Average-case linear-time similar substring searching by the q-gram distance

Hiroyuki Hanada[a], Mineichi Kudo[a], Atsuyoshi Nakamura[a]

[a]*Graduate School of Information Science and Technology, Hokkaido University*
Kita 14, Nishi 9, Kita-ku, Sapporo, Hokkaido, 060-0814, Japan.
Tel.: +81-11-706-6854
e-mail: {hanada,mine,atsu}@main.ist.hokudai.ac.jp

Abstract

In this paper we consider the problem of similar substring searching in the *q-gram distance*. The q-gram distance $d_q(x, y)$ is a similarity measure between two strings x and y defined by the number of different q-grams between them. The distance can be used instead of the edit distance due to its lower computation cost, $O(|x| + |y|)$ vs. $O(|x||y|)$, and its good approximation for the edit distance. However, if this distance is applied to the problem of finding all similar strings, in a long text t, to a given pattern p, the total computation cost is sometimes not acceptable. Ukkonen already proposed two fast algorithms: one with an array and the other with a tree. When "similar" means k or less in d_q, their time complexities are $O(|t|k + |p|)$ and $O(|t| \log k + |p|)$, respectively. In this paper, we propose two algorithms of average-case complexity $O(|t| + |p|)$, although their worst-case complexities are still $O(|t|k + |p|)$ and $O(|t| \log k + |p|)$, respectively. The linearity of the average-case complexity is analyzed under the assumption of random sampling of t and the condition that q is larger than a threshold. The algorithms exploit the fact that similar substrings in t are often found at very close positions if the beginning positions of the substrings are close. In the second proposed algorithm, we adopted a doubly-linked list supported by an array and a search tree to search for a list element in $O(\log k)$ time. Experimental results supported their theoretical average-case complexities.

Key words: String searching, Approximate string matching, q-gram distance

1. Introduction

1.1. Overview

The *q-gram distance* $d_q(x, y)$ is a similarity measure between two strings x and y, which is defined by the number of different q-grams (all substrings of length q) between x and y [1, 2, 3] (Section 2.2). Its promising usage is an alternative distance of the *edit distance* $d_e(x, y)$ [1, 2, 4, 5], the most popular similarity measure defined by the number of character insertions, deletions and/or substitutions to make x identical to y [6, 7]. The approximation accuracy of d_q for d_e has been well studied in recent researches [2, 3, 4].

At the expense of approximation, computation of the q-gram distance is much faster than that of the edit distance: the time complexity of computing $d_q(x, y)$ is $O(|x| + |y|)$ [1], while $d_e(x, y)$ needs $O(|x||y|)$. However, this is the cost of comparing two strings only once. This complexity might become larger in the situation in which distance calculation is repeated many times. For example of bioinformatics, given genome sequences as fragments of length tens to hundreds (*short reads*), we are often asked to assemble the total sequence referring to their similar appearances in another long sequence [8, 9]. In this case, we have to do many trial-and-error experiments to find the optimal matching. In this study, we consider the problem of searching all substrings s, in a text t, close to a query pattern p in the sense of $d_q(s, p) \leq k$. Here we assume k is proportional to $|p|$ as seen in many practical problems, that is, we assume $k = \Theta(|p|)$ unless otherwise specified.

Then there remains the problem of whether this searching problem with the q-gram distance can be done in $O(|t| + |p|)$ time. For d_e, the substring searching problem can be done in $O(|t|k + |p|) = O(|t||p|)$ in the worst cases [10, 11], which is almost optimal if we do not regard the value of k as a constant [7]. Thus, substring searching can be done with the same complexity as that for the single distance computation $O(|x||y|)$ between the two strings x and y. However, for d_q, currently the best algorithm needs $O(|t| \log k + |p|) = O(|t| \log |p| + |p|)$, which is larger than $O(|x| + |y|)$ in single distance computation.

