounds on the size of the Levenshtein k-neighbourhood. Myers provided recurrences for counting the number of distinct sequences of k edit operations that one could perform on a given word and notes that "such bounds would give a tighter characterisation of the running time of the algorithm" behind BLAST [8]. A word is called *unary* if it consists of a single element of Σ. The main result of this work can be formally stated as follows.

▶ **Theorem 2.** *Let $a \in \Sigma$ be an arbitrary element of alphabet Σ. For any positive integers n and k, we have $|N_{k,\Sigma}(a^n)| < |N_{k,\Sigma}(w)|$, for any non-unary word w of length n.*

The course of our proof is to construct, for every word $u \in N_{k,\Sigma}(a^n)$, a distinct word $u' \in N_{k,\Sigma}(w)$ that can be obtained by a similar sequence of edit operations. In particular, we show that, for any n, k, and Σ,

$$|N_{k,\Sigma}(a^n)| = \sum_{i=0}^{k} \sum_{j=i-k}^{k} \binom{n+j}{i} (\sigma - 1)^i$$

is the size of the smallest Levenshtein k-neighbourhood of a word of length n, where $a \in \Sigma$ and $\sigma = |\Sigma|$. We remark that our main result was posed as a conjecture by Dufresne in [5].

Organisation of the Paper. The basic definitions and notation used throughout are introduced in Section 2. In Section 3, we present the main result of this work for binary alphabets – apart from the strictness of the inequality. We then generalise this result to arbitrary alphabets in Section 4 and prove the strictness of the inequality directly in this more general case. We conclude this paper in Section 5 with some final remarks.

2 Preliminaries

An *alphabet* Σ is a finite non-empty set of size $\sigma = |\Sigma|$ whose elements are called *letters*. A *word* over Σ is a sequence of letters from Σ. We call a word w *unary* if it consists of a single letter of Σ and *non-unary* if it consists of at least two letters of Σ. Σ^n denotes the set of words of length n over Σ and Σ^* denotes the set of finite words over Σ. For a word w, by $|w|$ we denote its length, and by $w[i]$, for $i = 1, \ldots, |w|$, we denote its subsequent letters. The word of length 0 is the *empty word*, which we denote by ε.

We consider the following elementary edit operations: insertion, deletion, and substitution. For two words x and y, we define the *edit distance* (a.k.a. the *Levenshtein distance*) as the minimum number of edit operations that transform x to y, and we denote it by $\mathsf{Lev}(x, y)$. The function Lev is then a metric on Σ^* [4].

Given a word w, an alphabet Σ, and a positive integer k, we define $N_{k,\Sigma}(w)$ as the set of all words in Σ^* that are at Levenshtein distance at most k from w. Formally, we have that

$$N_{k,\Sigma}(w) = \{v \in \Sigma^* : \mathsf{Lev}(v, w) \leq k\}.$$

We call $N_{k,\Sigma}(w)$ the *Levenshtein* (k, Σ)*-neighbourhood* of w.

For any binary alphabet Σ, we define the *complement* of a word w over Σ as the word obtained by substituting $w[i]$ for letter $a \neq w[i]$, with $a \in \Sigma$, for all $i = 1, \ldots, |w|$. We denote the complement of w by $\overline{w}$ and we call a single such substitution operation a *flip*.

3 Main Result for Binary Alphabets

In this section we consider $\Sigma = \{a, b\}$, write $N_k(w)$ and refer to Levenshtein k-neighbourhood for simplicity. We present the main result but do not show the strictness of the inequality. We generalise this result to an arbitrary alphabet Σ and show the strictness in Section 4.

Let $N_k^j(w) = \{u \in N_k(w) : |u| = j\}$. Further let $\#_a(u)$ denote the number of a's in word u. We illustrate the main ideas of our approach on a simple case and first consider the words of the neighbourhood that are of length at most n.

▶ **Observation 3.** *Any $u \in N_k(a^n)$ with $|u| \leq n$ can be obtained from a^n by the following sequence of at most k edit operations: $n - |u|$ deletions of a's in the beginning of a^n followed by a sequence of $|u| - \#_a(u)$ flips.*

▶ **Example 4.** Let $w = aaaa$ and $k = 2$. Then $u = aba \in N_2(aaaa)$ can be obtained from $aaaa$ by deleting $n - |u| = 1$ letter a to obtain aaa and then by $|u| - \#_a(u) = 1$ flip to obtain aba.

Intuitively, the size of the set $N_k^j(a^n)$ is equal to the number of subsets of $\{1, \ldots, j\}$ of size at most $k - (n - j)$; $n - j$ is the number of deletions and $k - (n - j)$ the number of flips.

