



HOKKAIDO UNIVERSITY

Title	The q-gram Distance as an Approximation of the Edit Distance
Author(s)	花田, 博幸
Degree Grantor	北海道大学
Degree Name	博士(情報科学)
Dissertation Number	甲第11490号
Issue Date	2014-06-30
DOI	https://doi.org/10.14943/doctoral.k11490
Doc URL	https://hdl.handle.net/2115/64515
Type	doctoral thesis
File Information	Hiroyuki_Hanada.pdf



Doctoral Dissertation

The q -gram Distance as
an Approximation of the Edit Distance

Hiroyuki HANADA

Hokkaido University

Graduate School of Information Science and Technology

Division of Computer Science

Contents

1	Introduction	1
1.1	Overview	1
1.2	Contributions	2
2	Definitions	4
2.1	General Definitions	4
2.2	q -gram Distance	4
3	Related Researches	6
3.1	q -gram Distance and Related Distances	6
3.2	Other Approximation Distances for the Edit Distance	7
3.3	Cost Reduction for the Edit Distance without Approximation	8
3.4	Indices for the Edit Distance	10
4	Distortion Analysis of Edit Distance Approximation Algorithms	11
4.1	Distortion	11
4.1.1	Definition	11
4.1.2	Discussions in Distortion Analysis	11
4.2	Theoretical Comparison of Refined Distortions	13
4.2.1	Outline	13
4.2.2	Derivation of Distortions	13
4.2.3	Comparison of Calculated Distortions	15
4.3	Experimental Comparison of Distortions	18
4.3.1	Procedure	18
4.3.2	Results	19
4.4	Summary of Approximation Accuracy Analysis	22
5	Fast Similar Substring Searching Algorithm by the q-gram Distance	23
5.1	Outline	23
5.2	Problem Setting	25
5.3	Change Point	25
5.4	Ukkonen's Two Algorithms	29
5.4.1	Algorithm <i>ArraySearch</i>	29

5.4.2	Algorithm <i>TreeSearch</i>	29
5.5	Proposed Algorithm <i>ArrayBaselineSearch</i>	35
5.6	Proposed Algorithm <i>ListBaselineSearch</i>	40
5.6.1	Outline	40
5.6.2	Complexity Analysis	48
5.7	Experiment	51
5.7.1	Settings	51
5.7.2	Result	51
6	Conclusion	54
6.1	Results	54
6.2	Discussions	54
6.3	Future Directions	55
A	Detailed Distortion Analysis of Six Edit Distance Approximation Algorithms	57
A.1	Distortion Calculations from Inequalities	57
A.1.1	Algorithm <i>Bar-Yossef-2004</i>	57
A.1.2	Algorithm <i>Sokolov-2007</i>	57
A.2	Distortion Refinements without Big-oh Notations	58
A.2.1	Algorithm <i>Batu-2006</i>	58
A.2.2	Algorithm <i>Charikar-2006</i>	60
A.2.3	Algorithm <i>Andoni-2009</i>	60
B	Detailed Procedures of DEL-FIRST, DEL-MIDDLE and INS/DEL-LAST	62
B.1	Operation DEL-FIRST	62
B.2	Operation DEL-MIDDLE	62
B.3	Operation INS/DEL-LAST	62
	Publications	66
	References	67
	Acknowledgment	72

List of Figures

1	An example of similar substring searching by the edit distance in $O(t p)$ time ($t = \text{"abcaab"}, p = \text{"baa"}, k = 1$)	9
2	The concept of distortion K over a set S	12
3	Several situations in distortion evaluation	12
4	Conversion of lower and upper bounds to a distortion	15
5	Distortions of six approximation methods.	17
6	Experimental distortions of six algorithms	20
7	Experimental distortions of six data sets	21
8	The q -gram distance sequences $\mathbf{d}_{(1)} = \{d_q(t[1..j], p)\}$ and $\mathbf{d}_{(2)} = \{d_q(t[2..j], p)\}$ for $t = \text{"aaaccaaababc"}$ and $p = \text{"aaabbc"}$ with $q = 2$	28
9	Update procedure of <i>ArraySearch</i> in the example of Table 4	30
10	The binary search tree used in <i>TreeSearch</i> for the example of Table 4	33
11	Update procedure of <i>TreeSearch</i> in the example of Table 4	34
12	Algorithm <i>ArrayBaselineSearch</i>	36
13	Data structure of LsAT: A doubly-linked list L supported by an array A and a search tree T	41
14	Searching for a change point in an LsAT (example of finding $c_i = 15$ or the next larger element)	41
15	Update procedure of LsAT in the example of Table 4	43
16	Algorithm <i>ListBaselineSearch</i>	44
17	Case study of DEL-FIRST step in Algorithm <i>ListBaselineSearch</i>	45
18	Case study of DEL-MIDDLE in Algorithm <i>ListBaselineSearch</i> according to the position of change point c_i (corresponding to step (2) in Fig. 15)	46
19	Case study of INS/DEL-LAST step in Algorithm <i>ListBaselineSearch</i>	47
20	An example of removing the leftmost node of T by DEL-FIRST	50
21	Computation time of similar substring searching ($ t = 100,000, \Sigma = 20, q = 5$)	52
22	Case study of removing two elements in LsAT by DEL-MIDDLE	63

23	Case study of INS/DEL-LAST in LsAT	65
----	---	----

List of Tables

1	Refined distortions	14
2	Best algorithm according to the alphabet size $ \Sigma $, the string length n and the number of edits (an upper bound of the edit distance) e	19
3	Computational complexity of similar substring searching in d_e and d_q	24
4	The q -gram distance sequence $\mathbf{d}_{(i)}$ for $i = 1, 2, \dots, 12$ for $t =$ “aaaccaababc” and $p =$ “aaabbc” with $q = 2$ and $k = 3$. . .	28

1 Introduction

1.1 Overview

The amount of string data on computers is now increasing rapidly. As we all know, text data or other string-like data such as sound and video are easily available via the Internet and the amount is steadily increasing. In addition, in biology, genome sequences are also becoming easy to access as the technology of sequencing (retrieving sequences from genome molecules) is developed [1]. As a result, the importance of analyzing them is also increasing.

Given a query string, finding strings in databases that not only exactly but also approximately match the query is a useful tool. A representative similarity is the *edit distance* $d_e(x, y)$, which is defined by the minimum number of character insertions, deletions and/or substitutions to make x identical to y [2, 3]. The concept of the edit distance is first appeared in 1960s [4, 5], and it has been intensively studied from its wide applications: in fact, the edit distance reflects misspellings in text data, noise in sound data, mutations in genome or protein sequences and so on [3]. As a result, finding similar substrings by the edit distance results in error-tolerant searching and analysis. Another reason why the edit distance is intensively studied is its high computational cost $O(|x||y|)$ ($|\cdot|$: string length). Because we have to compute distances for many pairs of strings in most of applications, typically searching similar strings from a database, a large amount of effort has been devoted for faster computation.

Typical solutions for reducing the computation time for the edit distance are: indexing strings in databases such as suffix trees and space-partitioning search trees (Section 3.4), limiting the edit distance by certain upper bound (Section 3.3) and approximating the edit distance with small computational cost (Sections 3.1 and 3.2). In this paper we focus on the solution of approximating the edit distance, and especially on the *q -gram distance* [6, 7] as a costless edit-distance-approximation distance.

The *q -gram distance* $d_q(x, y)$ is defined by the number of different *q -grams* (all substrings of length q) between x and y (Section 2.2). The *q -gram distance* is regarded as a promising approximation of the edit distance from following reasons: one is that $d_q(x, y)$ can be computed in $O(|x| + |y|)$ time, much faster than that of $d_e(x, y)$ [6], the other is that some inequalities

between d_e and d_q have been derived to assure the range of the approximated edit distance [6–8] (Section 4.2.1).

1.2 Contributions

Although many theoretical properties of the q -gram distance as an approximation of the edit distance have been proven, for higher real-life usefulness, we consider the following two problems: one is to make clear when the q -gram-distance-based edit distance approximation distances are advantageous to others in approximation accuracy, the other is to speed up the similar substring searching, whose time complexity has been larger than the ideal one.

The first problem of the approximation accuracy is discussed in Section 4 and the shown result is from author’s work [9], refined from [10]. We consider comparing the *distortions*, a single-valued measure of function approximation accuracy (Section 4.1.1), to clarify the condition when the q -gram-distance-based approximation algorithms are accurate than others. In the existing approximation algorithms, the approximation accuracy expressions of two q -gram-distance-based algorithms by Bar-Yossef *et al.* [8] and Sokolov [7] are shown as inequalities, while others are in distortions of big-oh (asymptotic) notations in the string length n such as $O(n)$ and $O(n \log n)$ (Section 4.2.1). To compare them in a unified and practically useful measure, we derive their non-asymptotic distortions of them by converting inequalities and removing asymptotic analysis. In addition, we implemented these algorithms and compared distortions experimentally. As a result, the two q -gram-distance-based algorithms are proved to be advantageous in the distortion for small string length $n \leq 300$ from both theoretical and experimental analyses, and for large alphabet size from experimental analysis.

The second problem of faster substring searching is discussed in Section 5 and the shown result appears in author’s works [11], refined from [12]. The resulted algorithm achieves for the similar substring searching by the q -gram distance in $O(|t| + |p|)$ time on average (t : text string, p : pattern string). More precisely, given t , p , q and a threshold integer k ($= O(|p|)$): see Section 5.1), we consider finding all substrings of t whose q -gram distances to p is k or less. Then currently the best time complexity achieved by Ukkonen [6] is

$O(|t| \log k + |p|) = O(|t| \log |p| + |p|)$. The proposed algorithm [11] reduced the average-case complexity to $O(|t| + |p|)$, if q is larger than certain threshold and t is generated by randomly choosing every character, with keeping the worst-case complexity as $O(|t| \log k + |p|)$. The time complexity is achieved by replacing the structure storing the positions of substrings of t similar to p : from a search tree of size $O(k)$ (*TreeSearch* in Section 5.4.2) to an indexed doubly-linked list of size $O(k)$ (*ListBaselineSearch* in Section 5.6). In addition, experimental results assured the theoretical complexity as well as the three-time-faster computation time than the Ukkonen's algorithm for pattern length 500.

2 Definitions

2.1 General Definitions

We denote the alphabet (the set of characters) by Σ . Let Σ^n be the set of all strings of length n and $\Sigma^* \stackrel{\text{def}}{=} \{x \mid x \in \Sigma^n, n \in \{0, 1, 2, \dots\}\}$ be the set of all finite-length strings over Σ .

For a string x , we use the following notations. By $|x|$ we denote the length and by $x[i..j]$ we denote the substring starting from i and ending at j . When $j < i$, $x[i..j]$ denotes an empty string. Since we often consider substrings of length q , called q -grams, we use $x_{(i)}$ to specify the q -gram starting from the i th position, that is, $x_{(i)} \stackrel{\text{def}}{=} x[i..i+q-1]$. Similarly, we denote the q -gram ending at the j th position by $x_{<j>} \stackrel{\text{def}}{=} x[j-q+1..j]$. For example, if $x = \text{"abcdefgh"}$ and $q = 2$, then $|x| = 8$, $x[3..7] = \text{"cdefg"}$, $x_{(2)} = \text{"bc"}$ and $x_{<5>} = \text{"de"}$. For the cardinality of a set A , we use the same notation $|A|$.

2.2 q -gram Distance

For a string x , let H_x be the histogram of x over Σ^q , that is, H_x stores the appearance number of every q -gram in x . Furthermore, by $|H_x|$ we denote the total number of q -grams with duplication in x , that is, $|H_x| = |x| - q + 1$, and by $\#H_x(s)$ ($s \in \Sigma^q$) we denote the number of appearances of s in x . For example, for $x = \text{"abcdabcd"}$ and $q = 3$, $H_x = \{2/\text{"abc"}, 2/\text{"bcd"}, 1/\text{"cda"}, 1/\text{"dab"}\}$, $|H_x| = 6$, $\#H_x(\text{"abc"}) = 2$ and $\#H_x(\text{"aaa"}) = 0$.

Definition 2.1. [6] The q -gram distance $d_q(x, y)$ between two strings x and y is defined by the L_1 -distance between the two histograms H_x and H_y , that is,

$$d_q(x, y) \stackrel{\text{def}}{=} \|H_x - H_y\|_1 = \sum_{g \in \Sigma^q} |\#H_x(g) - \#H_y(g)|.$$

Note that the q -gram distance is *not* strictly a distance. d_q satisfies the axiom of the distance¹ except for the distance being possibly zero even if

¹For a set U , a function $d : U \times U \rightarrow \mathbb{R}$ is called a *distance* if, for any $x, y, z \in U$, (1) $d(x, x) = 0$, (2) $x \neq y \Rightarrow d(x, y) > 0$, (3) $d(x, y) = d(y, x)$ and (4) $d(x, y) + d(y, z) \geq d(x, z)$.

the strings are not the same, for example, $d_2(\text{"aba"}, \text{"bab"}) = 0$.

We can compute the q -gram distance $d_q(x, y)$ in $O(|x| + |y|)$ time, regardless of the value of q , as shown by Ukkonen [6]. The complexity is achieved by the help of a *suffix tree*² [2, 13] of either x or y . We add that we consider only the case when the suffix tree for a string of length n is constructed in $O(n)$ time and space, that is, Σ is a constant against n [14, 15].

As a property of the q -gram distance as an approximation of the edit distance, the q -gram distance bounds the edit distance from below as follows. As will be shown in Section 4.2.1, combinations of the q -gram distances derive more bounds.

Property 2.1. [6] *For any strings x and y , $d_e(x, y) \geq d_q(x, y)/(2q)$.*

Proof. Consider each of three edit operations of the edit distance (Section 1). For any string x ,

- If an insertion is conducted just after the i th character of x , at most $(q - 1)$ q -grams in x (from $x[(i - q + 2)..(i + 1)]$ to $x[i..(i + q - 1)]$) are changed into at most q new q -grams (from $x[(i - q + 2)..(i + 1)]$ to $x[(i + 1)..(i + q)]$). Therefore one insertion can change at most $(2q - 1)$ q -grams in x .
- Changes of q -grams by a deletion are the reverse of these for insertion. At most q q -grams in x are changed into at most $(q - 1)$ new q -grams. Therefore one deletion can change at most $(2q - 1)$ q -grams in x .
- If a substitution is conducted just after the i th character of x , at most q q -grams in x are changed into at most q new q -grams. Therefore one substitution can change at most $2q$ q -grams in x .

As a result, q -gram distance increases by at most $2q$ per one edit operation. Therefore we get intended inequality $d_q(x, y) \leq 2q \cdot d_e(x, y)$. $\square$

²In reality, Ukkonen [6] uses a *suffix automaton* instead, a compressed structure of the suffix tree. We explain it as a suffix tree through the paper for simplicity.

3 Related Researches

3.1 q -gram Distance and Related Distances

As far as the author knows, the q -gram distance $d_q(x, y)$ is first focused on as an approximation of the edit distance by Ukkonen [6]. The research showed that the distance can be computed in $O(|x| + |y|)$ time (Section 2.2), the substring searching can be done in $O(|t| \log k + |p|)$ time (Section 5), and a lower bound of the edit distance $d_e(x, y) \geq d_q(x, y)/2q$ (Section 2.2).

Later Bar-Yossef *et al.* [8] and Sokolov [7] developed q -gram-distance-based algorithms approximating the edit distance for two strings of the same lengths. They divide given strings into some overlapping substrings and then sum up the q -gram distances for corresponding locations of substrings, and proved some upper and lower bounds of the edit distance. (The bounds are shown in Section 4.2.1.) Their distances are defined as follows:

Definition 3.1. (*Bar-Yossef's distance*) [8] For two strings x and y ($|x| = |y|$) and an integer q ,

$$d_q^B(x, y) \stackrel{\text{def}}{=} \sum_{m=0}^{q-1} d_q(x[p_m..p_{m+1} + q - 1], y[p_m..p_{m+1} + q - 1]),$$

where $p_m \stackrel{\text{def}}{=} \lfloor mq/|x| \rfloor$.

Definition 3.2. (*Sokolov's distance*) [7] For two strings x and y ($|x| = |y|$), an integer w and two integers q_1 and q_2 ,

$$d_q^S(x, y) \stackrel{\text{def}}{=} \frac{\left[\sum_{i=1}^{n-w+1} \sum_{q=q_1}^{q_2} d_q(x[i..i + w - 1], y[i..i + w - 1]) \right]}{(n - w + 1)(q_2 - q_1 + 1)}.$$

The difference is how to take the substrings to be computed the q -gram distance between. Note that Sokolov's distance is equivalent to the q -gram distance when $w = n$ and $q_1 = q_2 = q$.

Not only approximating the edit distance, the idea of vectorization by q -gram appearances itself has been applied to the string classification problem. Especially, because strings can be treated as vectors, the *kernel methods* can be applied (*string kernel*), which has been developed for text classification, genome analysis and so on [16, 17]. Leslie *et al.* [16] consider the same vector

as the q -gram distance (consisting of numbers of q -gram appearances), while Lodhi *et al.* [17] consider the vector consisting of number of appearances of length- q subsequences (not substring).

3.2 Other Approximation Distances for the Edit Distance

To author’s knowledge, there exist at least three other classes of edit distance approximation algorithms than using the q -gram distance, on condition that the computational times are at most $O((|x| + |y|)^{1+\epsilon})$ (x, y : strings whose distance is approximated, ϵ : any positive number). In Section 4, the four algorithms shown in this section are compared to the two q -gram-distance-based algorithms (Bar-Yossef’s and Sokolov’s in the previous section).

The algorithm by Batu *et al.* [18] converts a substring of given strings into one character in a certain rule, for example, “abcbabac” to “XYX” with the rule “abc” $\rightarrow$ “X” and “bb” $\rightarrow$ “Y”. The approximated value of the edit distance is obtained by the edit distance of the two converted strings. String lengths after the conversion causes a trade-off between the computational time and the approximation accuracy: smaller length after the conversion saves more computational time and yields lower approximation accuracy. It achieves $O((|x| + |y|)^{1+\epsilon})$ time by controlling the parameter c , where at least c and at most $2c - 2$ characters are converted into one character.

Andoni *et al.* [19] focus on the fact that the high computational cost of the edit distance comes from the property of the *alignment*, that is, insertions and deletions are allowed to match two strings as possible. They proposed limiting the locations of insertions and deletions in a string (number of insertions and deletions are not limited) for a small number at first, for example, every 100 characters, and then increase the number of locations step by step. It achieves $O((|x| + |y|)^{1+\epsilon})$ time by controlling how many locations are allowed for insertions and deletions.

The algorithms by Charikar *et al.* [20] and Andoni *et al.* [21] are originally considered for the *Ulam metric*, the metric space of the edit distance where every string consists of distinct characters [20]. Although the Ulam metric accepts limited strings, it is applicable for the general edit distance by multiplying the *distortion* (Section 4.1.1) by $2 \cdot \max\{|x|, |y|\}$ with a simple string conversion [20] (detailed in Section 4.3.1). Because of the constraint on the strings, the distance can be approximated according to the positions

given characters are found. For example of Charikar’s algorithm [20], the approximated distance is computed as the sum of $|1/(x^{-1}[b] - x^{-1}[a]) - 1/(y^{-1}[b] - y^{-1}[a])|$ for all pairs $(a, b) \in \Sigma \times \Sigma$, where $x^{-1}[a]$ denotes the position of a found in the string x (omitted if a is not in x). These algorithms achieve $O(|x| + |y|)$ time.