There have been two substring searching algorithms in d_q, which were presented by Ukkonen [1]. In this paper we call them *Array-Search* and *Tree-Search*. The former needs $O(|t|k + |p|)$ time and the latter needs $O(|t| \log k + |p|)$ time even in the average case[1]. However, there has been no improvement after them.

[1] Ukkonen showed only the worst-case complexity in the reference [1], however, it is easy to show that the average-case complexity is the same as the worst-case complexity.

Table 1: 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.

Method	Distance	Time complexity	
		Average	Worst
Knuth *et al.* 1977 [12], Weiner 1973 [13], etc.	Exact	$O(\|t\| + \|p\|)$	$O(\|t\| + \|p\|)$
Ukkonen 1985 [10], Landau *et al.* 1989 [11], etc.	d_e	$O(\|t\|k + \|p\|)$	$O(\|t\|k + \|p\|)$
Ukkonen 1992 [1] (*Array-Search*)	d_q	$O(\|t\|k + \|p\|)^*$	$O(\|t\|k + \|p\|)$
Ukkonen 1992 [1] (*Tree-Search*)	d_q	$O(\|t\| \log k + \|p\|)^*$	$O(\|t\| \log k + \|p\|)$
Our *Array+Base-Search*	d_q	$O(\|t\| + \|p\|)$	$O(\|t\|k + \|p\|)$
Our *List+Base-Search*	d_q	$O(\|t\| + \|p\|)$	$O(\|t\| \log k + \|p\|)$

* These complexities have not been explicitly shown in the reference [1] but easily proven.

In this paper we propose two algorithms achieving $O(|t| + |p|)$ time, independent of the value of k in average cases, under the assumption of a random sampling of t from some distribution over an alphabet and the condition that q is larger than a small threshold value. The two algorithms, called *Array+Base-Search* and *List+Base-Search*, have the same the worst case complexities as those of Ukkonen's algorithms, $O(|t|k + |p|)$ and $O(|t| \log k + |p|)$, respectively.

1.2. Related research

As the first step, we show a summary of the computation times including our proposed algorithms in Table 1. 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 [12]). A *suffix tree* [6, 13] can also be used to achieve this complexity. For the edit distance, the first algorithm achieved $O(|t||p|)$ time [14] using dynamic programming. Later, Ukkonen [10] achieved $O(|t|k + |p|)$ time on average and Landau *et al.* [11] $O(|t|k + |p|)$ time even in worst cases. According to Navarro's survey [7], the complexity $O(|t|k + |p|)$ seems to be the best unless k is independent of $|p|$. For a small (constant) k, there exist more efficient algorithms: for example, some researchers (e.g., Ukkonen [10] and Melichar [15]) proposed to construct an automaton accepting all possible similar strings $\{s \mid d_e(s, p) \leq k\}$ and solving the corresponding exact-matching problem in $O(|p|)$ time. However, they consume at least $O(|p|^k)$ space and thus they are not feasible for a large value of k. Filtering of candidate substrings by exact matching has been also shown to be effective for the edit distance, as seen in [16, 17]. However, it does not improve the worst-case time complexities.

It is noted that the methodologies developed for the substring searching with

the edit distance are not always translated to the cases with the q-gram distance. The difference between d_e and d_q is mainly caused by the fact that d_q is almost independent of the appearance order of characters while d_e greatly depends on it. A remarkable example is seen by comparing x = "aaabbb" and y = "bbbaaa". For these strings, unlike as usual[2], $d_e(x, y)$ is larger than $d_q(x, y)$ since $d_e(x, y) = 6$ and $d_q(x, y) = 2$ for $q = 2$. More extremely, $d_e(x, y)$ can be any large number while keeping $d_q(x, y) = 2$ by fixing $q = 2$ and taking $x = a^n b^n$ and $y = b^n a^n$.