▶ **Lemma 5.** *If $j \leq n$, then $|N_k^j(a^n)| \leq |N_k^j(w)|$ for all $w \in \Sigma^n$.*

Proof. We use the characterisation of Observation 3. Let us argue why $|N_k^j(w)|$, for any word w of length n, is at least as big as $|N_k^j(a^n)|$. Consider the following procedure applied on word w: the deletion of the first $n - j$ letters of w followed by the flipping of at most $k - (n - j)$ letters. Clearly all words obtained by this procedure are distinct. This procedure thus gives us a subset of $N_k^j(w)$ that is of size equal to $|N_k^j(a^n)|$.   ◀

Let us now consider the case of the words of the neighbourhood that have length greater than n. In particular, we denote the neighbourhood $\bigcup\limits_{j > n} N_k^j(w)$ of these words by $N_k^{>n}(w)$ and thus $N_k^{\le n}(w) = N_k(w) \setminus N_k^{>n}(w)$. For a word $u \in N_k^{>n}(a^n)$, we distinguish between two cases depending on the number of a's in u:

- Case 1: $\#_a(u) \ge n$,
- Case 2: $\#_a(u) < n$.

The following observation states that in each case the word $u \in N_k^{>n}(a^n)$ can be obtained by a restricted sequence of edit operations.

▶ **Observation 6.**
(1) *Any $u \in N_k^{>n}(a^n)$ with $\#_a(u) \ge n$ can be obtained from a^n by the following sequence of at most k edit operations: $\#_a(u) - n$ insertions of a's in the beginning of a^n followed by a sequence of $|u| - \#_a(u)$ insertions of b's.*
(2) *Any $u \in N_k^{>n}(a^n)$ with $\#_a(u) < n$ can be obtained from a^n by the following sequence of at most k edit operations: $n - \#_a(u)$ flips followed by a sequence of $|u| - n$ insertions of b's. The insertions can be restricted to the part of the word after the rightmost flip.*

▶ **Example 7.** Let $w = aaaa$ and $k = 2$. For Case 1, $u = aaaaba \in N_2^{>n}(aaaa)$ with $\#_a(u) = 5 \ge n = 4$ can be obtained by $\#_a(u) - n = 1$ insertion of a in the beginning of $aaaa$ to obtain $aaaaa$ and then by $|u| - \#_a(u) = 1$ insertion of b to obtain $aaaaba$. For Case 2, $u = aabab \in N_2^{>n}(aaaa)$ with $\#_a(u) = 3 < n = 4$ can be obtained by $n - \#_a(u) = 1$ flip to obtain $aaba$ and then by $|u| - n = 1$ insertion of b to the right of the flip to obtain $aabab$.

Intuitively, in Case 1, we insert the relevant number of a's to reach $\#_a(u)$ because we have fewer a's than needed, and then insert the relevant number of b's. In Case 2, we flip the relevant number of a's to go down to $\#_a(u)$ because we have more a's than what is needed, and then insert the remaining b's to the right of the rightmost flip.

Proof Strategy.   Let u be an arbitrary element of $N_k(a^n)$, for some positive integers n and k. We define a function $f_u : \Sigma^n \to \Sigma^*$, such that:
1. $f_u(w) \in N_k(w)$, for all $w \in \Sigma^n$; and
2. Given w and $f_u(w)$ we can retrieve u.
Such an f_u directly yields the desired bound (apart from the strictness) since it implies that for any word w we cannot have $f_u(w) = f_{u'}(w)$ for $u, u' \in N_k(a^n), u \ne u'$. In particular, we have that $|N_k(a^n)| \le |N_k(w)|$ for any $w \in \Sigma^n$; see Table 1 for a complete example.

Note that for $|u| \le n$ we already used the same idea to lower bound $|N_k^{\le n}(w)|$ by $|N_k^{\le n}(a^n)|$. Indeed, we implicitly defined $f_u(w)$ for $u \in N_k^{\le n}(a^n)$ that consists in removing the first $n - |u|$ letters of w, resulting in a word w', and then flipping the letters of w' at positions j where $u[j] = b$ (see Lemma 5).

■ **Table 1** Let $n = 3$, $w = aab$ and $k = 2$. The table presents an assignment f_u from every word in $N_2(a^3)$ to a different word in $N_2(aab)$ that is used in the proof of the main result. Note, however, that as per Theorem 2 there is at least one more word in $N_2(aab)$, namely, the word a.

(a) f_u for $u \in N_2^{\leq 3}(a^3)$.