3.3 Cost Reduction for the Edit Distance without Approximation

Different from Sections 3.1 and 3.2, we overview in this section the algorithms that achieve fast edit distance computation without approximation. In this section we consider the similar substring searching by the edit distance, that is, to find substrings of the text string t whose edit distances to the pattern string p are k ($= O(|p|)$) or less (similar to the searching by the q -gram distance defined in Section 5.1).

In this section we use the notations t , p and k with the same meaning as in Section 1, except for d_q is replaced with d_e .

According to Navarro’s survey [3], the lowest time complexity for the searching problem seems to be $O(|t|k + |p|)$ if the space complexity is limited to $O(|t| + |p|)$. Thus smaller k makes the searching faster. Note that it is equal to $O(|t||p|)$ with our assumption $k = O(|p|)$ (Section 5.1), which is equal to the time complexity of single edit distance computation $O(|x||y|)$.

The most simple algorithm needs $O(|t||p|)$ time regardless of k [22], which is achieved by the dynamic programming technique (Fig. 3.3). It is later reduced to $O(|t|k + |p|)$ by Ukkonen [23] for average cases and by Landau and Vishkin [24] for even worst cases, both of which achieves the time by omitting computing some of the table cells in Fig. 3.3.

If larger space complexity than $O(|t| + |p|)$ is acceptable, it is clear that constructing an automaton accepting all possible similar strings $\{s \mid d_e(s, p) \leq k\}$ achieves $O(|t| + |p|)$ -time searching (e.g., Ukkonen [23] and Melichar [25]). However, such algorithms consume at least $O(|p|^k)$ space, or $O(|p|^{|p|})$ if $k = O(|p|)$.

Filtering of candidate substrings by exact matching has been also shown to be effective for the edit distance, as seen in [26, 27]. The “filtering” means, for example of $|x| = |y| = 200$ and $d_e(x, y) = 1$, a substring of length at least 100 matches exactly in x and y . However, the length of exactly matching

$p \setminus t$	a	b	c	a	a	b
	0	0	0	0	0	0
b	1					
a	2					
a	3					

(a) Initialized table

$p \setminus t$	a	b	c	a	a	b
	0	0	0	0	0	0
b	1	1	0	1	1	0
a	2	1	1	1	1	1
a	3	2	2	2	1	2

(b) Filled table

$p \setminus t$	ξ			
	$\vdots$		$\vdots$	
	$\cdots$	v_1		v_2
			$\searrow$	$\downarrow$
η	$\cdots$	v_3	$\rightarrow$	v
	$\vdots$			$\vdots$

(c) Determination of a cell value:

$$v = \min\{v_1 + (\xi = \eta ? 0 : 1), v_2 + 1, v_3 + 1\}$$

Figure 1: An example of similar substring searching by the edit distance in $O(|t||p|)$ time ($t = \text{"abcaab"}$, $p = \text{"baa"}$, $k = 1$) by [22]. First initialize the table in the way of Fig. (a) and then fill the values as in Fig. (b) by the rule in Fig. (c). Lastly we extract the distances k or less in the bottom row (**bold** numbers). In this case $t[2..4] = \text{"bca"}$ and $t[2..5] = \text{"bcaa"}$ is extracted with the distance to p being 1 ($\leq k$).

substring becomes smaller for larger edit distance, and thus ineffective for too large k .

From a practical point of view, fast distance computation in implementation is also important, even if it does not improve the theoretical time complexity. A typical way is to describe the dynamic programming table (Figure 3.3) bitwisely to exploit fast bitwise operations [28, 29].

3.4 Indices for the Edit Distance

Indexing is also an important approach of faster searching. This approach can also be separated by whether the distance is approximated or not.

For example of indexing without approximation, Cobbs proposed an algorithm that conduct similar substring searching by the edit distance on a suffix tree for faster searching [30]. Using space-partitioning trees [31] are also considered to exploit the axiom of the distance. Since trees for Cartesian space like kd-tree [32] cannot be applicable to the edit distance, trees that assumes only the axiom of the distance is often considered for the edit distance [33, 34] such as the *Vantage Point Tree* [35] and the *Generalized Hyperplane Tree* [36]. Although not a tree structure, the *reference-based index* is also an index based on the axiom of the distance considered for the edit distance [37]. Navarro *et al.* proposed an index based on both the suffix tree and the distance-based index [38].

For example of indexing with approximation, FASTA [39] and BLAST [40], searching tools for similar genome sequences, adopt the tactics that they begin searching similar substrings where the text and the pattern matches at least certain number of characters (the number may vary; six is a popular setting for genome sequences) [41]. As a result, the true answer may not be appeared in the result.

4 Distortion Analysis of Edit Distance Approximation Algorithms

Note: The content of the section is based on the author's works [9, 10].

As stated in Section 1, we compare two q -gram-distance-based and four other edit distance approximation algorithms explained in Sections 3.1 and 3.2 in the *distortion* to clarify the condition when the q -gram-distance-based ones achieves better approximation.

4.1 Distortion

4.1.1 Definition

We use the *distortion*, also known as the *approximation factor*, as a single-valued measure of approximation accuracy of a function, that is, how the approximation function is close to the original function. It is defined as follows:

Definition 4.1. [20, 42] Given a set S , a non-negative function $f(z)$ and a non-negative approximation function $\tilde{f}(z)$, the distortion of $\tilde{f}(z)$ to $f(z)$ is defined by the smallest $K \in [1, +\infty)$ such that

$$\exists K' \in (0, +\infty), \forall z \in S : f(z) \leq K' \tilde{f}(z) \leq K f(z).$$

The concept is illustrated in Fig. 2. Note that, in this paper, S is given as a set of pairs of strings $\{z = (x, y)\}$ since we consider $f(z) = d_e(x, y)$ and $\tilde{f}(z) = \tilde{d}_e(x, y)$, where $\tilde{d}_e(x, y)$ is a string distance approximating $d_e(x, y)$. The value of K shows the ratio of the upper bound $(K/K')f(z)$ to the lower bound $(1/K')f(z)$. A smaller value of distortion $K (\geq 1)$, therefore, means better approximation. Especially, $K = 1$ means that $f(z)$ and $\tilde{f}(z)$ are proportional to each other.

4.1.2 Discussions in Distortion Analysis

In this paper we may limit the string pairs to evaluate the distortion. We especially consider the following two limitations (Fig. 3). One is the condition $|x| = |y| = n$ (Fig. 3, (a) and (b)) because the value of distortion tends to become larger as the string length n increases. Taking this tendency into

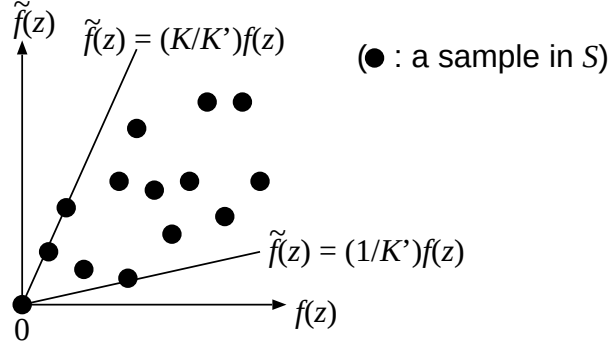


Figure 2: The concept of distortion K over a set S

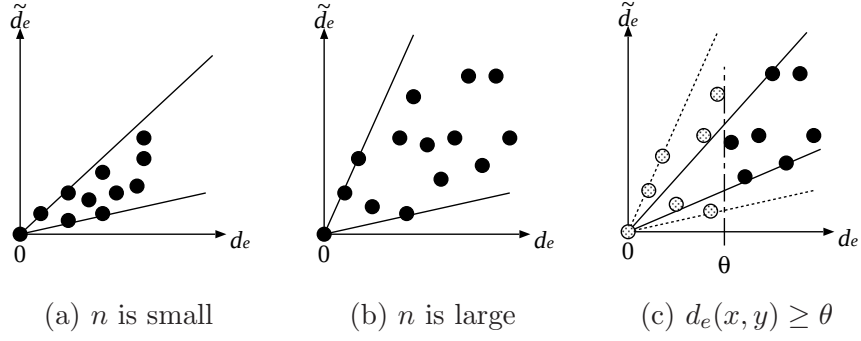


Figure 3: Several situations in distortion evaluation

account, many of existing papers (three algorithms out of six) evaluate the distortions by big-oh notations, that is, how slowly the value K increases as n increases. The other is the condition $d_e(x, y) \geq \theta$ (Fig. 3(b)(c)) because the distortion is often affected strongly by string pairs with a small value of d_e . To ignore such an exceptional situation, we evaluate the distortion only in the range of $d_e \geq \theta$ with a threshold θ (Fig. 3(c)).

As stated in Section 1, some of the distortions of the approximation distances have been known as only big-oh (asymptotic) notations, which we convert to non-asymptotic ones. This conversion results in not only making possible comparing distances in a unified measure but also clarify the range of n which distance yields the smallest distortion. For example, if two approximation algorithms have the distortions $n^2/10$ and $10n$, then we can switch two algorithms by $n \leq 100$ or not. Such judgment could not be made if it were $O(n^2)$ and $O(n)$.

4.2 Theoretical Comparison of Refined Distortions

4.2.1 Outline

We re-analyzed the six algorithms to obtain their non-asymptotic distortions. The results are shown in Table 1. As seen in the table, existing accuracy evaluations of q -gram-distance-based approximations are shown as inequalities, while others distortions of big-oh notations.

Before analyzing the table in detail in Section 4.2.3, in Section 4.2.2 we explain how the constant factors are extracted from big-oh notations, and how the accuracy evaluations with inequalities are converted to distortions with a threshold θ .

4.2.2 Derivation of Distortions

For each algorithm whose accuracy is bounded by inequalities (*Bar-Yossef-2004* and *Sokolov-2007*), we calculated its distortion by the following procedure. Detailed distortion calculations for the two algorithms are shown in Appendix A.1.

Let $\tilde{d}_e$ be bounded by two functions of d_e as $l(d_e) \leq \tilde{d}_e \leq u(d_e)$ for $d_e \geq \theta$ (Fig. 4(a)). Then the distortion K of $\tilde{d}_e$ for $d_e \geq \theta$ is upper-bounded by $K_\theta = u(\theta)/l(\theta)$ under the monotonicity of slopes $u(d_e)/d_e$ and $l(d_e)/d_e$. Indeed, if $u(d_e)/d_e$ and $l(d_e)/d_e$ are monotonically decreasing and increasing

Table 1: Refined distortions. Here, $\tilde{d}_e$ is the approximated distance of d_e ; the strings are limited to length n ; a threshold θ is employed to limit $d_e \geq \theta$ in some algorithms. In logarithms, the bases are 2 for $\lg$ and e for $\ln$, respectively.

Algorithm	Original distortion	Original inequality	Refined distortion
<i>q</i> -gram-distance-based approximation algorithms			
<i>Bar-Yossef-2004</i> [8]		$\begin{cases} d_e \leq k \Rightarrow \tilde{d}_e \leq 4kq,^\dagger \\ d_e \geq 13(kn)^{\frac{2}{3}} \Rightarrow \tilde{d}_e \geq 8kq \end{cases}$	$\frac{13}{2\theta^{1/3}}n^{2/3}$
<i>Sokolov-2007</i> [7]		$\begin{cases} d_e \leq k \Rightarrow \tilde{d}_e \leq \frac{2k(n+2)}{n}, \\ d_e > k \Rightarrow \tilde{d}_e \geq \frac{2k-8}{n} \end{cases}$	$\begin{cases} +\infty & (\theta \leq 5), \\ \frac{n\theta+2}{\theta-5} & (\theta > 5) \end{cases}$
Other approximation algorithms			
<i>Batu-2006</i> [18]	$\min \left\{ n^{\frac{1}{3}+o(1)}, (d_e)^{\frac{1}{2}+o(1)} \right\}$		$4(2c-1) \left(\lg((2c-3)k) + 1 + \frac{(c-1)^2}{c} \right)^\ddagger$
<i>Charikar-2006</i> [20]	$O(n \log n)^{\dagger\dagger}$		$48n(1 + \ln n) / \max\{1, \theta\}$
<i>Andoni-2009</i> [21]	$O(n)^{\dagger\dagger}$		$3400n$
<i>Andoni-2010</i> [19]	$12 \lg n^{\ddagger\dagger}$		$12 \lg n^{\ddagger\dagger}$

Note:

† q denotes the q -gram. In the algorithm, q is set to $n^{2/3}/(2k^{1/3})$.

‡ $c = \max\{(\lg \lg n)/(\lg \lg \lg n), 2\}$.

†† In *Charikar-2006* and *Andoni-2009*, the distortions are derived for the Ulam metric as $O(\log n)$ and $O(1)$, respectively. We multiplied them by $O(n)$ (more precisely, $2n$) so as to be applicable to general strings (Section 4.3.1).

‡† The distortion is shown in the original paper ([19], pp. 16 in the full version).

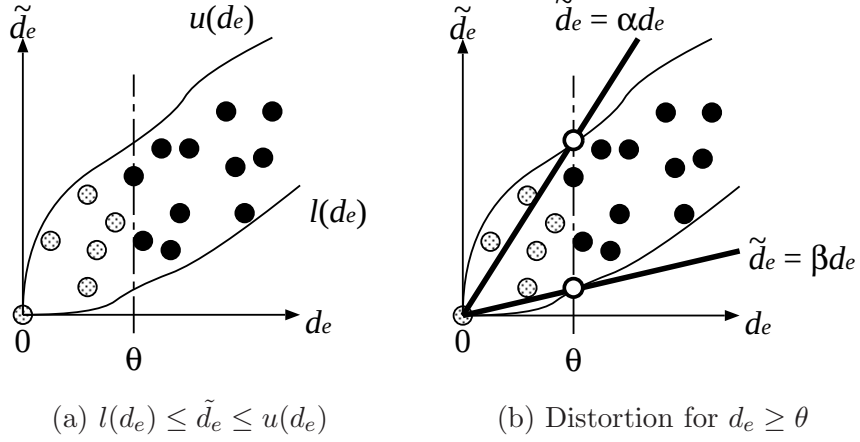


Figure 4: Conversion of lower and upper bounds to a distortion

in $d_e \geq \theta$, respectively, then $K = (\sup_{d_e \geq \theta} u(d_e)/d_e) / (\inf_{d_e \geq \theta} l(d_e)/d_e) \leq (u(\theta)/\theta) / (l(\theta)/\theta) = u(\theta)/l(\theta) = K_\theta$. Therefore we can obtain the distortion when the two monotonicity conditions are confirmed.

For each algorithm whose distortion is given in a big-oh notation (*Batu-2006*, *Charikar-2006*, *Andoni-2009* and *Andoni-2010*), we examined every step in the algorithm. The detailed derivations are given in Appendix A.2.

4.2.3 Comparison of Calculated Distortions

Now we examine the refined distortions shown in Table 1. We note that all these algorithms can be now compared in a unified expression.

First we classify these algorithms in the complexity order. Note that we can assume that θ takes an order between $O(1)$ and $O(n)$ since the edit distance takes a value between 0 and n . Assuming $\theta = O(1)$ as an ordinary case, they are ordered as:

- Sub-logarithmic ($O((\log \log n)^2)$): *Batu-2006*
- Logarithmic ($O(\log n)$): *Andoni-2010*
- Sublinear ($O(n^{\alpha(<1)})$): *Bar-Yossef-2004*
- Linear ($O(n)$): *Sokolov-2007*, *Andoni-2009*
- Super-linear ($O(n \log n)$): *Charikar-2006*

As a result, the q -gram-distance-based two algorithms *Bar-Yossef-2004* and *Sokolov-2007* is not good compared to *Batu-2006* and *Andoni-2010*. Note that, for $\theta = O(n)$, *Charikar-2006* is improved to $O(\log n)$ while others unchanged.

Next let us compare the distortions in more detail. As a result of non-asymptotic distortions we have obtained, we can compare algorithms for every specific value of n . We show the result in Fig. 5. In the figure we set $\theta = n$ (maximum θ) for *Bar-Yossef-2004*, *Charikar-2006* and *Sokolov-2007* to evaluate optimistic distortion values. It is observed that *Batu-2006* outperforms the others if n is large enough. However, when n is not so large, say, $n \leq 300$, *Bar-Yossef-2004* is the best. Such a range of effective n is not obtained until our analyses made clear the constant factors. In addition, the other q -gram-distance-based algorithm *Sokolov-2007* is the second best for $n \leq 30$. As a result, the q -gram-distance-based algorithms works well as approximations of the edit distance for small n .

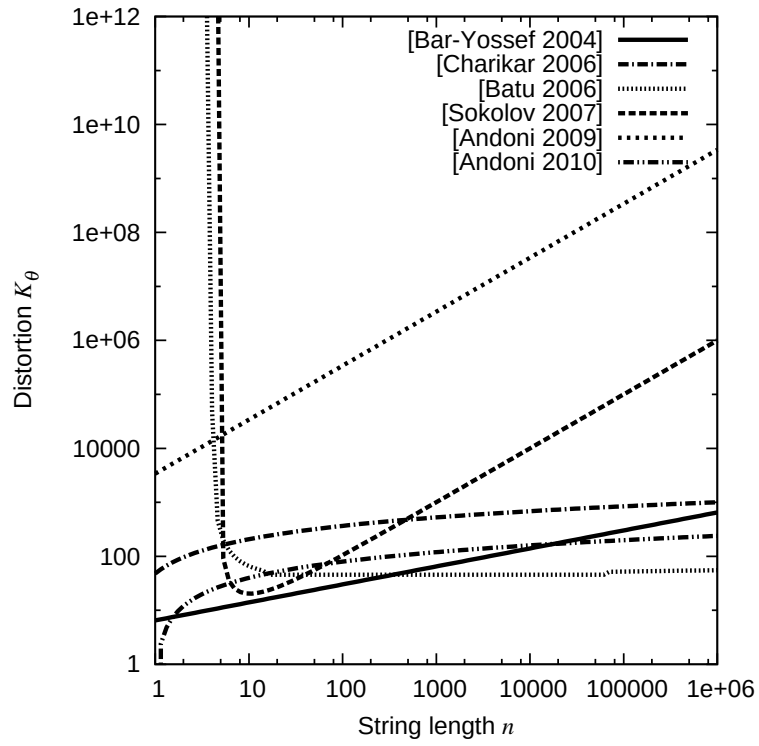


Figure 5: Distortions of six approximation methods.
Note: $\theta = n$ for *Bar-Yossef-2004*, *Charikar-2006* and *Sokolov-2007*.

4.3 Experimental Comparison of Distortions

4.3.1 Procedure

Next we compared them experimentally to know their practical usefulness.

For each data set that will be explained in detail later, we make ready a set S of 10,000 pairs of strings $S = \{(x_1, y_1), (x_2, y_2), \dots, (x_{10000}, y_{10000})\}$. We computed the distortion for S for the six approximation distances.

We used one artificial and two real-life data sets as follows:

Random ($n \in \{100, 300, 1000\}$, $|\Sigma| \in \{4, 20\}$, $e \in \{4, 30\}$):

First we choose x from Σ^n at random with equal probability and initialize y by x . Then we modify y until the total operation cost becomes e : (a) replace a randomly chosen character in y with a randomly chosen character from Σ (probability: $2/3$, cost: 1) or (b) delete a randomly chosen character in y and then insert a randomly chosen character at a randomly chosen position (probability: $1/3$, cost: 2), where all random choices of characters and positions are conducted with equal probability. Note that $d_e(x, y)$ equals e in most cases but can be less than e .