It seems that the problem of finding the minimum-sum interval of an integer sequence is also similar to the similar string searching in the q-gram distance, because the former aims at finding an interval with the minimum sum and the latter aims at finding a substring with the minimum L_1 distance. However, it should be noted that, unlike in the former case, the summed value of some position/cell can change depending on whether the number of q-grams in the substring of t is larger than that in p or not. Due to this reason, we cannot exploit such an excellent algorithm for finding the minimum-sum interval as seen in [18].

2. Definitions

2.1. Notations

We denote the alphabet (the set of characters) by Σ. Let Σ^q be the set of all strings of length q and Σ^* 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 ith position, that is, $x_{(i)} \stackrel{\text{def}}{=} x[i..i+q-1]$. Similarly, we denote the q-gram ending at the jth position by $x_{<j>} \stackrel{\text{def}}{=} x[j-q+1..j]$. For example, if x = "abcdefgh" and $q = 2$, then $|x| = 8$, $x[3..7]$ = "cdefg", $x_{(2)}$ = "bc" and $x_{<5>}$ = "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,

[2] $d_q(x, y)$ takes a larger value than $d_e(x, y)$ in high probability if q is sufficiently large [3].

for x = "abcdabcd" and $q = 3$, H_x = {2/"abc", 2/"bcd", 1/"cda", 1/"dab"}, $|H_x| = 6$, $\#H_x$("abc") = 2 and $\#H_x$("aaa") = 0.

Definition 1. *[1] 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 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 [1]. The complexity can be achieved by a suffix tree of either of two strings.

Now we state the problem setting formally:

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

$$d_q(t[i..j^*], p) \le k, \text{ and}$$
$$d_q(t[i..j^*], p) \le d_q(t[i..j], p) \text{ for } \forall j \ne j^*,$$

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 = "cabaab", p = "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 $1 (\le k)$. 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 $d_q(x, y) \ge \big|\, |H_x| - |H_y| \,\big| = |\, |x| - |y| \,|$ on condition that $x \ge q$ and $y \ge q$. Therefore, it is clear that the first condition $d_q(x[i..j], p) \le k$ 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 2. $scope(i) = [b_i, e_i] \stackrel{\text{def}}{=} [i + |p| - 1 - k, i + |p| - 1 + k]$.

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

2.3. Change Point

Now we explain the concept of the change point, which was introduced in Ukkonen's algorithm [1] and is also exploited in our algorithm.

Let us consider a sequence of all substrings starting from the ith position

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

and the corresponding sequence of distances to p

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

We show an example of $\mathbf{d}_{(i)}$ in Table 2.

First we note 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 2. 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. 1. We define the change points formally as follows.

Definition 3. *[1] The* change point c_i *in the distance sequence* $\mathbf{d}_{(i)}$ *is the position* j *such that*

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

We have the following properties on the change points.

Table 2: The q-gram distance sequence $\mathbf{d}_{(i)}$ for $i = 1, 2, \ldots, 12$ for $t =$ "aaaccaaababc" and $p =$ "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 jth position and $t_{<j>}$ is the q-gram ending at the jth 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	
$i \backslash t_{<j>}$		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 : $j \in scope(i) = [b_i, e_i] = [|p|+i-k-1, |p|+i+k-1]$)
- **Bold**: $j = c_i$ (change point)

Property 1. *[1] 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 $\{c_i \mid i = 1, \ldots, |t| - q + 1\}$ can be obtained in $O(1)$ time by classifying all of the positions $\{j | q \leq j \leq |t|\}$ by q-grams $t_{<j>}$ beforehand [1].