$u \in N_2^{\leq 3}(a^3)$	$f_u(aab)$
a	b
aa	ab
ab	aa
ba	bb
aaa	aab
aab	aaa
aba	abb
abb	aba
baa	bab
bab	baa
bba	bbb

(b) f_u for $\#_a(u) < 3$ (Case 2).

$u \in N_2^{>3}(a^3)$	$f_u(aab)$
$aabb$	$aaaa$
$abab$	$abba$
$abba$	$abbb$
$baab$	$baba$
$baba$	$babb$
$bbaa$	$bbab$

(c) f_u for $\#_a(u) \geq 3$ (Case 1).

$u \in N_2^{>3}(a^3)$	$f_u(aab)$
$aaaa$	$aaab$
$aaab$	$aaba$
$aaba$	$aabb$
$abaa$	$abab$
$baaa$	$baab$
$aaaaa$	$aaaab$
$aaaab$	$aaaba$
$aaaba$	$aaabb$
$aaabb$	$aabaa$
$aabaa$	$aabab$
$aabab$	$aabba$
$aabba$	$aabbb$
$abaaa$	$abaab$
$abaab$	$ababa$
$ababa$	$ababb$
$abbaa$	$abbab$
$baaaa$	$baaab$
$baaab$	$baaba$
$baaba$	$baabb$
$babaa$	$babab$
$bbaaa$	$bbaab$

Definition of f_u. Let us start by introducing the following edit operation on a non-empty word w. It takes as input parameters an integer $j \in [1, |w|]$ and a positive integer t.

$\mathsf{ins\text{-}diff}(w, j, t) :$ inserts a block of length t of letters equal to $\overline{w[j]}$ after the letter $w[j]$ (1)

See Figure 1 for an illustration of operation $\mathsf{ins\text{-}diff}(w, j, t)$.

$$\begin{array}{ccccccc} b & b & b & a & a & b & b \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ a & a & a & b & b & a & a \end{array}$$
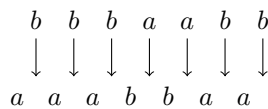
■ **Figure 1** For every position j of $w = aaabbaa$ (bottom), the letter (top) of which a block inserted by $\mathsf{ins\text{-}diff}(w, j, 1)$ after position j would comprise.

In what follows, we assume that all insertions are with respect to the original indices of w. This can be achieved, for example, by performing insertions in a right-to-left manner when they are given as an ordered batch. Before providing the definition of f_u, we define two auxiliary operators g_x and h_x, for a word x.

Let us start with g_x. For any word x starting with a, we define an operator g_x that can be applied to any word y such that $|y| = \#_a(x)$. Intuitively, to construct word $g_x(y)$, the letters

of y get in $g_x(y)$ the positions that letters a possess in x, and for every *maximal* block of b's in between in x, i.e. a block consisting of only b's that is neither preceded nor succeeded by a b, we apply an ins-diff operation on y. Specifically, we define $g_x(y) = v$ as follows: starting with $v = y$, for each maximal block $x[r \mathinner{..} r + t - 1]$ of b's in x, with $\#_a(x[1 \mathinner{..} r - 1]) = m$, perform ins-diff$(v, m, t)$. Note that $|g_x(y)| = |x|$; see Example 8.

▶ **Example 8.** Let $x = aababbababaab$ and $y = aaabbaa$; note that $\#_a(x) = 7 = |y|$. We have $g_x(y) = v = aababbbabaaab$; see also the figure below and recall that we perform this procedure from right to left. Starting from $v = y$, for the first maximal block $x[r \mathinner{..} r+t-1] = x[13 \mathinner{..} 13] = b$ with $\#_a(x[1 \mathinner{..} r - 1]) = \#_a(x[1 \mathinner{..} 12]) = m = 7$, we perform ins-diff$(v, 7, 1)$, which constructs $aaabbaab$. For the next maximal block $x[r \mathinner{..} r + t - 1] = x[10 \mathinner{..} 10] = b$ with $\#_a(x[1 \mathinner{..} r - 1]) = \#_a(x[1 \mathinner{..} 9]) = m = 5$, we perform ins-diff$(v, 5, 1)$, which constructs $aaabbaaab$. For the next maximal block $x[r \mathinner{..} r+t-1] = x[8 \mathinner{..} 8] = b$ with $\#_a(x[1 \mathinner{..} r-1]) = \#_a(x[1 \mathinner{..} 7]) = m = 4$, we perform ins-diff$(v, 4, 1)$, which constructs $aaababaaab$, and so on.

$$
\begin{array}{llllllllllllll}
x: & \boxed{a} & \boxed{a} & b & \boxed{a} & b & b & \boxed{a} & b & \boxed{a} & b & \boxed{a} & \boxed{a} & b \\
y: & a & a & & a & & & b & & b & & a & a & \\
g_x(y): & a & a & b & a & b & b & b & a & b & a & a & a & b
\end{array}
$$

For any word x, we also define an operator h_x that takes as input a word y of length $|x|$ and flips its letters on positions in which x has b's.