DDBJ ($n \in \{100, 300, 1000\}$):

DDBJ (DNA Data Bank of Japan) is a DNA nucleobase sequence database service [43]. We used “ddbjhum1” data ($|\Sigma| = 15$; 4 of them occupy 99.95%). To unify the string length in each data set, we constructed the data set as follows: For $n = 100$, we gathered strings of length 100 to 299 in ddbjhum1 and truncated the 101st character or after. Similarly, for $n = 300$ and $n = 1000$, we collected strings of length 300 to 999 for $n = 300$ and 1000 to 2999 for $n = 1000$, respectively.

UniProt ($n \in \{100, 300, 1000\}$):

UniProt (Universal Protein Resource) is an amino acid sequence (i.e. protein) database service [44]. We used “UniProtKB-SwissProt” data ($|\Sigma| = 25$; 20 of them occupy 99.99%). We conducted the data set constructions in the same manner as in DDBJ.

For the algorithms assuming the Ulam metric (*Charikar-2006* and *Andoni-2009*, Section 3.2), where all characters in a string are expected to be distinct, we “expanded” the alphabet from Σ to Σ^m for each string pair x, y so

Table 2: Best algorithm according to the alphabet size $|\Sigma|$, the string length n and the number of edits (an upper bound of the edit distance) e .

Edit distance	$ \Sigma = 4$			$ \Sigma = 20$		
	$n = 100$	$n = 300$	$n = 1000$	$n = 100$	$n = 300$	$n = 1000$
$e = 4$						
$e = 30$						
$e \sim n$ (real-life)						

that $(x[1..m], \dots, x[n - m + 1..n])$ are distinct and so do $(y[1..m], \dots, y[n - m + 1..n])$ with as small m as possible. It can be shown that the distortion with this expansion is at most $2m$ times of that under the Ulam metric [20].

When algorithms have parameters (*Bar-Yossef-2004*, *Batu-2006*, *Sokolov-2007* and *Andoni-2010*), we chose the smallest distortions over some candidates of parameters as follows:

- $q \in \{2, 4, 6\}$ for q -grams (*Bar-Yossef-2004* and *Sokolov-2007*³).
- $c \in \{2, 4\}$ and $j = 1$ for *Batu-2006* (see Appendix A.2 for details). As a result, the theoretical distortion of *Batu-2006* is $(2c - 1) \cdot [4c + \{8(2c - 3)k\}^{c-1}] / c = 12[1 + \lceil \lg |\Sigma| \rceil] = 72$ with $c = 2$ and $|\Sigma| = 20$, a constant against n . It needs $O(n^2)$ time.
- Tree node pruning (the trade-off between the computational time and the accuracy) is not conducted on *Andoni-2010* (the highest accuracy). It needs $\Omega(n^2)$ time.

4.3.2 Results

We show the experimental results in Fig. 6, Fig. 7 and Table 2. From Fig. 6 we see that actual values of distortion are far less than their theoretical

³Following the description in the papers *Bar-Yossef-2004* and *Sokolov-2007*, B and q_1 corresponds to q , respectively. In *Sokolov-2007*, parameter q_2 is also set to q .

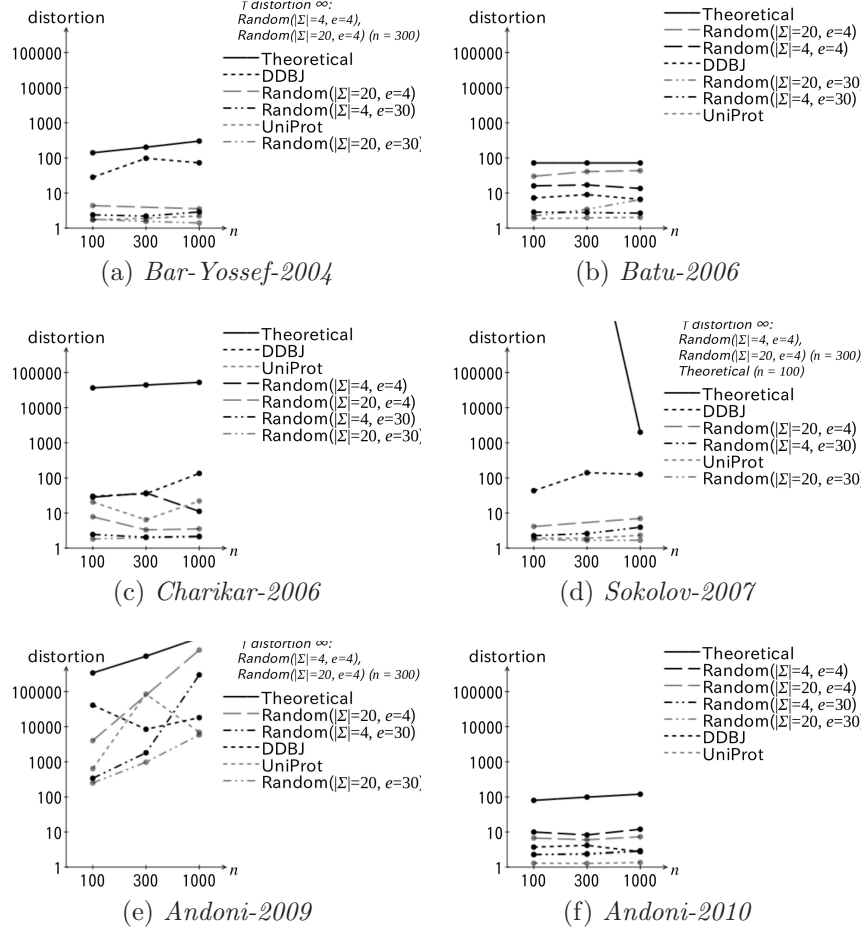


Figure 6: Experimental distortions of six algorithms. Gray lines denote $|\Sigma| = 20$ data sets including UniProt. The theoretical value of *Batu-2006* is different from that in Table 1 (constant against n ; see Section 4.3.1).

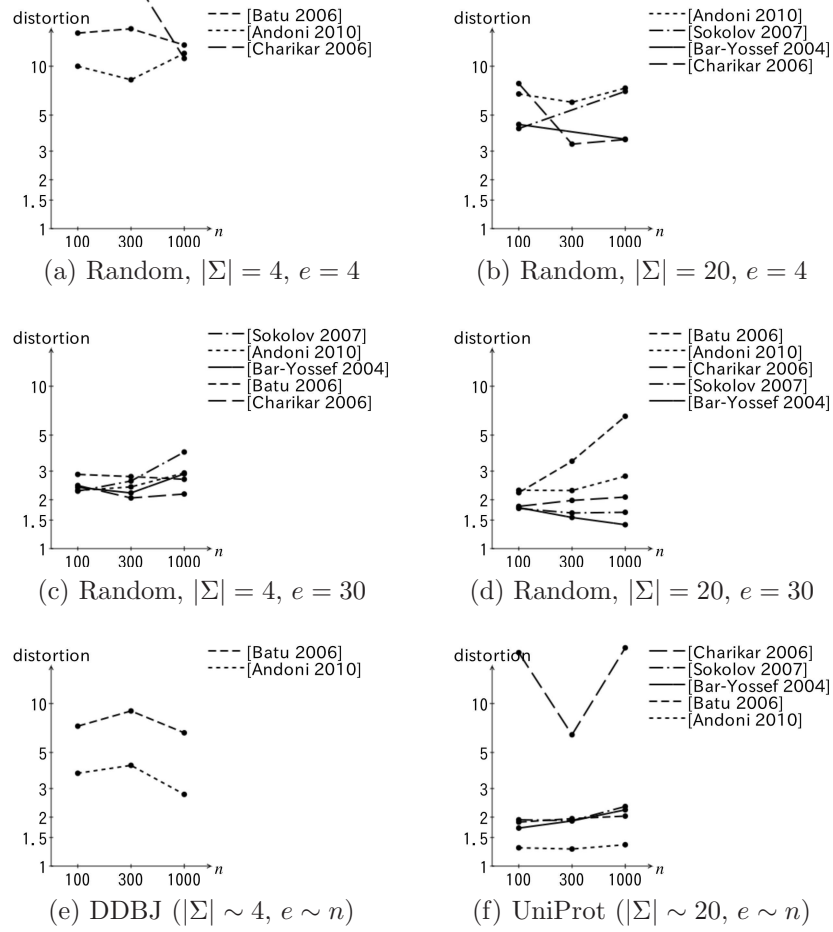


Figure 7: Experimental distortions of six data sets. Distortions larger than 30 are omitted from the charts.

values, often 10 times or more (one scale mark in Fig. 6). This made clear that theoretical distortions are obtained in the worst case but real data are not the case.

Then we list the best algorithms depending on $|\Sigma|$, n and e in Table 2 and the detailed comparison in Fig. 7. We assumed “ $e \sim n$ ” in the two real-life data sets (DDBJ and UniProt) in Table 2, since they contain strings coming from many organic components and thus most string pairs have large (nearly n) edit distance. First, in total, we can see that *Andoni-2010*, theoretically the second best, is almost always the best: it is the best for the two real-life data sets (DDBJ and UniProt) and nearly the best even for Random data set. On the other hand, theoretically the best algorithm *Batu-2006* did not yield the smallest distortion for any data set. Then, focusing on the two q -gram-distance-based algorithms *Bar-Yossef-2004* and *Sokolov-2007*, they become the best for some of the cases of small string lengths $n = 100$ and $n = 300$, as expected from the theoretical results. In addition, we can see that they tend to achieve small distortions for relatively large $|\Sigma|$. This seems to be because, if $|\Sigma|$ is large, then q -grams over Σ appearing in a string become more distinct even if the value of q is small. This means that the effect of appearance order⁴ disappears and thus q -gram distance becomes close to the edit distance.

4.4 Summary of Approximation Accuracy Analysis

In this section, theoretical and experimental analysis showed the condition each of six edit distance approximation algorithms achieves the smallest distortion. Especially, the q -gram-distance-based approximations *Bar-Yossef-2004* and *Sokolov-2007* is proved to yield small distortions for small string length about $n \leq 300$ compared to other approximations. In addition, experimental results implied that they are also advantageous for large alphabet size $|\Sigma|$. For other algorithms, *Batu-2006* is theoretically the best, while *Andoni-2010* is experimentally the best.

⁴A counter example is $x = \text{“abcdefgh”}$ and $y = \text{“efghabcd”}$: the difference of appearance order makes the edit distance be large ($d_e(x, y) = 8 = n$) while the 2-gram distance is small (2).

5 Fast Similar Substring Searching Algorithm by the q -gram Distance

Note: The content of the section is based on the author's work [11].

5.1 Outline

In this section we consider the similar substring searching by the q -gram distance, that is, to find substrings of the text string t whose q -gram distances to the pattern string p are k or less. (A formal definition is given in Section 5.2.) Note that we assume $k = O(|p|)$ since, in general, k takes at most $|p|$ to find the most similar substring. In addition, it is natural to set k as a certain percentage of the length of the pattern string.

As stated in Section 1, currently the best algorithm works in $O(|t| \log k + |p|) = O(|t| \log |p| + |p|)$ time both on average and in the worst case, which was proposed by Ukkonen [6] and which we call *TreeSearch* in this paper. However, seemingly no improvement has been conducted since then. The proposed algorithm improved the average-case time complexity to $O(|t| + |p|)$ without worsening the worst-case time complexity, which we call *ListBaselineSearch* in this paper.

We explain other two algorithms to explain the two algorithms above simpler: Ukkonen's *ArraySearch* [6] and a proposed *ArrayBaselineSearch*, which are simpler but needs $O(|t||p|)$ time in the worst case.

Let us review the time complexities of substring searching problem in various criteria (Table 3). Searching for exactly matching substrings is easier than searching for similar substrings and thus can be computed in $O(|t| + |p|)$ time (e.g., see Knuth-Morris-Pratt algorithm [45]). A *suffix tree* [2, 13] can also be used to achieve this complexity. For the edit distance, as discussed in Section 3.3, $O(|t|k + |p|)$ is the best if k cannot be assumed as a constant and the space is limited to $O(|t| + |p|)$.

Table 3: Computational complexity of similar substring searching in d_e and d_q . Here, t denotes a text, p denotes a pattern and k denotes a threshold of distance.

Distance	Algorithm	Time complexity	
		Average	Worst
Exact	Knuth <i>et al.</i> 1977 [45], Weiner 1973 [13], etc.	$O(t + p)$	$O(t + p)$
d_e	Ukkonen 1985 [23], Landau <i>et al.</i> 1989 [24], etc.	$O(t k + p)$	$O(t k + p)$
d_q	Ukkonen 1992 [6] (<i>ArraySearch</i>)	$O(t k + p)^*$	$O(t k + p)$
	Ukkonen 1992 [6] (<i>TreeSearch</i>)	$O(t \log k + p)^*$	$O(t \log k + p)$
	Proposed <i>ArrayBaselineSearch</i>	$O(t + p)$	$O(t k + p)$
	Proposed <i>ListBaselineSearch</i>	$O(t + p)$	$O(t \log k + p)$

* These complexities have not been explicitly shown in the reference [6] but easily proven.

5.2 Problem Setting

We define the problem of the similar substring searching formally as follows:

Problem Given a *text* string t , a *pattern* string p and two integers q and k , enumerate all $t[i..j^*]$ ($1 \leq i \leq j^* \leq |t|$) in the sense that

$$\begin{aligned} d_q(t[i..j^*], p) &\leq k, \text{ and} \\ d_q(t[i..j^*], p) &\leq d_q(t[i..j], p) \text{ for } \forall j \neq j^*, \end{aligned}$$

where the equality may hold only for $j < j^*$ in the second inequality expression.

Note that the last line guarantees the uniqueness of similar substrings starting from position i and the tie-breaking for the same distance cases. As a result, a solution $s = x[i..j^*]$ is the closest to p among all substrings $x[i..j]$ sharing the starting position i and longest if there are other substrings having the same distance. For the example of $t = \text{"cabaab"}$, $p = \text{"abab"}$, $k = 2$ and $q = 2$, when $i = 2$ then both $d_q(t[2..4], p)$ and $d_q(t[2..6], p)$ achieve the minimum distance $1 (\leq k)$. In this case we choose $j^* = 6$, the longer substring, for $i = 2$.

Comparing the numbers of q -grams in H_x and H_y , it is clear that $d_q(x, y) \geq ||H_x| - |H_y|| = ||x| - |y||$ on condition that $x \geq q$ and $y \geq q$. Therefore, it is clear that the first condition $d_q(x[i..j], p) \leq k$ holds only in a limited range of the value of j for a fixed value of i . Let us call the range the *scope* of i , which is defined as follows:

Definition 5.1. $scope(i) = [b_i, e_i] \stackrel{\text{def}}{=} [i + |p| - 1 - k, i + |p| - 1 + k]$.

We may assume that $j^* \in scope(i)$ for any i .

5.3 Change Point

The concept of the change point is a key for fast searching. This was introduced in Ukkonen's algorithms [6] and is also exploited in the proposed algorithms.

Under the problem setting in Section 5.2, we consider the best ending position j for each beginning position i . Thus let us consider a sequence of

all substrings starting from the i th position

$$\mathbf{t}_{(i)} = (t[i..i], t[i..i+1], \dots, t[i..|t|])$$

and the corresponding sequence of distances to p

$$\mathbf{d}_{(i)} = (d_q(t[i..i], p), d_q(t[i..i+1], p), \dots, d_q(t[i..|t|], p)) .$$

An example of $\mathbf{d}_{(i)}$ is shown in Table 4.

We first focus on the fact that $\Delta_j = d_q(t[i..j+1], p) - d_q(t[i..j], p)$ takes either $+1$ or -1 as shown in Table 4. Before the first q -gram appears in $t[i..j]$ (i.e. $j < i + q - 1$), it is clear that $\mathbf{d}_{(i)}(j) = |H_p| = |p| - q + 1$ because $H_{t[i..j]} = \emptyset$.

Let us examine how $\mathbf{d}_{(i)}$ changes to $\mathbf{d}_{(i+1)}$ when i is incremented to $i+1$, especially whether j^* changes or stays. The difference between $d_q(t[i..j], p)$ and $d_q(t[i+1..j], p)$ is that of $H_{t[i..j]}$ and $H_{t[i+1..j]}$. Precisely speaking, one more q -gram $t_{(i)} = t[i..i+q-1]$ is contained in $H_{t[i..j]}$ than in $H_{t[i+1..j]}$. Therefore, removal of $t_{(i)}$ in $\mathbf{d}_{(i+1)}$ is effective at a certain position j , that is, the value of $d_q(t[i..j], p)$ decreases by one in $d_q(t[i+1..j], p)$ if the number of $t_{(i)}$ in $H_{t[i..j]}$ was already more than the number of $t_{(i)}$ (can be zero) in H_p . Otherwise the distance increases by one. The situation, however, depends on how many $t_{(i)}$ had been stored in $H_{t[i..j]}$ at j and how many $t_{(i)}$ was in H_p . More precisely speaking, for a certain position c , if $t_{(i)}$ had been stored in $H_{t[i..j]}$ at $j = c - 1$ less than the number in H_p and the number becomes the same at $j = c$, then for any $j < c$ the removal of $t_{(i)}$ increases the distance, and for any $j \geq c$ the removal of $t_{(i)}$ decreases the distance. Therefore, we call such a point c a *change point* for i and denote it by c_i . In $\mathbf{d}_{(i+1)}$ compared with $\mathbf{d}_{(i)}$, $\Delta_i = d_q(t[i+1..j], p) - d_q(t[i..j], p) = +1$ for $j < c_i$ and $\Delta_i = -1$ for $j \geq c_i$. An example is shown in Fig. 8. We define the change points formally as follows.

Definition 5.2. [6] *The change point c_i in the distance sequence $\mathbf{d}_{(i)}$ is the position j such that*

$$\Delta_i(j) = d_q(t[i+1..j], p) - d_q(t[i..j], p) = \begin{cases} +1 & (\text{for } i+q \leq j < c_i), \\ -1 & (\text{for } j \geq c_i). \end{cases} \quad (1)$$

We have the following properties on the change points.

Property 5.1. [6] Let $s = t_{(i)}$ be the first q -gram in $\mathbf{t}_{(i)}$ and $m = \#H_p(s)$ be the number of occurrences of s in p . The change point c_i is the position of the last character at which s occurs for the $(m + 1)$ -th time in $t[i..|t|]$. Especially, $c_i = i + q - 1$ for $m = 0$ and $c_i = +\infty$ for when $t[i.., |t|]$ includes s for m times or fewer. Clearly, $t_{<c_i>} = s$ except for $c_i = +\infty$.

Since c_i is obtained as the $(\#H_p(t_{(i)}) + 1)$ -th q -gram starting from i , it is easily shown that each of the change points $\{c_i \mid i = 1, \dots, |t| - q + 1\}$ can be obtained in $O(1)$ time by classifying all of the positions $\{j \mid q \leq j \leq |t|\}$ by q -grams $t_{<j>}$ beforehand [6].