3. Ukkonen's substring searching algorithms by the q-gram distance

In this section we give an outline of Ukkonen's two algorithms [1] called *Array-Search* and *Tree-Search* 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

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

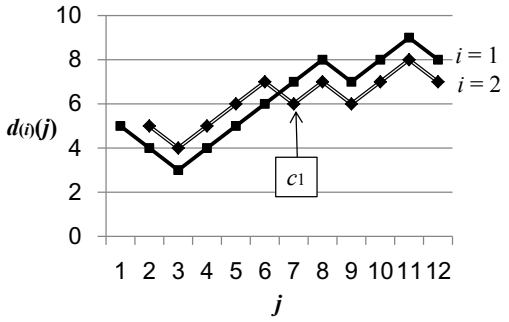


Figure 1: The q-gram distance sequences $\mathbf{d}_{(1)} = \{d_q(t[1..j], p)\}$ and $\mathbf{d}_{(2)} = \{d_q(t[2..j], p)\}$ for $t =$ "aaaccaaababc" and $p =$ "aaabbc" with $q = 2$. Before the change point $c_1 = 7$, $\mathbf{d}_{(2)}(j)$ is larger than $\mathbf{d}_{(1)}(j)$ by one; otherwise it is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

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

3.1. Ukkonen's array algorithm

First we explain *Array-Search*. In the algorithm, $\mathbf{d}_{(i)}$ 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 $j \in [b_1, e_1] = [|p| - k, |p| + k]$.

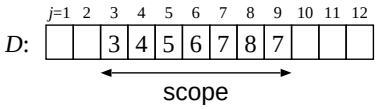
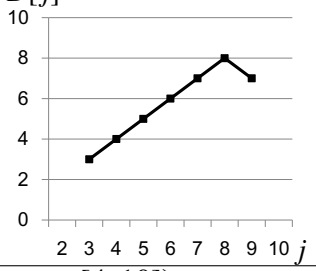
When we increment i to $i + 1$, we update the value of $D[j]$ as follows, noticing that the scope is also changed from $[b_i, e_i]$ to $[b_{i+1}, e_{i+1}] = [b_i + 1, e_i + 1]$: (1) erase the value at $D[b_i]$, (2) increase $D[j]$ by one for every j such that $b_i + 1 \leq j < c_i$ and decrease $D[j]$ by one for every j such that $c_i \leq j \leq e_i$, 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. 2.

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 [1] for details). Thus the updating procedure can be done in $O(|t|k)$ time, and the total time complexity of *Array-Search* is $O(|t|k + |p|)$, adding the time to build the suffix tree of p.

3.2. Ukkonen's tree algorithm

In *Tree-Search*, $\mathbf{d}_{(i)}$ 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 or a red-black tree [19] (Fig. 3). As in *Array-Search*, we assume i is fixed and show the update process of T_i from i to $i + 1$.

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

Array representation	Corresponding graph of distances
	

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

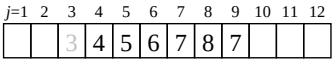
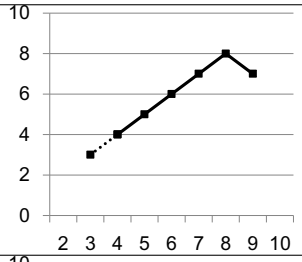
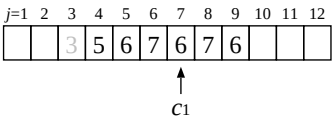
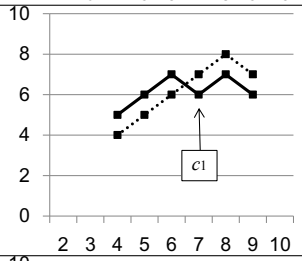
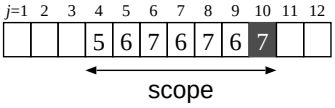
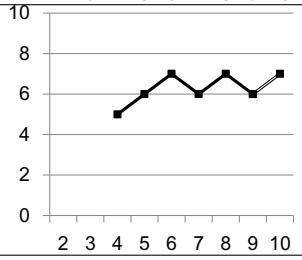
Process	Array representation	Corresponding graph of distances
(1) Erase the leftmost value.		
(2) Add 1 for $j < 7 (= c_1)$ and -1 for $j \geq 7$.		
(3) Expand the array to the right. (Array for $i = 2$ is constructed.)		

Figure 2: Update procedure of *Array-Search* in the example of Table 2. The array D stores $D[j] = \mathrm{d}_{(i)}(j) = d_q(t[i..j], p)$ 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 $D[e_{i+1}]$.