We are now in a position to define $f_u(w)$, for all $w \in \Sigma^n$. Recall that $u \in N_k^{>n}(a^n)$. We have the following two cases for f_u.

Case 1: $\#_a(u) \geq n$. Let us split u in its shortest suffix s that contains n a's and the remaining (possibly empty) prefix p. We then define $f_u(w)$ for words u and w in this case as follows (see Example 9):

$$f_u(w) = p \cdot g_s(w). \tag{2}$$

▶ **Example 9.** Let $w = aaabbaa$ and $k = 4$; note that $n = 7$. If $u = abaaaaaabaa$, then $\#_a(u) = 9 \geq n$, so we are in Case 1. We have $p = aba$ and $s = aaaaabaa$ is the shortest suffix that contains $n = 7$ occurrences of the letter a. Then $f_u(w)$ is constructed by concatenating p with the word $g_s(w)$ as shown in the figure below.

$$
\begin{array}{lllllllllll}
u: & a & b & a & \boxed{a} & \boxed{a} & \boxed{a} & \boxed{a} & \boxed{a} & b & \boxed{a} & \boxed{a} \\
w: & & & & a & a & a & b & b & & a & a \\
f_u(w): & a & b & a & a & a & a & b & b & a & a & a
\end{array}
$$

Case 2: $\#_a(u) < n$. In this case we split u in its shortest suffix s' that contains $|u| - n$ b's and the remaining prefix p'. Note that p' is always non-empty. We then define $f_u(w)$ for words u and w in this case as follows (see Example 10):

$$f_u(w) = h_{p'}(w') \cdot g_x(w)[|p'| + 1 \mathinner{..} |u|], \text{ where } x = a^{|p'|}s' \text{ and } w' = w[1 \mathinner{..} |p'|]. \tag{3}$$

In particular, $\#_a(x) = |x| - \#_b(x) = |u| - (|u| - n) = n$, and so applying g_x is well-defined.

▶ **Example 10.** Let $w = aaabbaa$ and $k = 4$; note that $n = 7$. If $u = aababbaab$, then $\#_a(u) = 5 < n$, so we are in Case 2. We have $p' = aabab$ and $s' = baab$ is the shortest suffix

that contains $|u| - n = 2$ occurrences of letter b. Then $f_u(w)$ is constructed by concatenating two words: the first one is $h_{p'}(w')$, where $w' = w[1 \mathinner{.\,.} |p'|]$; and the second one is composed of the final $|s'|$ letters of $g_x(w)$, where x is the word obtained from u by changing the first $n - \#_a(u) = 2$ occurrences of b to a as shown in the figure below.

$u:$	a	a	b	a	b	b	a	a	b
$x:$	a	a	a	a	a	b	a	a	b
$w:$	a	a	a	b	b		a	a	
$g_x(w):$	a	a	a	b	b	a	a	a	b
$h_{p'}(w'):$	a	a	b	b	a				
$f_u(w):$	a	a	b	b	a	a	a	a	b

In Table 1 we provide a complete example of applying f_u, for each $u \in N_k(a^3)$, to $w = aab$. Let us now show the following fact.

▶ **Fact 11.** $|f_u(w)| = |u|$ and $\mathsf{Lev}(w, f_u(w)) \leq \mathsf{Lev}(a^n, u)$.

Proof. The first part can be readily verified. As for the second part, one can obtain $f_u(w)$ from w by the same sequence of edit operations (types and positions) that yield u from a^n, according to Observation 6.   ◀

We next prove the main lemma on which our main result relies. A pseudocode implementing the algorithm used in the proof of this lemma can be found as Algorithm 1. Note that by considering a's as 0's and b's as 1's, we have that $h_x(y) \otimes y = x$, where $\otimes$ denotes the XOR operation. Consider, for instance, $x = aabab$, $y = aaabb$ and $h_x(y) = aabba$.

■ **Algorithm 1** $\textsc{Retrieve}(w, f_u(w))$ for $\Sigma = \{a, b\}$.

Input: Two words w and $f_u(w)$.
Output: A word u.