Table 4: The q -gram distance sequence $\mathbf{d}_{(i)}$ for $i = 1, 2, \dots, 12$ for $t = \text{"aaaccaababbc"}$ and $p = \text{"aaabbc"}$ with $q = 2$ and $k = 3$. The value at (i, j) represents $\mathbf{d}_{(i)}(j) = d_q(t[i..j], p)$. Here, $t_{(j)}$ is the q -gram starting from the j th position and $t_{<j>}$ is the q -gram ending at the j th position.

j	1	2	3	4	5	6	7	8	9	10	11	12
$t_{(j)}$	aa	aa	ac	cc	ca	aa	aa	ab	ba	ab	bc	
$i \setminus t_{<j>}$		aa	aa	ac	cc	ca	aa	aa	ab	ba	ab	bc
1	5	4	3	4	5	6	7	8	7	8	9	8
2		5	4	5	6	7	6	7	6	7	8	7
3			5	6	7	8	7	6	5	6	7	6
4				5	6	7	6	5	4	5	6	5
5					5	6	5	4	3	4	5	4
6						5	4	3	2	3	4	3
7							5	4	3	4	5	4
8								5	4	5	6	5
9									5	6	5	4
10										5	4	3
11											5	4
12												5

Histogram of
 q -grams in p :
 aa: ■■ 2
 ab: ■ 1
 bb: ■ 1
 bc: ■ 1

- **Shaded** : $j \in \text{scope}(i) = [b_i, e_i] = [p|i-i-k-1, p|i+i-k-1]$
- **Bold**: $j = c_i$ (change point)

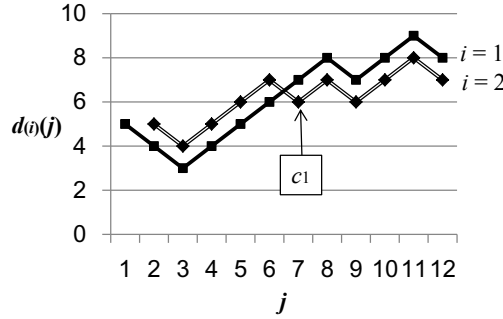


Figure 8: The q -gram distance sequences $\mathbf{d}_{(1)} = \{d_q(t[1..j], p)\}$ and $\mathbf{d}_{(2)} = \{d_q(t[2..j], p)\}$ for $t = \text{"aaaccaababbc"}$ and $p = \text{"aaabbc"}$ with $q = 2$. Before the change point $c_1 = 7$, $\mathbf{d}_{(2)}(j)$ is larger than $\mathbf{d}_{(1)}(j)$ by one; otherwise it is smaller than $\mathbf{d}_{(1)}(j)$ by one.

5.4 Ukkonen's Two Algorithms

As the basis of the proposed algorithm, let us glimpse Ukkonen's two algorithms [6] called *ArraySearch* and *TreeSearch* in this paper. They run in $O(|t|k + |p|)$ and $O(|t| \log k + |p|)$ time, respectively.

The common strategy of the two algorithms is to keep and update efficiently the distance information only in $scope(i)$ as

$$\mathbf{d}_{(i)} = (d_q(t[i..b_i], p), d_q(t[i..b_i + 1], p), \dots, d_q(t[i..e_i], p)).$$

Here, the same notation $\mathbf{d}_{(i)}$ is used even when the range is restricted to $scope(i)$.

5.4.1 Algorithm *ArraySearch*

In the algorithm *ArraySearch*, $\mathbf{d}_{(i)}$ is stored in an array $D = \{D[j] \mid j \in scope(i)\}$ as $D[j] = \mathbf{d}_{(i)}(j)$. We initialize D with $i = 1$ just by calculating the q -gram distances to p for $j \in [b_1, e_1] = [|p| - k, |p| + k]$.

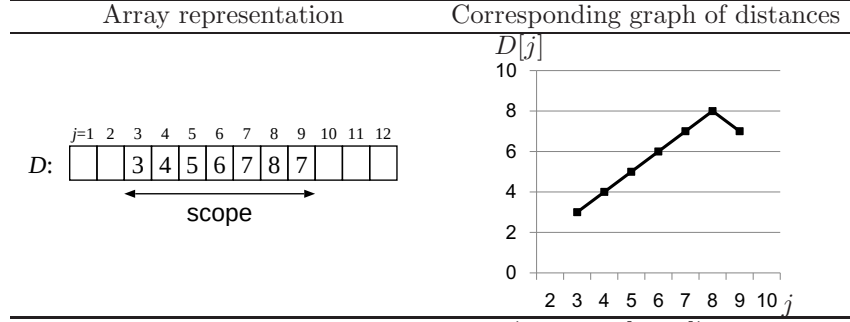
When we increment i to $i + 1$, we update the value of $D[j]$ as follows, noticing that the scope is also changed from $[b_i, e_i]$ to $[b_{i+1}, e_{i+1}] = [b_i + 1, e_i + 1]$: (1) erase the value at $D[b_i]$, (2) increase $D[j]$ by one for every j such that $b_i + 1 \leq j < e_i$ and decrease $D[j]$ by one for every j such that $e_i \leq j \leq e_i$, and (3) calculate $D[e_{i+1}]$ by $D[e_{i+1}] = D[e_i] - 1$ if $H_{t[i+1..e_{i+1}]}(t_{<e_{i+1}>}) \leq H_p(t_{<e_{i+1}>})$, otherwise by $D[e_{i+1}] = D[e_i] + 1$. We do the first step just for cleaning up, so a cyclic array of size $2k + 1$ can be used instead. An example is shown in Fig. 9.

Step (2) needs $O(k)$ steps since $|e_i - b_i + 1| = 2k + 1$. Searching for j^* is also carried out in $O(k)$. Therefore, in total, we need $O(k|t|)$ steps. Note that computing $H_{t[i+1..e_{i+1}]}(t_{<e_{i+1}>})$ and $H_p(t_{<e_{i+1}>})$ in step (3) can be done in $O(1)$ time (not $O(q)$ time) with the suffix tree of p (see [6] for details). Thus the updating procedure can be done in $O(|t|k)$ time, and the total time complexity of *ArraySearch* is $O(|t|k + |p|)$, adding the time to build the suffix tree of p .

5.4.2 Algorithm *TreeSearch*

In *TreeSearch*, $\mathbf{d}_{(i)}$ is stored in a binary search tree T instead of an array. Typically, we use a binary self-balancing search tree such as an AVL Tree

Initial array ($i = 1$, scope: $[3, 9]$)



Update of the array for $i = 1$ to 2 (scope: $[4, 10]$)

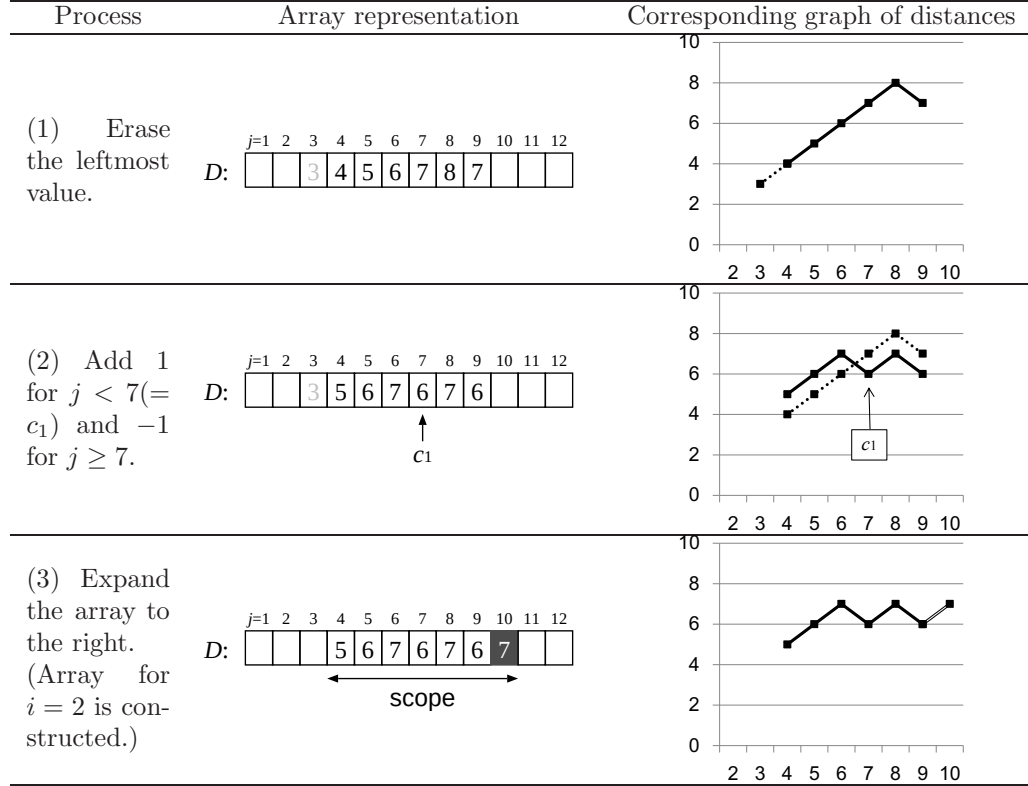


Figure 9: Update procedure of *ArraySearch* in the example of Table 4. The array D stores $D[j] = d_{(i)}(j) = d_q(t[i..j], p)$ for the current i . It is updated for $i + 1$ by (1) erasing the value of $D[b_i]$, (2) adding 1 to every $D[j]$ if $j < c_i$ and -1 otherwise, and (3) calculating the value of $D[c_{i+1}]$.

or a red-black tree [46] (Fig. 10). As in *ArraySearch*, we assume i is fixed and show the update process of T_i from i to $i + 1$.

The position j^* appears at the root node of T by obtaining information from its descendants in a bottom-up way as explained below.

A leaf keeps the difference in two distances as $\Delta(j) = D[j + 1] - D[j]$, thus, ± 1 with the position number j such as $(j, \Delta(j))$. At an inner node, we have the following quadruplet:

- $J = [j_1..j_2]$: interval specified by leaves under the node,
- $\Delta(J) = \sum_{\ell=j_1}^{j_2} \Delta(\ell)$: sum of differences in the interval $[j_1, j_2]$,
- $j^*[J]$: index achieving the minimum sum in this interval (can be $j_1 - 1$ if $\Delta([j_1..\ell]) > 0$ for any $\ell \in J$), and
- $\Delta(j^*[J]) = \sum_{\ell=j_1}^{j^*[J]} \Delta(\ell)$: achieved minimum sum.

Since the root node of T_i has all of the leaves corresponding to $j \in \text{scope}(i)$, it is clear that the root node represents $j^* = j^*[b_i..e_i]$. In addition, to obtain the true distance from the distance differences, it is enough to keep $o = D[b_i - 1]$ as the baseline (Fig. 10).

Thus, in the update stage of T_i to T_{i+1} , it is sufficient to show how to (1) remove the leftmost leaf, (2) search for the change point and (3) add the rightmost leaf, as well as the update procedure of all information stored in each node. An example is shown in Fig. 11.

The information at an inner node n is calculated from its left child lc and right child rc (Fig. 10). Let us distinguish the quadruplets of these nodes by its suffix such as J_n for the node n . Then, we have

- $J_n \leftarrow J_{lc} \cup J_{rc}$
- $\Delta(J)_n \leftarrow \Delta(J)_{lc} + \Delta(J)_{rc}$
- $j^*[J]_n \leftarrow \begin{cases} j^*[J]_{lc} & \text{if } \Delta(j^*)_{lc} < \Delta(J)_{lc} + \Delta(j^*[J])_{rc} \\ j^*[J]_{rc} & \text{otherwise} \end{cases}$
- $\Delta(j^*[J])_n \leftarrow \begin{cases} \Delta(j^*[J])_{lc} & \text{if } \Delta(j^*)_{lc} < \Delta(J)_{lc} + \Delta(j^*[J])_{rc} \\ \Delta(J)_{lc} + \Delta(j^*[J])_{rc} & \text{otherwise} \end{cases}$

where $j^*[J]$ is briefly written as j^* as long as it is clear from the node suffix.

When a node is changed, deleted or added, we update all of the nodes in the middle of the path from the node to the root according the procedure shown above. It costs $O(\log k)$ because of its balance property. It requires another $O(\log k)$ for searching from the root to the leaf corresponding to the change point c_i . Therefore, in total, *TreeSearch* needs $O(|t| \log k + |p|)$ time.

Initialized tree ($i = 1$, $scope(i) = [3, 9]$)

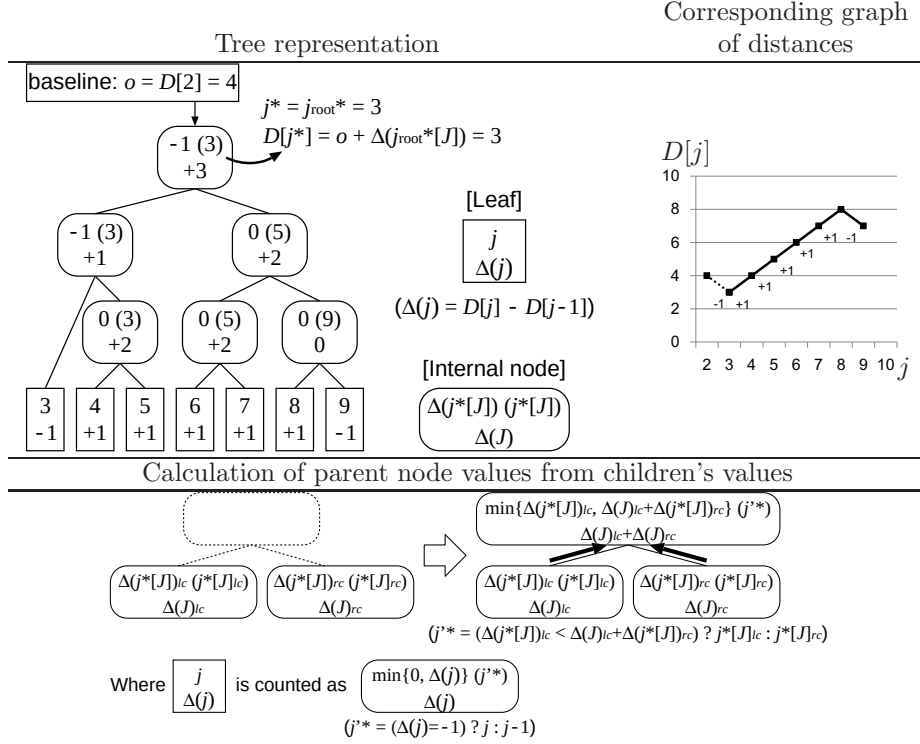


Figure 10: The binary search tree used in *TreeSearch* for the example of Table 4. This tree represents $D[j] = d_q(t[i..j], q)$ (right graph) effectively. A leaf holds a pair $(j, \Delta(j))$ ($\Delta(j) = D[j] - D[j-1]$), an internal node shows the local minimum value $\Delta(j^*[J]) = D[j^*[J]] - o$ and the position $j^*[J]$. For example, the left child of the root node shows that the minimum value $\Delta(j^*[J]) = -1$ among $J = \{3, 4, 5\}$ is obtained at $j^*[J] = 3$. The third value in the node is the sum of $\Delta(j)$'s of the descendants. The lower figure shows how to calculate these values from its children.

Update of the tree for $i = 1$ to 2 (scope: $[4, 10]$)

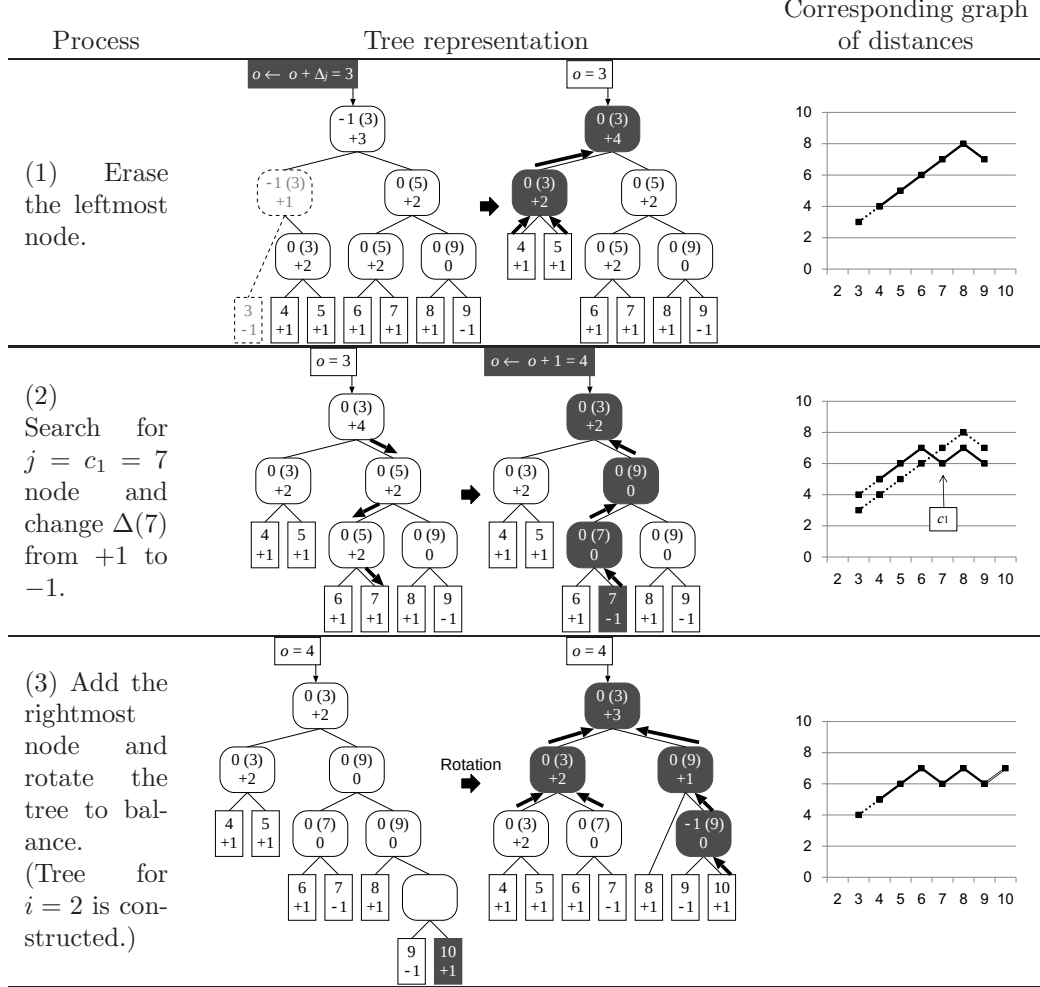


Figure 11: Update procedure of *TreeSearch* in the example of Table 4. The tree is updated from i to $i + 1$: (1) remove the leftmost node and change the value of baseline by $o \leftarrow o + \Delta(j)$ for $j = b_i$, (2) search for the change point c_i and change the value of $\Delta(j)$ from +1 to -1 at $j = c_i$ and (3) add a new node ($j = e_{i+1}$) to rightmost. In each of the three steps, we keep the consistency on the path from the deleted, added, or modified leaf to the root by recalculating the values on the path in a bottom-up way. After these steps, we rotate the tree counterclockwisely to maintain the balance.