1: $u \leftarrow \varepsilon$
2: $k_1 \leftarrow |w|$
3: $k_2 \leftarrow |f_u(w)|$
4: **while** $k_2 > k_1$ **and** $k_1 > 0$ **do**
5:     **if** $w[k_1] = f_u(w)[k_2]$ **then**
6:         $k_1 \leftarrow k_1 - 1$
7:         $\mathsf{prepend}(a, u)$
8:     **else**
9:         $\mathsf{prepend}(b, u)$
10:     $k_2 \leftarrow k_2 - 1$
11: **if** $k_1 = 0$ **then**   \\ $k_2 > 0$; Case 1 with $p \neq \varepsilon$
12:     $\mathsf{prepend}(f_u(w)[1 \mathinner{.\,.} k_2], u)$
13: **else**   \\ $k_1 = k_2$; Case 2 (or Case 1 with $p = \varepsilon$)
14:     $\mathsf{prepend}(w[1 \mathinner{.\,.} k_1] \otimes f_u(w)[1 \mathinner{.\,.} k_2], u)$
15: **return** u

▶ **Lemma 12.** *Let u be an arbitrary element of $N_k^{>n}(a^n)$, for some positive integers n and k. Given w and $f_u(w)$, for any $w \in \Sigma^n$, we can retrieve u.*

Proof. Let us note that, as we only had insertions and flips of letters, conceptually for each position in w there is a corresponding position in $f_u(w)$. The correspondence is given by ignoring the letters of $f_u(w)$ that were inserted by operation ins-diff. Our aim is to find all such pairs of corresponding positions in order to retrieve u.

To this end, we will swipe both w and $f_u(w)$ from right to left and prepend letters to an initially empty word u, which in the end will be equal to $u \in N_k^{>n}(a^n)$. We will maintain a position in each of the words, k_1 initiated as $|w|$ and k_2 initiated as $|f_u(w)|$.

Intuitively, we are first processing the part of $f_u(w)$ that comes from an application of operator g to w or to a suffix of w, depending on which case we are in. While processing this part, we maintain the invariant that the letter of $f_u(w)$ corresponding to $w[k_1]$ is the rightmost occurrence of $w[k_1]$ in $f_u(w)[1 \mathinner{\ldotp\ldotp} k_2]$, relying on the definition of g. Then we can apply the following procedure repeatedly; see Lines 4-10 in Algorithm 1. We compute the rightmost occurrence of $w[k_1]$ in $f_u(w)[1 \mathinner{\ldotp\ldotp} k_2]$; let it be at position j. We have that $u[j \mathinner{\ldotp\ldotp} k_2] = ab^{k_2-j}$. We prepend ab^{k_2-j} to u, decrement k_1 and set k_2 to $j-1$; see Example 13.

Let us now focus on the stopping condition of this procedure, i.e. the point where the remaining prefix of $f_u(w)$ does not originate from an application of g. If we are in Case 1, while $k_1 > 0$ we must have that $k_2 \geq k_1 + |p|$. If we are in Case 2, while $k_2 > k_1$ we must have that $k_1 \geq |p'|$. Overall, while $0 < k_1 < k_2$, we must have that $k_2 > |p|$ if we are in Case 1 or $k_2 > |p'|$ if we are in Case 2.

If at some point k_1 reaches 0, i.e. we have consumed all of w, then we are in Case 1. Thus, $f_u(w)[1 \mathinner{\ldotp\ldotp} k_2] = p$ and we prepend this prefix to u; see Lines 11-12 in Algorithm 1.

Else, if at some point $k_1 = k_2$, i.e. we are left with equal-length prefixes of w and $f_u(w)$, then we are either in Case 2 or in Case 1 with $p = \varepsilon$. By using the XOR operation $w[1 \mathinner{\ldotp\ldotp} k_1] \otimes f_u(w)[1 \mathinner{\ldotp\ldotp} k_2]$ in the former case we retrieve p' and in the latter case we get a^{k_1} which is the missing prefix of s. In either case we prepend the result to u; see Lines 13-14 in Algorithm 1. ◀

▶ **Example 13.** Let $u = abba \in N_2(a^3)$, $w = aab$, and $f_u(w) = abbb$. We have $k_1 = 3$ and $k_2 = 4$. At the first iteration of the while loop in Algorithm 1 we have $w[3] = f_u(w)[4] = b$ and so we set $k_1 = 2$, $u = a$ and $k_2 = 3$. At the second iteration we have $w[2] = a \neq f_u(w)[3] = b$ and so we get $u = ba$ and $k_2 = 2$. At this point we exit the while loop (because $k_1 = k_2$), and since we are at Case 2 we prepend $w[1 \mathinner{\ldotp\ldotp} 2] \otimes f_u(w)[1 \mathinner{\ldotp\ldotp} 2] = aa \otimes ab = ab$ to $u = ba$, which gives us $u = abba$. At this point we have retrieved $u = abba \in N_2(a^3)$.

By combining Lemmas 5 and 12 and Fact 11 we get $|N_k^j(a^n)| \leq |N_k^j(w)|$ for every j. This implies our main result for binary alphabets, apart from the strictness of the inequality. We leave the latter for the next section.

4   Generalisation to Arbitrary Alphabets and Strictness

For an arbitrary alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$ we only need to make minor adjustments in the definition of function f_u and in the algorithm for retrieving u from w and $f_u(w)$. Specifically, we replace the XOR operation by addition/subtraction modulo σ. Intuitively, one can think of 0's as a's in the binary case, and of non-0's as b's in the binary case.

Definition of f_u. Let u and v be two words of equal length. Let us denote by $u \oplus v$ the position-wise sum of words u and v modulo σ, e.g. for $\sigma = 4$ we have $1312 \oplus 1112 = 2020$. We analogously denote by $u \ominus v$ the position-wise subtraction of words u and v modulo σ.

We adapt operation ins-diff as follows, with z being a word containing only positive letters:

$$\text{ins-diff}(w, j, z) = \text{insert word } z \oplus (w[j])^{|z|} \text{ after the letter at position } j \text{ in word } w. \quad (4)$$

Operator g_x can now be applied to any word y such that $|y| = \#_0(x)$. g_x considers maximal blocks of letters in x not containing 0's instead of maximal blocks of b's (in the binary case). For such a block z, it performs an $\text{ins-diff}(y, j, z)$ operation on a word y.

The definition of f_u becomes as follows.

Case 1: $\#_0(u) \geq n$. We split the word u into p and s exactly as in the binary case and define $f_u(w) = p \cdot g_s(w)$; see Example 14.

▶ **Example 14.** Let $\Sigma = \{0, 1, 2\}$, $w = 201120$ and $k = 7$; note that $n = 6$. If $u = 0210001200210$, then $\#_0(u) = 7 \geq n$, so we are in Case 1. We have $p = 021$ and $s = 0001200210$ is the shortest suffix that contains $n = 6$ occurrences of the letter 0. Then $f_u(w)$ is constructed by concatenating p with the word $g_s(w)$ as shown in the figure below.

u:	0	2	1	0	0	0	1	2	0	0	2	1	0
w:				2	0	1			1	2			0
$f_u(w)$:	0	2	1	2	0	1	2	0	1	2	1	0	0

Case 2: $\#_0(u) < n$. The split of u into p' and s' is the same as in the binary case, but instead of s' containing $n - \#_a(u)$ b's it now contains $n - \#_0(u)$ non-0's. $f_u(w) = (p' \oplus w[1 \ldots |p'|]) \cdot g_x(w)[|p'| + 1 \ldots |u|]$, where $x = 0^{|p'|} s'$; see Example 15.

▶ **Example 15.** Let $\Sigma = \{0, 1, 2\}$, $w = 201120$ and $k = 5$; note that $n = 6$. If $u = 21021020$, then $\#_0(u) = 3 < n$, so we are in Case 2. We have $p' = 2102$ and $s' = 1020$ is the shortest suffix that contains $|u| - n = 2$ occurrences of letters different than 0. Then $f_u(w)$ is constructed by concatenating two words: the first one is $h_{p'}(w')$, where $w' = w[1 \ldots |p'|]$; and the second one is composed of the final $|s'|$ letters of $g_x(w)$, where x is the word obtained from u by changing the first $n - \#_0(u) = 3$ occurrences of non-0 letters to 0 as shown in the figure below.

u:	2	1	0	2	1	0	2	0
x:	0	0	0	0	1	0	2	0
w:	2	0	1	1		2		0
$g_x(w)$:	2	0	1	1	2	2	1	0
$h_{p'}(w')$:	1	1	1	0				
$f_u(w)$:	1	1	1	0	2	2	1	0

Algorithm 2 is an adaptation of Algorithm 1 for $\Sigma = \{0, \ldots, \sigma - 1\}$. Note that the two constructions are identical for $|\Sigma| = 2$, $a = 0$ and $b = 1$.

The proof of Lemma 5 that considers words of length at most n in $N_k(w)$ can be directly generalised for arbitrary alphabets, by allowing substitutions of letters instead of flips. This concludes the description of the generalisation.

The following theorem summarises all the results and introduces strictness in the inequality.

▶ **Theorem 2.** *Let $a \in \Sigma$ be an arbitrary element of alphabet Σ. For any positive integers n and k, we have $|N_{k,\Sigma}(a^n)| < |N_{k,\Sigma}(w)|$, for any non-unary word w of length n.*

▮ **Algorithm 2** RETRIEVE$(w, f_u(w))$ for $\Sigma = \{0, \ldots, \sigma - 1\}$.