5.5 Proposed Algorithm *ArrayBaselineSearch*

Before explaining the fastest algorithm *ListBaselineSearch* ($O(|t| + |p|)$ time on average and $O(|t| \log k + |p|)$ time in the worst case), we consider a simpler algorithm *ArrayBaselineSearch*, a simple improvement upon existing *ArraySearch*, to evaluate the computational time of *ListBaselineSearch* easier. *ArrayBaselineSearch* runs in $O(|t| + |p|)$ time on average and $O(|t|k + |p|)$ ($= O(|t||p|)$) time in the worst case.

The common idea of the improvement for both *ArrayBaselineSearch* and *ListBaselineSearch* comes from the fact that, if c_i is outside $scope(i + 1) = [b_{i+1}, e_{i+1}]$, then j^* for $i + 1$ must be either of: (1) unchanged from that for i or (2) changed to e_{i+1} . Thus, in this case, we have only to compute a constant number of q -gram distances.

More specifically, in the case of (1), $\mathbf{d}_{(i+1)}(j) = \mathbf{d}_{(i)}(j) - 1$ for any $j \in scope(i + 1)$ if $c_i < b_{i+1}$ and $\mathbf{d}_{(i+1)}(j) = \mathbf{d}_{(i)}(j) + 1$ if $c_i > e_{i+1}$, and thus j^* is the same for i and $i + 1$. This means that we may update only the value of a “baseline” o by increasing/decreasing one instead of all values of $\mathbf{d}_{(i)}$ to have $\mathbf{d}_{(i+1)}$. The actual value of $\mathbf{d}_{(i)}(j)$ is $o + \mathbf{d}_{(i)}(j)$, that is, now $\mathbf{d}_{(i)}$ holds “offset” distances from the baseline o .

Since increasing/decreasing the value of o is made in $O(1)$, the complexity is dramatically reduced for the problems in which the change points seldom exist in the scope.

The algorithm *ArrayBaselineSearch* is obtained by applying the discussion above to *ArraySearch*. In *ArrayBaselineSearch* a baseline variable o is introduced in addition to the distance array D adopted in Ukkonen’s *ArraySearch* algorithm. The idea is simple: use D to store the “offset” values from the baseline o and increase or decrease the value of o instead of updating all of the values of D if they are equivalent. The judgment is made by checking whether $c_i \in scope(i + 1)$ or not. The algorithm *ArrayBaselineSearch* is described formally in Fig. 12.

Then, we can show the following probabilistic evaluation of the complexity of this algorithm.

Theorem 5.1. *Let α be the probability that the event $c_i \in scope(i + 1)$ happens. Then *ArrayBaselineSearch* runs in $O(\alpha k|t| + |t| + |p|)$ time on average and $O(|t|k + |p|)$ in the worst case.*

Note that, in the case of fixed t , α is defined as $\alpha \stackrel{\text{def}}{=} |\{i \mid c_i \in scope(i + 1)\}| / |t|$.

```

Algorithm ArrayBaselineSearch( $t, p$ : strings,  $k, q$ : integers)
    // Initialize the array for  $i = 1$ .
1    $i \leftarrow 1$ .
2    $b \leftarrow |p| - k$ ;  $e \leftarrow |p| + k$ .
3   Calculate  $D[j] = d_q(t[1..j], q)$  for every  $j \in [b_1, e_1]$ .
4    $o \leftarrow D[j^*]$  with  $j^* = \underset{j}{\operatorname{argmin}} D[j]$ .
5    $j^* \leftarrow \underset{j \in [b, e]}{\operatorname{argmin}} D[j]$ .
6   if  $o + D[j^*] \leq k$  then output  $t[1..j^*]$ .

    // Find similar substrings for  $i \geq 2$ 
7   while  $i < |t| - q + 1$  do
8       Calculate the change point  $c_i$ .
9        $i \leftarrow i + 1$ 
10       $b \leftarrow b + 1$ ;  $e \leftarrow e + 1$ 
11      if  $c \in [b, e]$  then :
12           $D[j] \leftarrow D[j] + 1$ ;  $\forall j < c_i$ ;    $D[j] \leftarrow D[j] - 1$ ,  $\forall j \geq c_i$ .
13           $j^* \leftarrow \underset{j \in [b, e]}{\operatorname{argmin}} D[j]$ .
14      else if  $c < b$  then :
15           $o \leftarrow o - 1$ .
16      else : //  $c > e$ 
17           $o \leftarrow o + 1$ .
18           $D[e] \leftarrow H_{t[i..e]}(t_{<e>}) \leq H_p(t_{<e>}) ? D[e] - 1 : D[e] + 1$ .
19          if  $D[e] \leq D[j^*]$  then  $j^* \leftarrow e$ .
20          if  $o + D[j^*] \leq k$  then output  $t[i..j^*]$ .

```

Figure 12: Algorithm *ArrayBaselineSearch*

$1), 1 \leq i \leq |t| - q\} / (|t| - q).$

Proof of Theorem 5.1. We evaluate the computational costs separately according to the above two cases. When the change point is outside the scope, $O(1)$ suffices to change the value of o and add $D[e_{i+1}]$. Otherwise we need $O(k)$ for updating D and for searching for j^* in $scope(i + 1)$ because $|scope(i + 1)| = 2k + 1$. Thus the expected computation time becomes

$$\alpha|t| \cdot O(k) + (1 - \alpha)|t| \cdot O(1) = O(\alpha k|t| + |t|).$$

In the worst case of $\alpha = 1$, we have $O(k|t|)$. Adding the complexity $O(|p|)$ for constructing a suffix tree of p , we have the theorem. $\square$

The following corollary is obvious.

Corollary 5.1. *ArrayBaselineSearch runs in $O(|t| + |p|)$ time if $\alpha = O(1/|t|)$ or $\alpha = O(1/|p|)$ provided that $k = \Theta(|p|)$.*

It should be reminded that we assume that k increases linearly in $|p|$. In this sense, the condition means that the probability α of the event $c_i \in scope(i + 1)$ has to decrease at speed of at least $1/|t|$ or $1/|p|$. In either case, we can ignore the first term of the complexity in Theorem 1. The first condition is met, for example, when the number of events $c_i \in scope(i + 1)$ is constant regardless of the length of the text. Then α decreases in inverse proportion to $|t|$ as $|t|$ increases.

The second condition can be satisfied by increasing the value of q in proportion to the increasing value of $|p|$. For simplicity, suppose that every q -gram appears in a text randomly and uniformly at probability $1/|\Sigma^q|$. The occurrence of $c_i \in scope(i + 1)$ implies that $t_{(i)} \in \Sigma^q$ appears again at some $j \in scope(i + 1)$ as $t_{<j>}$. Therefore, we evaluate the probability that a q -gram appears at least once in a range of length $2k + 1$ as an upper bound of α (the probability $c_i \in scope(i + 1)$ occurs). Such a probability decreases exponentially for increasing value of q . Therefore, to make the second condition hold, we have to decrease the value of α by increasing the value of q when the length of p increases. For example, we may take q such that $q = 2 \log_{|\Sigma|} |p|$. This situation is stated formally as follows:

Corollary 5.2. *Let t consist of characters chosen randomly according to some distribution over Σ . Then ArrayBaselineSearch runs in $O(|t| + |p|)$*

time on average if $q = \omega(\log |p|)$ for $k = \Theta(|p|)$, more precisely, if $q \geq 2 \log_{1/r_{max}} |p|$, where r_{max} is the maximum probability of occurrence of characters over Σ .

This corollary holds for a slightly wider condition of sampling as shown below. Let us remind first the fact that the probability of a certain q -gram g appearing in a text does not depend on the position in the text as long as the text is composed of characters randomly chosen from Σ with/without replacement, or equivalently, when we consider all possible permutations of a fixed text [47]. The probability g is found in any position in t , denoted by $\Pr(g)$, is the product of matching probabilities of all characters of g . Especially the probability is $1/|\Sigma|^q$ when all characters appear in the same probability $1/|\Sigma|$.

Proof of Corollary 5.2. Let us assume that text t is generated by sampling characters according to some distribution over Σ and let $r_{max} = \max_{c \in \Sigma} \Pr(c) < 1$ (otherwise the problem becomes trivial). Then any q -gram g satisfies $\Pr(g) \leq r_{max}^q$. The condition $q = \omega(\log |p|)$ means that $q \geq \log |p| \cdot c_0$ for every constant $c_0 > 0$. We can specify the value of c_0 when the occurrence probabilities of characters are known. Take $c_0 = 2/(\log 1/r_{max})$, then we can have $q \geq 2 \log |p| / \log(1/r_{max}) = \log_{1/r_{max}}(1/|p|^2)$. This means $r_{max}^q \leq 1/|p|^2$. The “small-omega” notation generalizes this specification for any value of $2/\log(1/r_{max})$. Especially it means $|\Sigma|^q \geq |p|^2$ for $r_{max} = 1/|\Sigma|$ (uniformly distributed case).

Since a necessary condition of this complexity is that a q -gram $s = t_{(i)}$ appears at least once again in the range of $scope(i + 1)$, we bound the probability under the assumption of random sampling:

$$\begin{aligned}
\alpha &= \Pr(c_i \text{ appears in } [b_{i+1}, e_{i+1}]) \\
&\leq \Pr(s = t_{(i)} \text{ appears at least once again in } 2k + 1 \text{ positions}) \\
&\leq \Pr(t_{(b_{i+1})} = s) + \Pr(t_{(b_{i+1}+1)} = s) + \cdots + \Pr(t_{<e_{i+1}>} = s) \\
&= r_{max}^q + r_{max}^q + \cdots + r_{max}^q \\
&= (2k + 1)r_{max}^q \\
&= \Theta(|p|)r_{max}^q \\
&= O(1/|p|) \quad (\text{from } r_{max}^q \leq 1/|p|^2)
\end{aligned}$$

This gives the proof from Corollary 1. □

As a special case, this corollary means that if every character forming a text t is chosen randomly and uniformly, then it suffices to take q such that $|\Sigma|^q \geq |p|^2$ to attain a linear-time complexity. The practical meaning is that we can expect a linear time if we take a large value of q almost satisfying $|\Sigma|^q \geq |p|^2$ for a given p and Σ .

5.6 Proposed Algorithm *ListBaselineSearch*

Let us move to the main algorithm *ListBaselineSearch*, whose time complexity is $O(|t| + |p|)$ on average and $O(|t| \log k + |p|)$ in the worst case.

Ukkonen's *TreeSearch* algorithm requires $O(|t| \log k + |p|)$ time even in the average case, that is, requires $O(\log k)$ time for each i even if $c_i \notin \text{scope}(i+1)$ ((1) and (3) in Fig. 11). This is because we have to remove and add elements in a search tree. Thus, to reduce the time complexity to $O(1)$ for $c_i \notin \text{scope}(i+1)$ case, *ListBaselineSearch* use a doubly-linked list supported by an array and a search tree instead of a single search tree, called *LsAT* in this paper.

5.6.1 Outline

The outline of *LsAT* is as follows. The list L , called the *candidate list*, efficiently holds the current j^* for i and all of the candidates of j^* for the future $i+1, i+2, \dots$. In Fig. 13, the current j^* , designated as the first element of L , is 6 and the next j^* is 6, 9 or 10 for $i+1$. A link enables us to do deletion and insertion of an element in $O(1)$ time as long as the position of the element is known, but searching for the change point c_i needs $O(k)$ (maximum size of the link being $2k - 1$). Thus we implement them by an array A and a tree T . The array A helps to reduce the complexity of searching for c_i in L into $O(\log k)$ by allowing a binary searching. However, such a fast searching becomes impossible once some list elements are removed, causing some holes (empty cells) in A . In Fig. 13, there are six holes at array indices 2, 3, 6, 7, 12 and 13. To compensate this defect, we use a binary self-balancing search tree T , in a higher level, so as to keep the ranges of array chunks (sets of consecutive non-empty elements) as the array indices (denoted by *IndexRange*) and the range of list values (denoted by *ValueRange*⁵). With the help of T , we can conduct a binary search on a target chunk. For example, if we want to find the position $c_i = 15$ or the next larger one in Fig. 13, we first search for c_i in T by *ValueRange* (starting from the root $(6, 10]$ and then moving to the right and finding $(10, 20]$), then search the chunk $A[8..11]$, specified by *IndexRange*, with binary searching. Note that the array index does not cause $O(k)$ -time element insertion since element

⁵ *ValueRange* stores a semi-open interval because, if c_i is not in L , we have to find the next larger element of L . See the algorithm **DEL-MIDDLE** in Figs. 15 and 18.

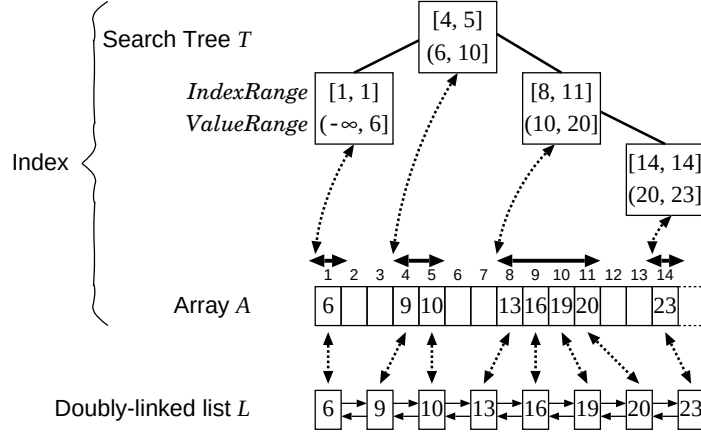


Figure 13: Data structure of LsAT: A doubly-linked list L supported by an array A and a search tree T . Each array cell stores a list element as a pointer, and each tree node stores a connected array range by both array indices and values.

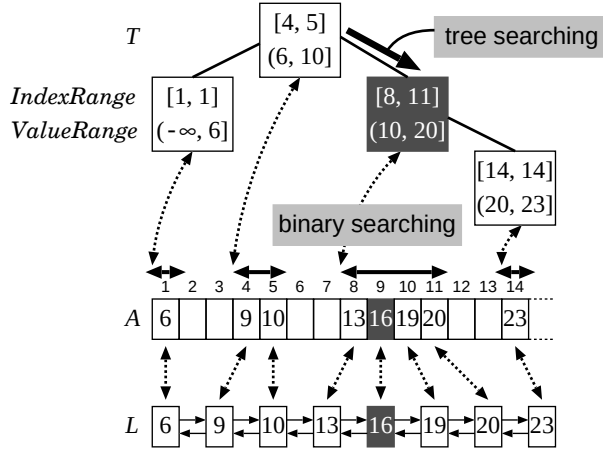


Figure 14: Searching for a change point in an LsAT (example of finding $c_i = 15$ or the next larger element). First we search T for the node including c_i by comparing $ValueRange$ and then search $IndexRange$ of A with binary searching. Finally we reach the corresponding element in L .

insertions are done only at the end in the algorithm (explained later). In addition, we can implement A by a cyclic array with length $2k - 1$ since the right end of A can be extended by at most one for an increment of i , and the left end is removed then.

We still need $O(\log k)$ time for some operations, such as deletion of a node in T , even if we use this special data structure LsAT. However, as will be described in detail in the proof of Theorem 5.2, it can be shown that the cost of necessary operations for the case of $c_i \notin \text{scope}(i + 1)$ is less than $O(\log k)$ on average.

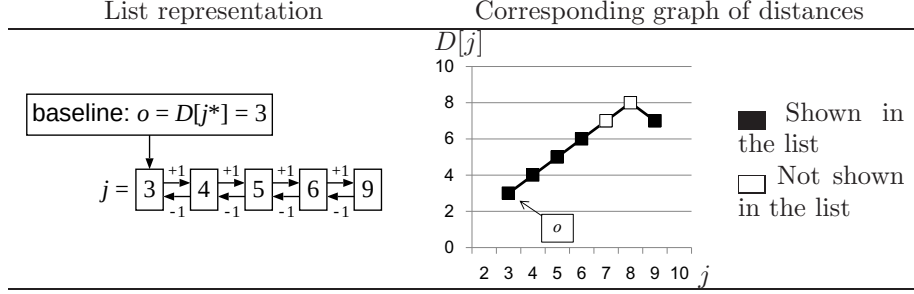
In the following, it will be explained how to maintain the candidate list L according to several possible cases and then to analyze the time complexity. The way to maintain the auxiliary array and tree is described in Appendix B.

Let us explain in detail what each link in the list L represents. Suppose that a distance (offset) sequence $D[j] = \mathbf{d}_{(i)}(j)$ is given for a fixed i and the position j^* in $\text{scope}(i)$ achieving the minimum distance is known. Let us consider the case that the position j^* changes if we change the starting position from i to $i + 1$. This means $j^* < c_i$, otherwise j^* stays at the same position from the definition of the change point. After incrementing i , the value of $D[j]$ is decreased by one at every position $j \geq c_i$, and is increased by one at $j < c_i$. That is, the difference of two occurs in both sides of the boundary c_i . Let us denote the new position of j^* by j^{**} , that is, j^{**} turns j^* once we increase the value of i by one. Since $D[j^*] < D[j^{**}]$ at present, $D[j^*]$ increases by one and $D[j^{**}]$ decreases by one after incrementing i , we can conclude that $D[j^{**}] = D[j^*] + 1$ or $D[j^{**}] = D[j^*] + 2$. Taking this conclusion into consideration, we store in L the position j 's in $\text{scope}(i)$ attaining $D[j^*], D[j^*] + 1, D[j^*] + 2, \dots$ in order to hold the future candidates of j^* .