Input: Two words w and $f_u(w)$.
Output: A word u.

1: $u \leftarrow \varepsilon$
2: $k_1 \leftarrow |w|$
3: $k_2 \leftarrow |f_u(w)|$
4: **while** $k_2 > k_1$ **and** $k_1 \geq 1$ **do**
5:     prepend$(f_u(w)[k_2] \ominus w[k_1], u)$
6:     **if** $w[k_1] = f_u(w)[k_2]$ **then**
7:         $k_1 \leftarrow k_1 - 1$
8:     $k_2 \leftarrow k_2 - 1$
9: **if** $k_1 = 0$ **then** $\backslash\backslash$ $k_2 > 0$; Case 1 with $p \neq \varepsilon$
10:     prepend$(f_u(w)[1 \mathinner{\ldotp\ldotp} k_2], u)$
11: **else** $\backslash\backslash$ $k_1 = k_2$; Case 2 (or Case 1 with $p = \varepsilon$)
12:     prepend$(f_u(w)[1 \mathinner{\ldotp\ldotp} k_2] \ominus w[1 \mathinner{\ldotp\ldotp} k_1], u)$
13: **return** u

Proof. We have $|N_{k,\Sigma}^j(a^n)| \leq |N_{k,\Sigma}^j(w)|$ for every j by combining the counterparts of Lemmas 5 and 12 and Fact 11 for an arbitrary alphabet. It thus suffices to find some value of j for which this inequality is strict.

Let us first consider the case that $k < n$, in which we claim that

$$|N_{k,\Sigma}^{n-k}(w)| > |N_{k,\Sigma}^{n-k}(a^n)| = 1.$$

Note that in this case words of length $n - k$ can be obtained only by performing k deletions, i.e. no insertions or substitutions are allowed. Hence $|N_{k,\Sigma}^{n-k}(a^n)| = 1$. For a non-unary w, we can, for instance, delete letters in lexicographic or reverse lexicographic order, breaking ties arbitrarily, obtaining words with different multiplicities for some letter.

Let us now proceed to the complementary case that $k \geq n$.

Then, each word $u \in N_{k,\Sigma}^{k+1}(a^n)$ can be obtained by exactly $k+1-n$ insertions and at most $n-1$ substitutions. Let us restrict ourselves to determining the size of $N'(w) \subseteq N_{k,\Sigma}^{k+1}(w)$, defined as the set of elements of $N_{k,\Sigma}^{k+1}(w)$ that can be obtained from w using exactly $k+1-n$ insertions and at most $n-1$ substitutions. In particular, one letter from w remains unchanged and gets shifted to the right by at most $k + 1 - n$ positions – possibly not shifted at all. Thus, each word $u \in N'(w)$ can be obtained as follows. We first choose the position i in u where the shifted letter has landed. For such a position i, it is a letter c occurring in $w[\max(1, i - (k + 1 - n)) \mathinner{\ldotp\ldotp} \min(i, n)]$ – any of those letters can be chosen by picking a right layout of insertions. We then put c at position i and fill the remaining k positions arbitrarily; see Example 16.

Let us do the above process once for each position i, with a fixed letter $\lambda(i)$, arbitrarily chosen from the possible ones. In total, we obtain all words from Σ^{k+1} apart from the ones which differ from $\lambda(i)$ on every position i. In particular, the total number of words that we get for this specific choice of $\lambda(i)$'s is $\sigma^{k+1} - (\sigma - 1)^{k+1}$ and this is equal to $|N'(a^n)| = |N_{k,\Sigma}^{k+1}(a^n)|$. Then, at some position j, since w is non-unary we can actually choose a letter $c \neq \lambda(j)$ instead; for instance any position j such that $w[j - 1] \neq w[j]$ will work. Let us now pick this letter c and fill each other position i with a letter different from $\lambda(i)$. This way we obtain a word that was not obtained with the previous choice of $\lambda(i)$'s and hence $|N_{k,\Sigma}^{k+1}(w)| \geq |N'(w)| > |N_{k,\Sigma}^{k+1}(a^n)|$.   ◄

▶ **Example 16.** Let us consider word $w = abc$ and $k = 5$. Every word $u \in N'(w)$ is obtained by 3 insertions and up to 2 substitutions. If the letter a from w is not substituted for, it can land at any of the positions from 1 to 4 in u; similarly, b and c can land at positions from 2 to 5 and from 3 to 6, respectively. This is shown schematically in the following table.

position in u	1	2	3	4	5	6
landing positions of a	a	a	a	a		
landing positions of b		b	b	b	b	
landing positions of c			c	c	c	c

Then the i–th column specifies the possible choices for $\lambda(i)$, e.g., $\lambda(2) \in \{a, b\}$. Note that if w was unary, then all those sets would be singletons.