Definition 5.3. A candidate list L for i is a doubly-linked list such that

- (1) the first element of L is j^* ,
- (2) the last element of L is e_i ,
- (3) the r th element of L , denoted by j^r , satisfies $D[j^r] - D[j^*] = r - 1$,
and

Initial doubly-linked list L ($i = 1, j \in [3, 9]$)



Update of L for $i = 1$ to 2 ($j \in [4, 10]$)

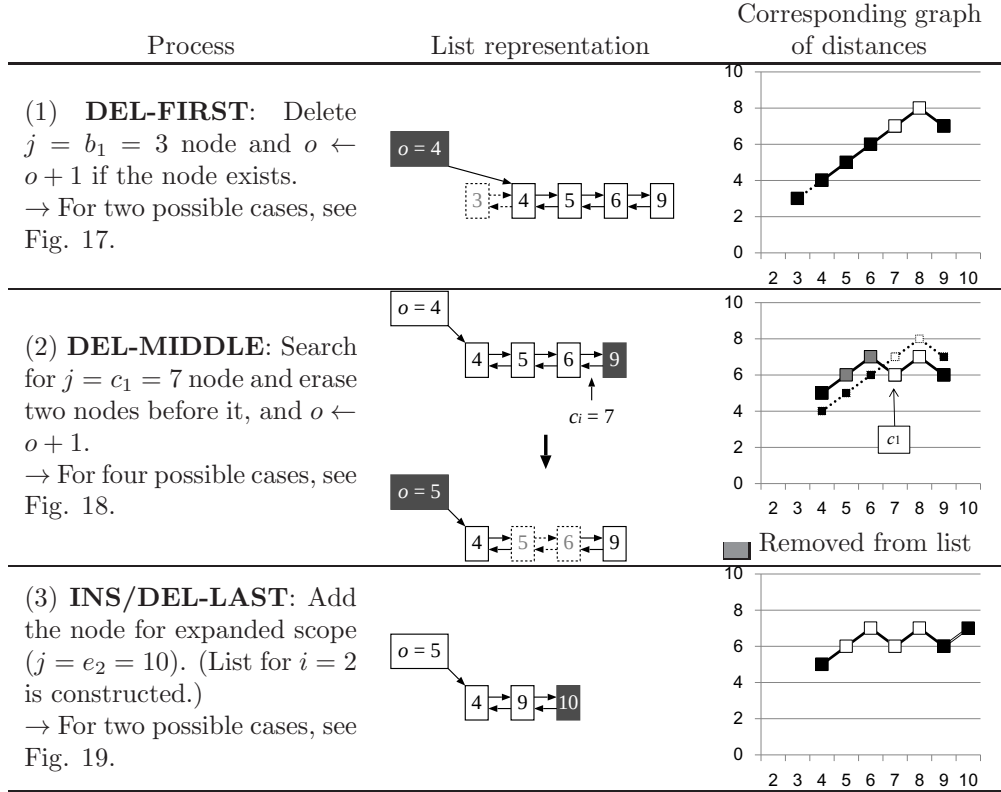


Figure 15: Update procedure of LsAT in the example of Table 4. The list L in i starts with j^* (the largest position attaining the minimum distance o) and then j^2 (the largest positron attaining the distance $o + 1$) and so on, within $scope(i)$. L is updated for i to $i + 1$ as follows: (1) if b_i exists in L as the beginning of L , then remove it from L , A and T , (2) search T and A to locate c_i in L then delete/change a necessary number of preceding elements to keep the consistency of L and change o if necessary (in this example, two previous elements are deleted), and (3) add a new node for $j = e_{i+1}$ to the end of L , or remove the last node of L and change the new last element, to keep L consistent.

```

Algorithm ListBaselineSearch( $t, p$ : strings,  $k, q$ : integers)
    // Data structure LsAT is of (list  $L$ , array  $A$ , tree  $T$ )
    // Find a similar substring for  $i = 1$ 
1    $i \leftarrow 1$ .
2    $b \leftarrow |p| - k$ ;  $e \leftarrow |p| + k$ .
3   Calculate distance  $d_q(t[1..j], p)$  for all  $j \in [b, e]$ .
4   Set the value of  $o$  by the minimum distance.
5   Initialize  $L$  by connecting the largest positions attaining
    the distances  $o, o + 1, \dots, o + l$ , where  $l$  satisfies
     $d_q(t[1..e], p) = o + l$ .
6   Initialize array  $A$  and tree  $T$  according to  $L$ .
    // Find similar substrings for  $i \geq 2$ 
7   while  $i < |t| - q + 1$  do
8       Calculate the change point  $c_i$ .
9        $i \leftarrow i + 1$ .
10       $b \leftarrow b + 1$ ;  $e \leftarrow e + 1$ .
11      if  $c \in [b, e]$  then :
12          Search for the element of  $L$  equal to  $c_i$  or
            the next larger one.
13          Execute DEL-FIRST in LsAT.
14          Execute DEL-MIDDLE in LsAT.
15          Execute INS/DEL-LAST in LsAT.
16      else : //  $c < b$  or  $c > e$ 
17          Execute DEL-FIRST in LsAT.
18          Execute INS/DEL-LAST in LsAT.
19      if  $o \leq k$  then output  $t[i..j^*]$ , where  $j^*$  is the 1st element of  $L$ .

```

Figure 16: Algorithm *ListBaselineSearch*. Detailed procedures of **DEL-FIRST**, **DEL-MIDDLE** and **INS/DEL-LAST** are shown in Appendix B.

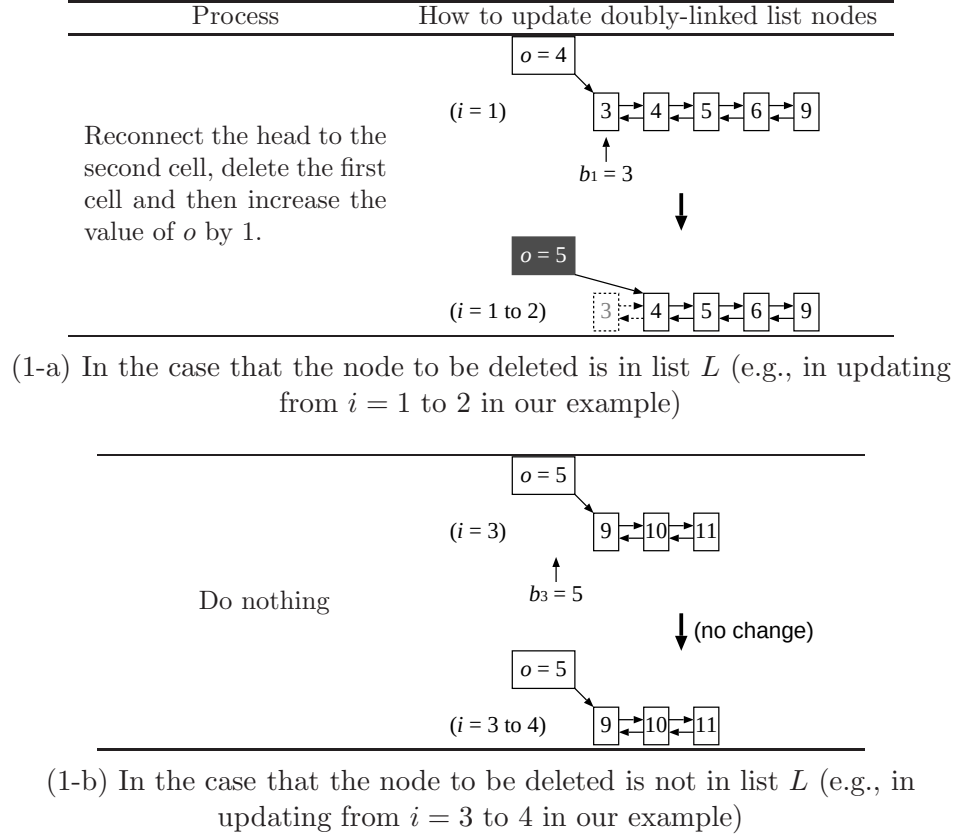


Figure 17: Case study of **DEL-FIRST** step in Algorithm *ListBaseline-Search* (corresponding to step (1) in Fig. 15). Case (1-a): When the node b_i is included in the 1st cell of L , then reconnect the pointer to the next and increase the value of o by one. Case (1-b): When the node b_i is not included in L , nothing is necessary to do.

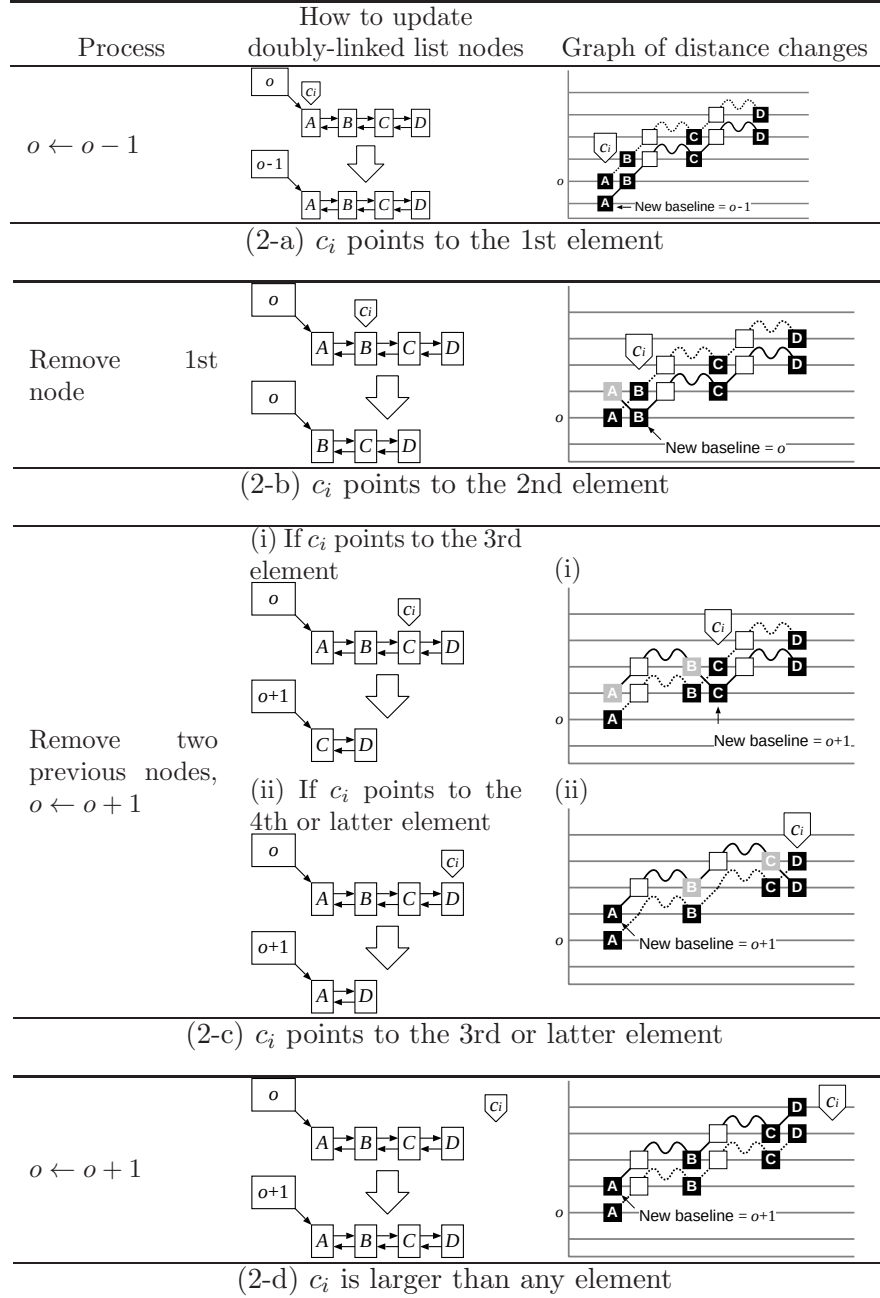
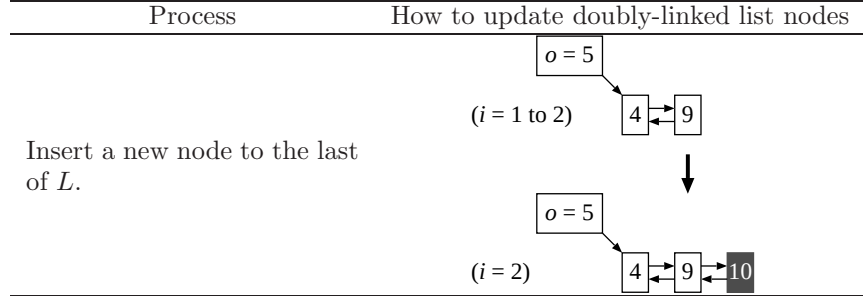
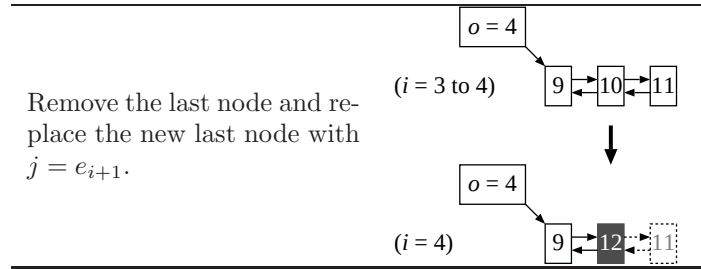


Figure 18: Case study of **DEL-MIDDLE** in Algorithm *ListBaselineSearch* according to the position of change point c_i (corresponding to step (2) in Fig. 15). From top to bottom: (2-a) in case when c_i points to the 1st element, (2-b) the 2nd, (2-c) the 3rd or latter, (2-d) greater than the last.



(3-a) In the case that addition of a new node increases the distance by one (e.g., in updating from $i = 1$ to 2 in Table 4).



(3-b) In the case that addition of a new node decreases the distance by one (e.g., in updating from $i = 3$ to 4 in Table 4).

Figure 19: Case study of **INS/DEL-LAST** step in Algorithm *ListBaselineSearch* (corresponding to step (3) in Fig. 15). Case (3-a): when the new element at e_{i+1} to be added increases the distance by one, then add the new node to the last of list L . Case (3-b): when the new element at e_{i+1} to be added decreases the distance by one, then delete the last two elements of L and insert it to the last.

- (4) j^r is the largest one in $[b_i, e_i]$ among sharing the same distance $D[j^r] - D[j^*] = r - 1$.

An example is shown in Fig. 15.

The algorithm is shown in Fig. 16. In the algorithm, the fundamental tree operations are **DEL-FIRST**, **DEL-MIDDLE** and **INS/DEL-LAST** as shown in Fig. 15. Here **DEL-FIRST** is the operation of deleting the first element from L of LsAT, **DEL-MIDDLE** is the operation of deleting some elements before c_i , and **DEL/INS-LAST** is the operation of either deleting or inserting an element at the end of L . These three operations are furthermore divided into two or three cases (Figs. 17, 18 and 19). Detailed operations for the tree are shown in Appendix B.

5.6.2 Complexity Analysis

Let us analyze the necessary cost for each operation of the algorithm. We have the following theorem.

Theorem 5.2. *Let α be the probability that the event $c_i \in \text{scope}(i + 1)$ happens. Then *ListBaselineSearch* runs in $O(\alpha|t| \log k + |t| + |p|)$ time.*

The complexity is better than that in *ArrayBaselineSearch* ($O(\alpha|t|k + |t| + |p|)$ in Theorem 5.1). From Theorem 5.2, it is obvious that the algorithm works in $O(|t| \log k + |p|)$ time in the worst case.

Proof of Theorem 5.2. Assume that, if $c_i \notin \text{scope}(i + 1)$ then the cost of updating LsAT is $O(\log k)$ in probability β ($\beta \leq \alpha$) and $O(1)$ in probability $1 - \beta$, otherwise the cost is always $O(\log k)$. Then, separating the time complexity into one for $c_i \in \text{scope}(i + 1)$ and the other for $c_i \notin \text{scope}(i + 1)$, we can evaluate it as

$$\begin{aligned}
& \alpha|t| \cdot O(\log k) + (1 - \alpha)|t| \cdot (\beta O(\log k) + (1 - \beta)O(1)) \\
= & \alpha|t| \cdot O(\log k) + (1 - \alpha)\beta|t| \cdot O(\log k) + (1 - \alpha)(1 - \beta)|t| \cdot O(1) \\
\leq & \alpha|t| \cdot O(\log k) + (1 - \alpha)\alpha|t| \cdot O(\log k) + |t| \cdot O(1) \\
\leq & \alpha|t| \cdot O(\log k) + \alpha|t| \cdot O(\log k) + |t| \cdot O(1) \\
= & O(\alpha|t| \log k + |t|).
\end{aligned}$$

Adding the complexity $O(|t|)$ for calculating change points c_i and $O(|p|)$ for constructing a suffix tree of p , we have the theorem.

Now let us confirm that the assumption holds by evaluating the complexities of operations: **DEL-FIRST**, **DEL-MIDDLE**, **DEL/INS-LAST** and searching for the position of c_i in L . We will prove that

- searching for the position of c_i in L and **DEL-MIDDLE**, both of which need to be carried out only if $c_i \in \text{scope}(i + 1)$, can be done in $O(\log k)$ time, and
- **DEL-FIRST** and **DEL/INS-LAST**, both of which are needed even if $c_i \notin \text{scope}(i + 1)$, can be done in $O(\log k)$ time in probability β and $O(1)$ in probability $1 - \beta$.

First we calculate the complexities of searching for the position of c_i in L and **DEL-MIDDLE**. Searching for the position of c_i can be done in $O(\log k)$ since we conduct a search in T of at most $2k + 1$ nodes and conduct a binary search on a chunk of A of at most $2k + 1$ cells. **DEL-MIDDLE** can also be done in $O(\log k)$ since an addition of a node or a removal of at most two nodes is sufficient to arrange T , as described in detail in Appendix B.

Next we analyze the complexities of **DEL-FIRST** and **DEL/INS-LAST**. We prove that **DEL-FIRST** can be done in $O(\log k)$ time in probability β ($\beta \leq \alpha$) and $O(1)$ in probability $1 - \beta$. In **DEL-FIRST**, the cost is clearly $O(1)$ if we do not need to remove the first element of list L . Even if this is not the case, a constant time is sufficient if the removal of the first element of L does not propagate to the removal of the leftmost node of T . (In this case the node contains two elements and therefore we have only to rewrite *IndexRange* and *ValueRange* of the node.) The exceptional case can arise only if the leftmost tree node holds only one array cell, that is, if the second cell of A is empty. In this case we need $O(\log k)$ to remove the leftmost node and rotate T (see Fig. 20). Similarly, in **INS/DEL-LAST**, if it requires addition of a new element to the last of L and does not cause a rearrangement of T , then the cost is $O(1)$. Otherwise (removal of the last element of L) we can conduct the removal in $O(1)$ or $O(\log k)$ time in a way similar to that in **DEL-FIRST** (detailed in Appendix B).

By β we denote the probability of events of reconstructing the tree. Such an event occurs only when at least one hole exists in array A . A hole is generated only when a change point ever occupied somewhere in A , that

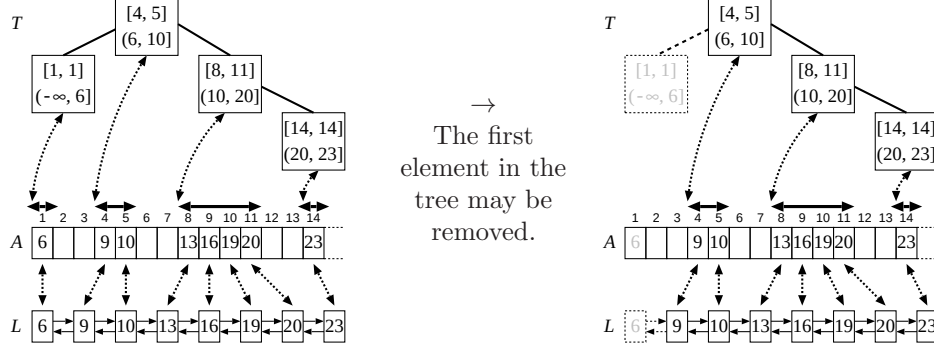


Figure 20: An example of removing the leftmost node of T by **DEL-FIRST**

is, when $c_i \in \text{scope}(i + 1)$ held for some past i . Therefore, we can conclude that $\beta|t| \leq \alpha|t|$, deriving $\beta \leq \alpha$.

With all these analyses, we have proved the theorem. $\square$

Under our assumption $k = \Theta(|p|)$, this complexity becomes $O(\alpha|t| \log |p| + |p|)$. In a way similar to that for *ArrayBaselineSearch* (Corollaries 5.1 and 5.2), we obtain the following two corollaries.

Corollary 5.3. *ListBaselineSearch runs in $O(|t| + |p|)$ time if $\alpha = O(1/|t|)$ or $\alpha = O(1/\log |p|)$ provided that $k = \Theta(|p|)$.*

Corollary 5.4. *Let t consist of characters chosen randomly according to some distribution over Σ . Then ArrayBaselineSearch runs in $O(|t| + |p|)$ time on average if $q = \omega(\log |p|)$ for $k = \Theta(|p|)$, more precisely, if $q \geq \log_{1/r_{\max}}(|p| \log |p|)$, where $r_{\max}$ is the maximum probability of occurrence of characters over Σ .*

The average-case complexity of *ListBaselineSearch* for random t (Corollary 5.4) is the same as that of *ArrayBaselineSearch* (Corollary 5.2). Note that, in addition to the worst-case time complexity, *ListBaselineSearch* is also advantageous to *ArrayBaselineSearch* in q : the required condition is relaxed from $q \geq 2 \log_{1/r_{\max}} |p|$ to $q \geq \log_{1/r_{\max}}(|p| \log |p|)$. Thus α is likely to be smaller with the same q than that in *ArrayBaselineSearch*.