One possible choice of $\lambda(1), \ldots, \lambda(6)$ is a, a, c, a, b, c. For it, we generate all the words but the 2^6 words that have no positions in common with $aacabc$. For a different choice of $\lambda(i)$'s, say, a, b, c, a, b, c, we obtain a word that was not generated before, e.g., $bbabca$ that has exactly one position in common with $abcabc$.

Let us now complete the picture by showing a closed formula for obtaining the tight lower bound implied by Theorem 2 and thus an efficient way to compute this bound.

▶ **Fact 17.**

$$|N_{k,\Sigma}(a^n)| = \sum_{i=0}^{k} \sum_{j=i-k}^{k} \binom{n+j}{i} (\sigma - 1)^i.$$

Proof. We first choose the number i of letters that are different from a in some $u \in N_{k,\Sigma}(a^n)$ and then the length $n + j$ of the word. Note that $j \geq i - k$ since we can have at most $k - i$ deletions as we need at least i insertions or substitutions to have i letters different from a. We then have $\binom{n+j}{i}$ options to choose the positions where the letter is not a and $(\sigma - 1)$ letters to choose from for each such position. ◀

▶ **Remark 18.** $|N_{k,\Sigma}(a^n)|$ can be computed with $\mathcal{O}(k^2)$ arithmetic operations.

5   Final Remarks

We showed a tight lower bound on the size of the Levenshtein k-neighbourhood. In particular, we defined a function f_u for each word $u \in N_{k,\Sigma}(a^n)$, such that, for any given $w \in \Sigma^n$, we have that $f_u(w) \in N_{k,\Sigma}(w)$ and $f_u(w) \neq f_{u'}(w)$ for $u \neq u'$. Our construction is not the only one possible. For example, in Case 1 of our construction, one could take $f_u(w) = q \cdot g_s(w)$, where $q = p \oplus 1^{|p|}$ (for the binary case, this corresponds to the negation of p). However, our construction has a neat property that $f_u(a^n) = u$, for any $u \in N_{k,\Sigma}(a^n)$.

The following two questions remain unanswered:

1. Can a similar approach be employed for showing a tight upper bound on $|N_{k,\Sigma}(w)|$?
2. Touzet gave an algorithm for computing $|N_{k,\Sigma}(w)|$ for a word w of length n over an alphabet Σ that works in time linear in n but exponential in k [11]. Can this computation be done in polynomial time or is this problem $\#P$-hard?

References

1   Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. `doi:10.1016/S0022-2836(05)80360-2`.

2   Leonor Becerra-Bonache, Colin de la Higuera, Jean-Christophe Janodet, and Frédéric Tantini. Learning balls of strings from edit corrections. *The Journal of Machine Learning Research*, 9:1841–1870, 2008. URL: `https://dl.acm.org/citation.cfm?id=1442793`.

3   Edgar Chávez, Gonzalo Navarro, Ricardo A. Baeza-Yates, and José L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001. `doi:10.1145/502807.502808`.

4   Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings.* Cambridge University Press, 2007.

5   Yoann Dufresne. An exploration of Levenshtein neighborhood densities. 14th Workshop on Compression, Text and Algorithms (WCTA), 2019. Talk.

6   Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10:707, 1966.

7   Stoyan Mihov and Klaus U. Schulz. Fast approximate search in large dictionaries. *Computational Linguistics*, 30(4):451–477, December 2004. `doi:10.1162/0891201042544938`.

8   Gene Myers. What's behind BLAST. In Cédric Chauve, Nadia El-Mabrouk, and Eric Tannier, editors, *Models and Algorithms for Genome Evolution*, pages 3–15. Springer, 2013. `doi:10.1007/978-1-4471-5298-9_1`.

9   Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001. `doi:10.1145/375360.375365`.

10   Marie-France Sagot and Yoshiko Wakabayashi. Pattern inference under many guises. In Bruce A. Reed and Cláudia L. Sales, editors, *Recent Advances in Algorithms and Combinatorics*, pages 245–287. Springer New York, 2003. `doi:10.1007/0-387-22444-0_8`.

11   Hélène Touzet. On the Levenshtein automaton and the size of the neighbourhood of a word. In *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Proceedings*, pages 207–218, 2016. `doi:10.1007/978-3-319-30000-9_16`.