5.7 Experiment

Let us verify in an experiment the theoretical computational complexities of the proposed two algorithms *ArrayBaselineSearch* and *ListBaselineSearch*, and Ukkonen's two original algorithms *ArraySearch* and *TreeSearch*.

5.7.1 Settings

The experiment was carried out in the following way:

- Compose a text string t of length 100,000 by randomly choosing the characters from Σ ($|\Sigma| = 20$) with equal probability. For each of $|p| \in \{10, 20, 30, \dots, 500\}$, generate 100 patterns p by taking substrings of length $|p|$ from t starting from random positions.
- Enumerate every substring similar to p in t with $d_q(t[i..j], p) \leq k$, where $k = |p|$ and $q = 5$. Note that $k = \Theta(|p|)$ is the original assumption of the complexity analysis.
- Assuming that the suffix tree of t is built before searching, only the times consumed both for construction of the suffix tree of p and for similar substring searching are measured. The resulted time is averaged for 100 trials in each $|p|$.

In the implementation, the red-black tree [46] is used for realizing the search trees in *ListBaselineSearch* and *TreeSearch*. The programs were written in C++, compiled by GCC 4.6.3 and run on a computer with CPU of AMD Opteron 1352 (clock rate: 2.1 GHz), 4GB RAM and operating system of Ubuntu 12.04 64bit.

5.7.2 Result

The result is shown in Fig. 21. Only $|p|$ is changed. As seen in Fig. 21(a), the proposed algorithms *ArrayBaselineSearch* and *ListBaselineSearch* improved Ukkonen's corresponding algorithms *ArraySearch* and *TreeSearch*, respectively.

Let us discuss the complexity of those four algorithms in the setting of this experiment. Let us examine the complexity in $|p|$. Noticing $|t|$ ($= 100,000$) $\gg |p|$ (≤ 500) and $|t|$ is constant, we may omit $|t|$ and $|p|$ from

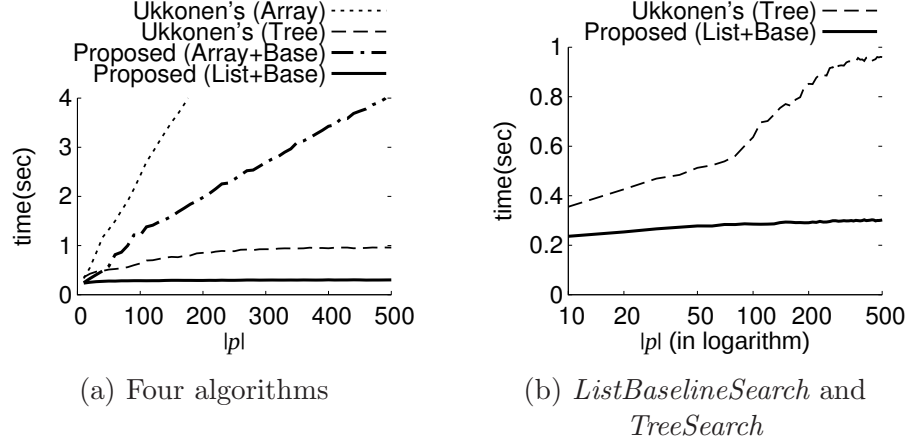


Figure 21: Computation time of similar substring searching ($|t| = 100,000$, $|\Sigma| = 20$, $q = 5$)

evaluation if they appear solely in the expression and replace k with $|p|$. For example, we use the reduction $O(|t| + |p|) = O(|t|) = O(1)$. Then their theoretical complexities are simplified as follows:

- Ukkonen's *ArraySearch*: $O(|t|k) = O(|t||p|)$,
- Ukkonen's *TreeSearch*: $O(|t| \log k + |p|) = O(|t| \log |p| + |p|) = O(|t| \log |p|)$,
- proposed *ArrayBaselineSearch*: $O(|t| + |p|) = O(|t|) = O(1)$, if $q \geq 2 \log_{1/r_{\max}} |p|$, and
- proposed *ListBaselineSearch*: $O(|t| + |p|) = O(|t|) = O(1)$, if $q \geq \log_{1/r_{\max}} (|p| \log |p|)$.

The conditions for the last two algorithms are come from Corollaries 5.2 and 5.4, respectively, and they limit the validity of their theoretical complexities. In this experiment, $1/r_{\max} = 20$ and $q = 5$, so that both conditions are satisfied.

From Fig. 21, we can see that the results are consistent with their theoretical complexities derived above except for *ArrayBaselineSearch*: linear, logarithmic and constant in $|p|$ for *ArraySearch*, *TreeSearch* and *ListBaselineSearch*, respectively. A possible explanation for *ArrayBaselineSearch* relies on the condition. This evaluation (the third evaluation above) holds

only when $q \geq 2 \log_{1/r_{max}} |p|$, and this condition is satisfied in this experiment because $5 \geq 2 \log_{20} 500 = 4.1$ even for the largest $|p| = 500$. On the other hand, the large-oh evaluation holds only when $|p|$ is sufficiently large. As a result, although the condition is satisfied, it is not sufficient for making sure of the complexity. While, the condition for *ListBaselineSearch* is sufficiently satisfied by $5 \geq \log_{20}(500 \log 500) = 2.7$.

6 Conclusion

6.1 Results

In this paper we discussed the q -gram distance $d_q(x, y)$ as an approximation of the edit distance $d_e(x, y)$. Especially, for real-life usefulness, we solved the following two problems.

The first problem is that it is not clear when the q -gram distance approximates the edit distance well. We compared two q -gram-distance-based and four other edit distance approximation algorithms in the distortion: a measure of function approximation accuracy. For real-life usefulness in addition to unified measure for the comparison, we derived the distortions without big-oh notation, different from existing evaluations. As a result, the approximations by the two q -gram-distance-based distances are more accurate than others for small string length about $n \leq 300$, in both theoretically and experimentally. In addition, experimental result showed the possibility that they may be also advantageous if the alphabet size is large.

The second problem is that the similar substring searching by the q -gram distance may be faster: the best existing algorithm needs $O(|t| \log k + |p|) = O(|t| \log |p| + |p|)$, larger than the ideal $O(|t| + |p|)$, where we search substrings of the text string t whose q -gram distance to the pattern string p is k or less. The proposed algorithm achieved $O(|t| + |p|)$ time on average with the worst-case unchanged, in case t is randomly chosen and $q = \omega(\log |p|)$. In addition, experimental results assured the time complexity and faster computation of the proposed algorithm than the existing one by about three times.

6.2 Discussions

For the first problem, a new approach is to compare edit distance approximation algorithms with a unified measure: the distortion without big-oh notation. In addition, subsequently, we can know that which approximation algorithm should be chosen according to string lengths n , which cannot be proven with big-oh notation. However, further discussions are needed on the fact the distortions are quite different between the theoretical and the experimental results. Because the difference causes different n for the choice above, the difference should be reduced by, for example, theoretically assuring experimental results under the condition that strings are randomly

chosen.

For the second problem, a large improvement is that the substring searching by the q -gram distance is proved to be done in the ideal $O(|t| + |p|)$ time in the average case of not so restricted condition $q = \omega(\log |p|)$. In addition, the $O(|t| + |p|)$ time is confirmed experimentally as well as faster computation than the existing algorithm. An important future task would be the development of an index that reduces the searching time to $o(|t|)$, that is, less than proportional to the text length. It is desirable for real-life searching problem because the computation time does not increase much even if the amount of string database is increased. Moreover, from a theoretical point of view, proving whether $O(|t| + |p|)$ time is achieved or not in the worst case (including indexing if exists).

In total, the q -gram distance can be a good replacement of the edit distance in comparison with other approximation algorithms: If the string length is small, it yields high approximation accuracy. If the string length is large, although the accuracy becomes worse, it has the advantage in the computation time.

6.3 Future Directions

As stated in Section 1, the importance of the fast similar substring searching is expected to remain growing as the amount of available data growing. As a result, in the future, it would be the best if the fast q -gram distance directly replaces the edit distance. Thus it might be useful to apply existing techniques used for the edit distance to the q -gram distance. Here are two possible improvements:

One is to adopt a distance-based index, as shown in Section 3.4. The q -gram distance is advantageous compared to the edit distance because it can use the index structure for Cartesian space (Section 3.4), which cannot be used for the edit distance. Although it does not directly solve the indexing issue for the substring searching in the previous section, the problem “finding similar strings from a set of strings to a query”, which is another important setting than the substring searching, is expected to be solved fast.

The other is to generalize the q -gram distance to reflect the character-wise “weights”. For the edit distance, the increase of the distance by insertion, deletion and substitution are often weighted (i.e. made different)

by the changed character from the nature of data [3, 5]. It would be useful to set weights for q -grams to reflect the weights for characters to fit the q -gram distance to the situation, but the setting would be carefully examined. Because the generalization has not been considered for other types of edit distance approximation algorithms (Sections 3.2 and 4) either, it is valuable even if an approximation other than the q -gram distance reflects the generalization.

Appendix

A Detailed Distortion Analysis of Six Edit Distance Approximation Algorithms

Note: The content of the section is based on the author's works [9, 10].

In this appendix, detailed calculations for the non-asymptotic distortions in Section 4.2 are shown.

A.1 Distortion Calculations from Inequalities

A.1.1 Algorithm *Bar-Yossef-2004*

The upper and the lower bounds of the distance of algorithm *Bar-Yossef-2004* [8] are given by

$$\begin{cases} d_e \leq k \Rightarrow \tilde{d}_e \leq 4kq, \\ d_e \geq 13(kn)^{\frac{2}{3}} \Rightarrow \tilde{d}_e \geq 8kq. \end{cases} \quad (\text{with } q = n^{2/3}/(2k^{1/3}))$$

As a result we obtain

$$\frac{4}{13}d_e \leq \tilde{d}_e \leq 2(d_en)^{2/3}.$$

As shown in Section 4.2.2, since $u(d_e)/d_e = 2(n^2/d_e)^{1/3}$ and $l(d_e)/d_e = 4/13$ are monotonically decreasing and increasing, respectively, the distortion for $d_e \geq \theta$ is $K_\theta = u(\theta)/l(\theta) = 2(\theta n)^{2/3}/(\frac{4}{13}\theta) = (13n^{2/3})/(2\theta^{1/3})$.

A.1.2 Algorithm *Sokolov-2007*

The upper and the lower bounds of the distance of algorithm *Sokolov-2007* [7] are given by

$$\begin{aligned} d_e(x, y) \leq k &\Rightarrow \tilde{d}_e(x, y) \leq (2k(n+2))/n, \\ d_e(x, y) > k &\Rightarrow \tilde{d}_e(x, y) \geq 2(k-4)/n. \end{aligned} \quad (2)$$

Note that the distortion should be treated as $+\infty$ if $\theta \leq 5$ since $\tilde{d}_e(x, y)$ can be zero if $d_e(x, y)$ is less than 5, that is, k is less than 4, from (2).

Otherwise we obtain

$$2(d_e - 5)/n \leq \tilde{d}_e \leq (2d_e(n + 2))/n.$$

As shown in Section 4.2.2, since $u(d_e)/d_e = (2d_e(n+2))/(nd_e)$ and $l(d_e)/d_e = 2(d_e - 5)/(nd_e)$ are monotonically decreasing and increasing, respectively, the distortion for $d_e \geq \theta$ is $K_\theta = u(\theta)/l(\theta) = [(2\theta(n+2))/n]/[2(\theta - 5)/n] = (\theta(n+2))/(\theta - 5)$.

A.2 Distortion Refinements without Big-oh Notations

Let $\log_b^* x$, called the *iterated logarithm* [46], be the minimum $i \geq 0$ such that $\underbrace{\log_b(\log_b(\dots \log_b x))}_{i \text{ 'log's'}} \leq 1$. If $x \leq 1$ then $\log_b^* x \stackrel{\text{def}}{=} 0$. The value $\log_b^* x$ grows very slowly compared to the growth of x , e.g. the value is $\lg^* x = 3$ if $x \in (4, 16]$ and $\lg^* x = 4$ if $x \in (16, 65536]$.

A.2.1 Algorithm *Batu-2006*

In Batu's algorithm [18], we first divide a string x into blocks of length c to $2c - 1$ and compute the edit distance block-wise (i.e. treating a block as a character). As a result, the computational cost becomes $O((n/c)^2)$ after one division. The algorithm has two parameters $c \geq 2$, $j \geq 1$.⁶ j describes the number of the *alphabet reductions* (a string conversion process that only determines the boundaries of blocks). Note that we need to increase c in accord with n by $c = \omega(1)$ to assure $o(n^2)$ -time computation. The authors of the paper take $c = (\lg \lg n) / \lg \lg \lg n$ (the end of Section 5 of [18]). In Section 4.2 we took $c = \max\{\lg \lg c / (\lg \lg \lg c), 2\}$ instead. In Section 4.3 we fixed $c = 2$ for the theoretical distortion since we took only $c = 2, 4$ for the experiment.

⁶There is another parameter ℓ , but we fixed $\ell = 1$ since it is enough for the single use of the distance ([18], pp. 799).

The distortion K is given by

$$K = (2c - 1) \cdot O((3c^2 \log c)^c / c + \log^* kc) \quad (3)$$

(Theorem 4.1 in [18], pp. 797)

$$= (2c - 1) \cdot [4c(\log^* kc + O(1)) + O((3c^2 \log c)^c)] / c$$

(Lemma 4.5 in [18], pp. 797)

$$= (2c - 1) \cdot [4cj + O((3c^2 \log c)^c)] / c. \quad (4)$$

(Lemma 4.5 in [18], pp. 797)

where $k = \lceil \lg |\Sigma| \rceil$ is the number of bits to describe a character. The remained big-oh notation $O((3c^2 \log c)^c)$ is evaluated as follows: $O((3c^2 \log c)^c)$ is obtained from 2^{k_j} where $k_i = (c - 1) \cdot (\lceil \lg((2c - 3)k_{i-1}) \rceil + 2)$, $k_0 = k$ (pp. 796 in [18]).

The case of $j = 1$

If $j = 1$, used in Section 4.3, then $k_1 = (c - 1) \cdot (\lceil \lg((2c - 3)k_{i-1}) \rceil + 2) \leq (c - 1) \cdot (\lg((2c - 3)k) + 3)$ and thus the distortion becomes

$$K \leq (2c - 1) \cdot [4c + \{8(2c - 3)k\}^{c-1}] / c. \quad (5)$$

The case when j is large enough

Then we consider the case when j is large enough for the small distortion. In this case k_j becomes the fixed point of $k_i = (c - 1) \cdot (\lceil \lg((2c - 3)k_{i-1}) \rceil + 2)$. We can easily confirm that $k_j \leq 4(c - 1)^2$ since it is larger than $(c - 1) \cdot (\lceil \lg((2c - 3)k) \rceil + 2)$ for any $c \geq 2$.⁷ In addition, j is large enough with $\lg((2c - 3)k) + 1$ if $k \geq k_j$ since the number of binary digits of k_i in the recurrence is reduced by at least one except for the final recurrence. As a result, from the expression (4), an upper bound of the distortion becomes

$$\begin{aligned} K &= (2c - 1) \cdot [4cj + O((3c^2 \log c)^c)] / c \\ &\leq 4(2c - 1) \left(\lg((2c - 3)k) + 1 + \frac{(c - 1)^2}{c} \right). \end{aligned}$$

⁷We found an upper bound $\hat{k}_j = 4(c - 1)^2$ as follows: since k is asymptotically larger than $(c - 1) \cdot (\lceil \lg((2c - 3)k) \rceil + 2)$ in k , $\hat{k}_j$ must satisfy $\hat{k}_j \geq (c - 1) \cdot (\lceil \lg((2c - 3)\hat{k}_j) \rceil + 2)$. As a result, $k_j = \omega(c)$ is required. Thus we first take $k = \gamma(c - 1)^2$ and then supplied the constant γ to satisfy the inequality.

A.2.2 Algorithm *Charikar-2006*

The distortion of Charikar's distance [20] is evaluated as $O(\log n)$ for Ulam metric. First we show its value without big-oh notation. The approximation function $\|f(P) - f(Q)\|$, where P and Q are strings satisfying the Ulam condition, is evaluated as follows in [20]:

$$\begin{aligned} \|f(P) - f(Q)\| &\leq 3(1 + \ln n) \leq 3(1 + \ln n) \frac{d_e(P, Q)}{\theta} \\ &\quad (\text{if } P \neq Q; \text{ in Lemma 2.2, pp.211 in [20]}) \\ \|f(P) - f(Q)\| &\geq d_e(P, Q)/8 \\ &\quad (\text{in Lemma 2.3, pp.212 in [20]}) \end{aligned}$$

Thus we get $d_e(P, Q)/8 \leq \|f(P) - f(Q)\| \leq 3(1 + \ln n) \frac{d_e(P, Q)}{\max\{1, \theta\}}$, where θ is replaced with $\max\{1, \theta\}$ since the expression above does not consider the case $d_e(P, Q) = 0$. This concludes the distortion of $\|f(P) - f(Q)\|$ for the Ulam metric is $\frac{24(1 + \ln n)}{\max\{1, \theta\}}$.

In addition, in the manner in Section 4.3.1, the distortion for any strings is $\frac{24(1 + \ln n)}{\max\{1, \theta\}} \cdot 2n = \frac{48n(1 + \ln n)}{\max\{1, \theta\}}$ since t is at most n .

A.2.3 Algorithm *Andoni-2009*

The distortion of the distance in algorithm *Andoni-2009* [21] is concluded as $O(1)$ for the Ulam metric. We have removed the big-oh notation as follows: The approximation function $d_{\text{NEG}, \infty, 1}(\phi(P), \phi(Q))$, where P and Q are strings satisfying the Ulam condition, is evaluated as follows in [21]:

$$\begin{aligned} d_{\text{NEG}, \infty, 1}(\phi(P), \phi(Q)) &\geq \underline{d_e}(P, Q)/50 \\ &\quad (\text{Proof of Theorem 1.1, pp.870}) \\ d_{\text{NEG}, \infty, 1}(\phi(P), \phi(Q)) &\leq 17\underline{d_e}(P, Q) \\ &\quad (\text{Proof of Theorem 1.1, pp.871}) \\ \underline{d_e}(P, Q) \leq d_e(P, Q) &\leq 2\underline{d_e}(P, Q) \\ &\quad (\text{Section 1.5, pp.868}) \end{aligned}$$

As a result, the distortion for Ulam metric is calculated as $50 \cdot 17 \cdot 2 = 1700$. In addition, in the manner in Section 4.3.1, the distortion for any

strings is $1700 \cdot 2n = 3400n$ since t is at most n .

B Detailed Procedures of DEL-FIRST, DEL-MIDDLE and INS/DEL-LAST

Note: The content of the section is based on the author's work [11].

The appendix explains the detailed procedures of **DEL-FIRST**, **DEL-MIDDLE** and **INS/DEL-LAST** in the algorithm *ListBaselineSearch* (Section 5.6) together with their time complexities.

B.1 Operation DEL-FIRST

In operation **DEL-FIRST**, as shown in Fig. 20 (Section 5.6.2), if the first element of L is removed, then we arrange the leftmost tree node: remove it from T if it holds only one list element or shrink the ranges *IndexRange* and *ValueRange* of the node otherwise. As shown in Section 5.6.2, node removals occur in probability at most β , where β is the probability of tree node addition in **DEL-FIRST** and thus $\beta \leq \alpha$. Therefore, in total, the time complexity for **DEL-FIRST** is $\beta|t|O(\log k) + (1 - \beta)|t|O(1) = O(\alpha|t| \log k + |t|)$.

B.2 Operation DEL-MIDDLE

In operation **DEL-MIDDLE**, we remove at most two successive elements from L (Fig. 18 in Section 5.6.1). First we just erase the corresponding cells in A . Then we arrange the nodes of T (Fig. 22). In this case, the number of array chunks (i.e., the number of tree nodes) can increase by one or decrease by one or two. Thus the removal can be done in $O(\log k)$ time since we have only to insert or remove at most two nodes of T . The remained operations of removals from both L and A can be clearly done in $O(1)$ time. Since **DEL-MIDDLE** is needed only if $c_i \in \text{scope}(i + 1)$, the time complexity for **DEL-MIDDLE** in total is $\alpha|t| \cdot O(\log k) = O(\alpha|t| \log k)$.

B.3 Operation INS/DEL-LAST

Operation **INS/DEL-LAST** is carried out as one of the following two operations: (a) adding a new element at the end of L and (b) replacing the two last elements of L with a new element. Moreover, the latter operation is divided into two cases: (b-1) the rightmost node of T is removed and

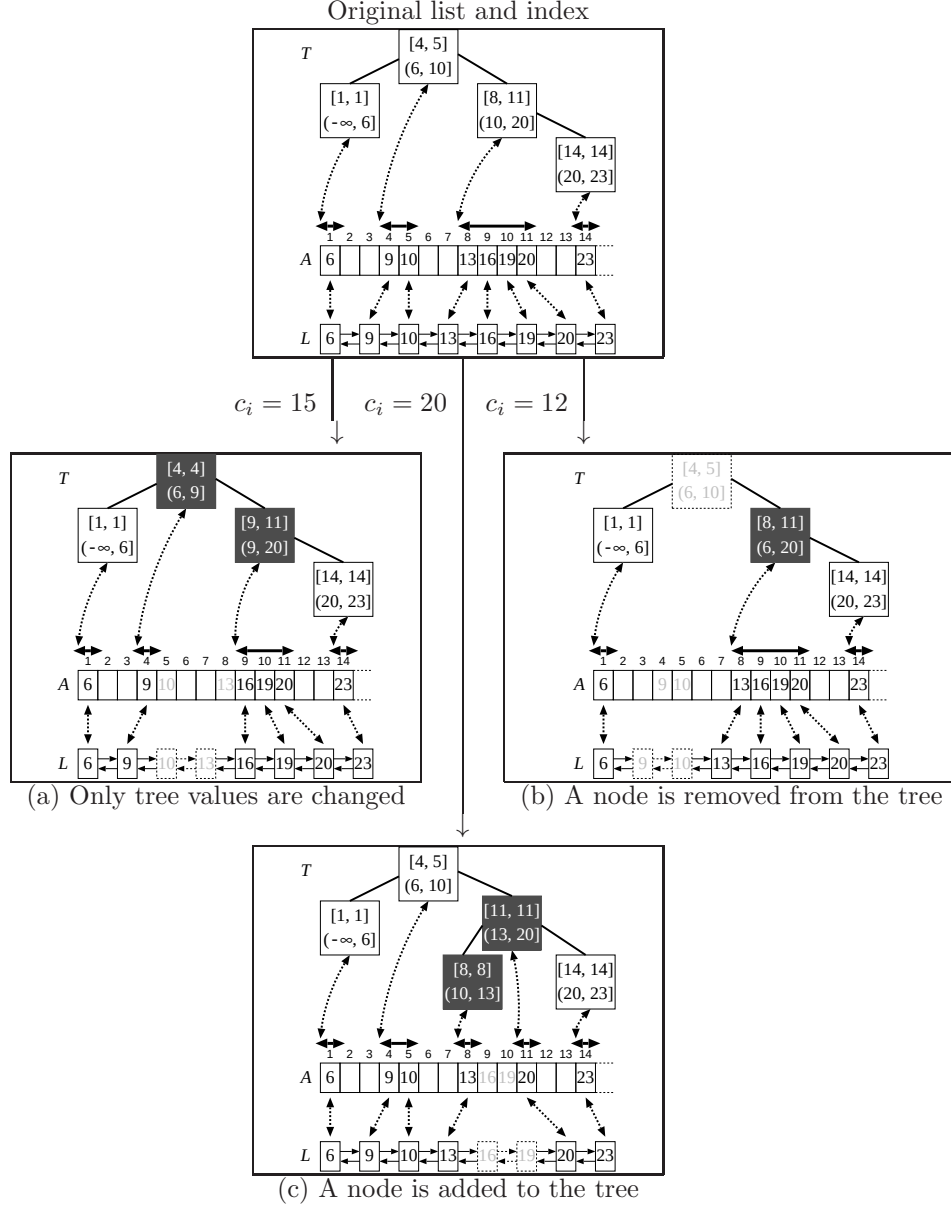


Figure 22: Case study of removing two elements in LsAT by **DEL-MIDDLE**. There are three cases of (a) only node values are changed in T , (b) one or two nodes are removed from T , and (c) a node is added to T . These processes can be done in $O(\log k)$ time in T and $O(1)$ time in A and L .

(b-2) the node is not removed. It is clear that these operations on both L and A can be done in $O(1)$ time in any of these cases. Thus it is sufficient to analyze the computational time of operations on T . In case (a), T can be arranged in $O(1)$ time since we do not have to insert or remove any tree node but arranging the rightmost tree node values. In case (b), T can be arranged in $O(\log k)$ time for case (b-1) and $O(1)$ for case (b-2). Since case (b-1) can occur at most the number of tree node insertions, the probability of (b-1) is bounded by β with the same reason as in **DEL-FIRST**, where β is the probability of tree node addition in **DEL-MIDDLE**. Thus **INS/DEL-LAST** can be done in $O(\log k)$ time in probability β and $O(1)$ in probability $1 - \beta$.

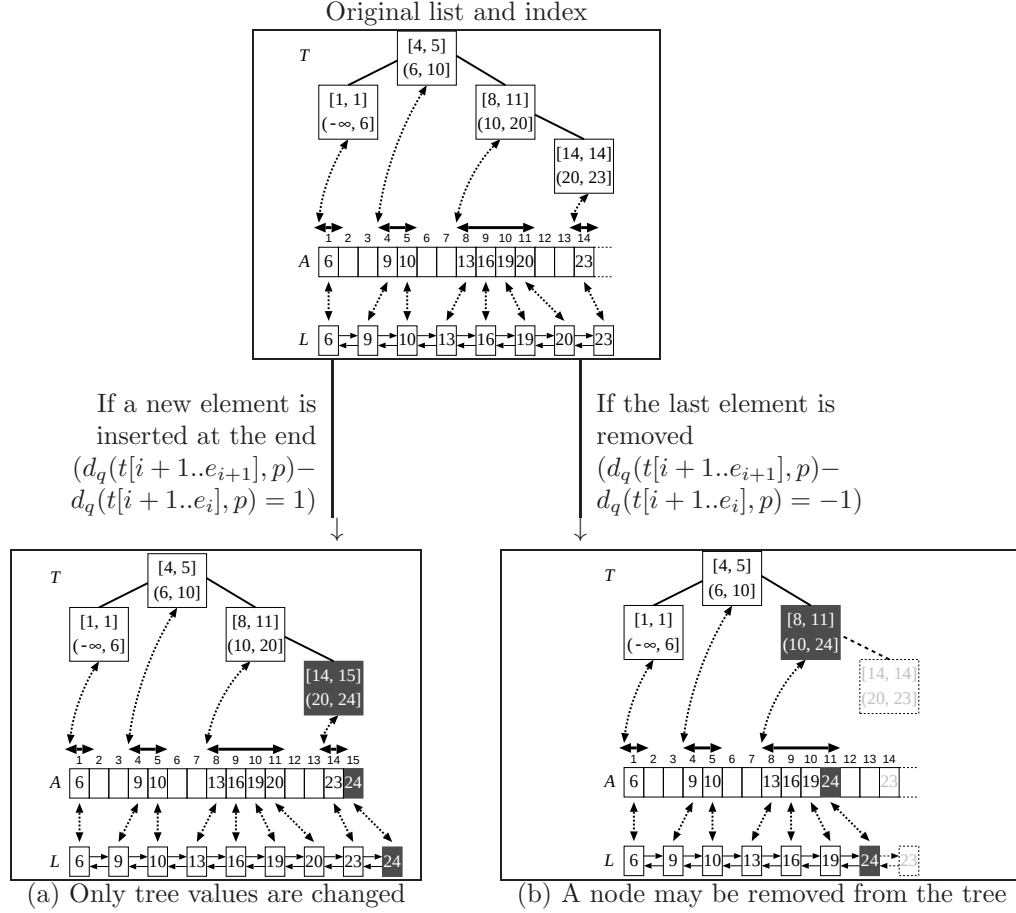


Figure 23: Case study of **INS/DEL-LAST** in LsAT. (a) If we add an element at the end, we change the values of the rightmost node in T without adding or removing any node. (b) If we remove an element at the end, the rightmost node in T is removed if it holds only one list element, otherwise we have only to change its values. The process can be done in $O(\log k)$ time in probability β (probability of tree node addition in **DEL-MIDDLE**) and $O(1)$ in probability $1 - \beta$.

Publications

Published Journal Paper

- [11] H. Hanada, M. Kudo and A. Nakamura, Average-case linear-time similar substring searching by the q-gram distance. *Theoretical Computer Science*, **530**(2014), 23–41.

Under-review Journal Paper

- [9] H. Hanada, M. Kudo and A. Nakamura, On Practical Accuracy of Edit Distance Approximation Algorithms. Submitted to *Algorithmica* (under review) in February 2014.

Conference Proceedings

- [10] H. Hanada, A. Nakamura and M. Kudo, A Practical Comparison of Edit Distance Approximation Algorithms. *Proceedings of the 2011 IEEE International Conference on Granular Computing*, 2011, 231–236.
- [12] H. Hanada, M. Kudo and A. Nakamura, Quasi-Linear-Time Substring Searching by q-gram Distance. *Proceedings of the 4th International Conference on Data Mining and Intelligent Information Technology Applications*, 2012, 540–545.

References

- [1] *BioMEMS and Biomedical Nanotechnology Volume II: Micro/Nano Technologies for Genomics and Proteomics*. Springer Science+Business Media, 2006.
- [2] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [3] G. Navarro, A guided tour to approximate string matching. *ACM Computing Surveys*, **33**-1(2001), 31–88.
- [4] V. I. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, **10**-8(1966), 707–710.
- [5] S. B. Needleman and C. D. Wunsch, A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, **48**-3(1970), 443–453.
- [6] E. Ukkonen, Approximate String-Matching with q-grams and Maximal Matches. *Theoretical Computer Science*, **92**(1992), 191–211.
- [7] A. M. Sokolov, Vector Representations for Efficient Comparison and Search for Similar Strings. *Cybernetics and Systems Analysis*, **43**-4(2007), 484–498.
- [8] Z. Bar-Yossef, T. S. Jayram, R. Krauthgamer and R. Kumar, Approximating Edit Distance Efficiently. *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science*, 2004, 550–559.
- [9] H. Hanada, M. Kudo and A. Nakamura. On Practical Accuracy of Edit Distance Approximation Algorithms. Submitted to *Algorithmica* (under review) in February 2014.
- [10] H. Hanada, A. Nakamura and M. Kudo, A Practical Comparison of Edit Distance Approximation Algorithms. *Proceedings of the 2011 IEEE International Conference on Granular Computing*, 2011, 231–236.
- [11] H. Hanada, M. Kudo and A. Nakamura, Average-case linear-time similar substring searching by the q-gram distance. *Theoretical Computer Science*, **530**(2014), 23–41.

- [12] H. Hanada, M. Kudo and A. Nakamura, Quasi-Linear-Time Substring Searching by q-gram Distance. *Proceedings of the 4th International Conference on Data Mining and Intelligent Information Technology Applications*, 2012, 540–545.
- [13] P. Weiner, Linear pattern matching algorithms. *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, 1973, 1–11.
- [14] U. Manber and G. Myers, Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, **22**-5(1993), 935–948.
- [15] D. K. Kim, J. S. Sim, H. Park and K. Park, Linear Time Construction of Suffix Arrays. *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, 2003, 186–199.
- [16] C. Leslie, E. Eskin and W. S. Noble, The Spectrum Kernel: A String Kernel for SVM Protein Classification. *Proceedings of Pacific Symposium on Biocomputing 2002*, 2002, 566–575.
- [17] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini and C. Watkins, Text Classification Using String Kernels. *Journal of Machine Learning Research*, **2**(2002), 419–444.
- [18] T. Batu, F. Ergun and C. Sahinalp, Oblivious String Embeddings and Edit Distance Approximations. *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, 2006, 792–801.
- [19] A. Andoni, R. Krauthgamer and K. Onak, Polylogarithmic Approximation for Edit Distance and the Asymmetric Query Complexity. *Proceedings of the 51st IEEE Annual Symposium on Foundations of Computer Science*, 2010, 377–386, (Full version available at <http://arxiv.org/abs/1005.4033>).
- [20] M. Charikar and R. Krauthgamer, Embedding the Ulam Metric into l_1 . *Theory of Computing*, **2**-11(2006), 207–224.
- [21] A. Andoni, P. Indyk and R. Krauthgamer, Overcoming the l_1 Non-embeddability Barrier: Algorithms for Product Metrics. *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2009, 865–874.

- [22] P. H. Sellers, The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Journal of Algorithms*, **1**-4(1980), 359–373.
- [23] E. Ukkonen, Finding Approximate Patterns in Strings. *Journal of Algorithms*, **6**-1(1985), 132–137.
- [24] G. M. Landau and U. Vishkin, Fast Parallel and Serial Approximate String Matching. *Journal of Algorithms*, **10**-2(1989), 157–169.
- [25] B. Melichar, String Matching with k Differences by Finite Automata. *Proceedings of the 13th International Conference on Pattern Recognition*, **2**(1996).
- [26] S. Wu and U. Manber, Fast Text Searching: Allowing Errors. *Communications of the ACM*, **35**-10(1992), 83–91.
- [27] W. I. Chang and E. L. Lawler, Sublinear Approximate String Matching and Biological Applications. *Algorithmica*, **12**-4(1994), 327–344.
- [28] G. Myers, A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming. *Journal of the ACM*, **46**-3(1999), 395–415.
- [29] H. Hyvärinen and G. Navarro, Faster Bit-Parallel Approximate String Matching. *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, 2002, 203–224.
- [30] A. L. Cobbs, Fast Approximate Matching using Suffix Trees. *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, 1995, 41–54.
- [31] P. Zuzula, G. Amato, V. Dohnal and M. Batko, *Similarity Search: The Metric Space Approach*. Springer US, 2006.
- [32] J. L. Bentley, Multidimensional binary search trees used for associative searching. *Communications of the ACM*, **18**-9(1975), 509–517.
- [33] S. C. Sahinalp, M. Tasan, J. Macker and Z. M. Ozsoyoglu, Distance based indexing for string proximity search. *Proceedings of the 19th International Conference on Data Engineering*, 2003, 125–136.

- [34] E. Karakoc, Z. Ozsoyoglu, S. Sahinalp, M. Tasan and X. Zhang, Novel Approaches to Biomolecular Sequence Indexing. *Bulletin of the IEEE Technical Committee on Data Engineering*, **27**-3(2004), 40–47.
- [35] P. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces. *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1993, 311–321.
- [36] J. K. Uhlmann, Satisfying General Proximity / Similarity Queries with Metric Trees. *Information Processing Letters*, **40**-4(1991), 175–179.
- [37] J. Venkateswaran, T. Kahveci, C. Jermaine and D. Lachwani, Reference-Based Indexing for Metric Spaces with Costly Distance Measures. *The VLDB Journal*, **17**-5(2008), 1231–1251.
- [38] G. Navarro and E. Chávez, A Metric Index for Approximate String Matching. *Theoretical Computer Science*, **352**(2006), 266–279.
- [39] D. J. Lipman and W. R. Pearson, Rapid and sensitive protein similarity searches. *Science*, **227**-4693(1985), 1435–1441.
- [40] S. F. Altschul, W. Gish, W. Miller, E. W. Myers and D. J. Lipman, Basic local alignment search tool. *Journal of Molecular Biology*, **215**-3(1990), 403–410.
- [41] 阿久津達也, アルゴリズム・サイエンスシリーズ 12: バイオインフォマティクスの数理とアルゴリズム. 共立出版, 2007 (Tatsuya Akutsu, Mathematics and Algorithms in Bioinformatics. Kyoritsu Shuppan, 2007, in Japanese).
- [42] J. Matoušek, *Lectures on Discrete Geometry*. Springer-Verlag, 2002.
- [43] H. Sugawara, K. Ikeo, S. Fukuchi, T. Gojobori and Y. Tatenno, DDBJ dealing with mass data produced by the second generation sequencer. *Nucleic Acids Research*, **37**(2009), Database issue D16–D18, (<http://www.ddbj.nig.ac.jp/index-e.html>).
- [44] UniProt Consortium, The Universal Protein Resource (UniProt). *Nucleic Acids Research*, **37**(2009), D169–D174, (<http://www.pir.uniprot.org/>).

- [45] D. E. Knuth, J. H. Morris Jr. and V. R. Pratt, Fast Pattern Matching in Strings. *SIAM Journal on Computing*, **6-2**(1977), 323–350.
- [46] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms* (2nd edition). The MIT Press, 2001.
- [47] G. Blom, L. Holst and D. Sandell, Problems and Snapshots from the World of Probability. In chapter 2 “Basic probability theory I” Springer-Verlag, 1994, 13–22.

Acknowledgment

I, the author, would like to express my deepest gratitude to Dr. Mineichi Kudo, laboratory professor and the chief examiner, for large amount of advice during all the progress of the research including directions of researches, thinking ways and presentation of the results. I would also like to express my great appreciation to Dr. Atsuyoshi Nakamura, laboratory associate professor, for a lot of advice especially in the theme of the research and in the content of researches such as algorithmic issues and choices of paper submissions. I am also greatly thankful to Dr. Jun Toyama, former laboratory assistant professor, for suggestive advice especially mathematical points of views on algorithmic and experimental results I derived.

In addition, I am very grateful to Dr. Hiroki Arimura, examiner, for wide-ranged comments such as related researches and paper expressions. I am also very grateful to Dr. Takuya Kida, examiner, for helpful comments such as interesting others' researches and problem specifications.

It is also a large help for me that Graduate School of Information Science and Technology, my affiliation, let me work as a research assistant and concentrate the researches.

Then I would like to show my thanks to the members in Laboratory for Pattern Recognition and Machine Learning, my affiliation for eight years and a half, for joining the discussion on my research and enjoying daily life together.

Lastly, I would like to greatly thank my family for supporting me for a long student life.