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 $j \in \mathit{scope}(i)$, 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. 3).

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. 4.

We show how to calculate all of the information at an inner node n from its left child lc and right child rc (Fig. 3). 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, *Tree-Search* needs $O(|t| \log k + |p|)$ time.

4. Proposed algorithms

4.1. Outline

We propose two improved algorithms *Array+Base-Search* and *List+Base-Search*. The common idea for the improvement 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. Then the remaining problem is how to maintain the distance information efficiently. We propose two algorithms for solving this problem.

4.2. Algorithm Array+Base-Search

In *Array+Base-Search* algorithm, we introduce a baseline variable o in addition to the distance array D adopted in Ukkonen's *Array-Search* 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 *Array+Base-Search* is described formally in Fig. 5.

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

Theorem 1. *Let α be the probability that the event $c_i \in scope(i+1)$ happens. Then Array+Base-Search 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 $\alpha \stackrel{\text{def}}{=} |\{i \mid c_i \in scope(i+1), 1 \leq i \leq |t| - q\}|/(|t| - q)$.

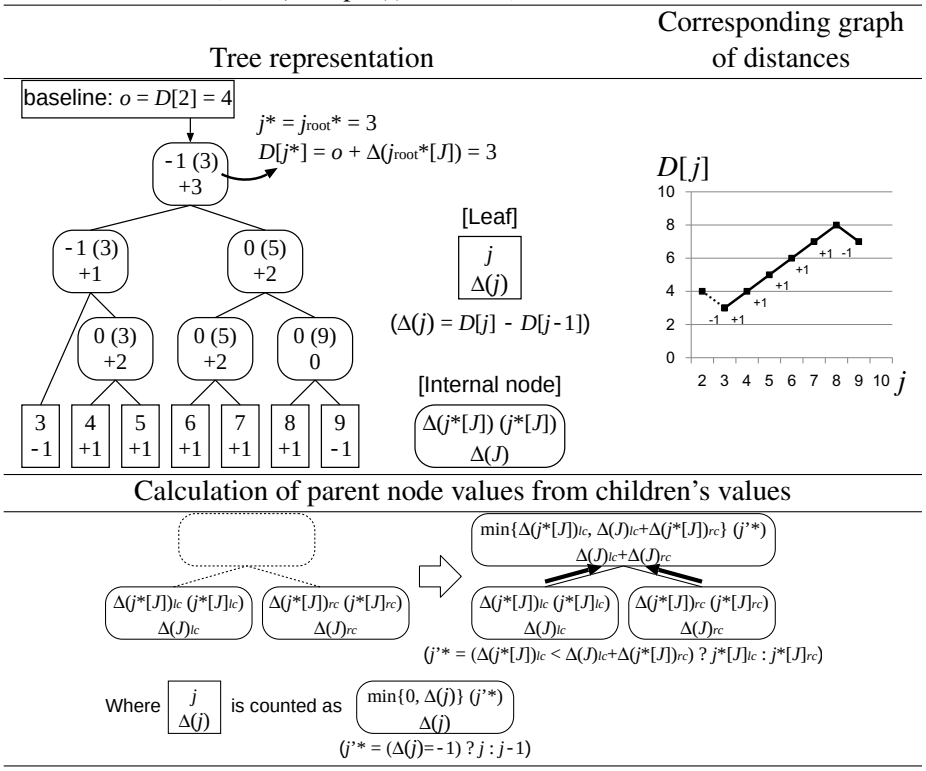


Figure 3: The binary search tree used in *Tree-Search* for the example of Table 2. This tree represents $D[j] = d_q(t[i..j], q)$ (right graph) effectively. A leaf holds a pair $(j, \Delta(j))$ $(\Delta(j) = D[j] - D[j-1])$, an internal node shows the local minimum value $\Delta(j^*[J]) = D[j^*[J]] - o$ 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: $[4, 10]$)

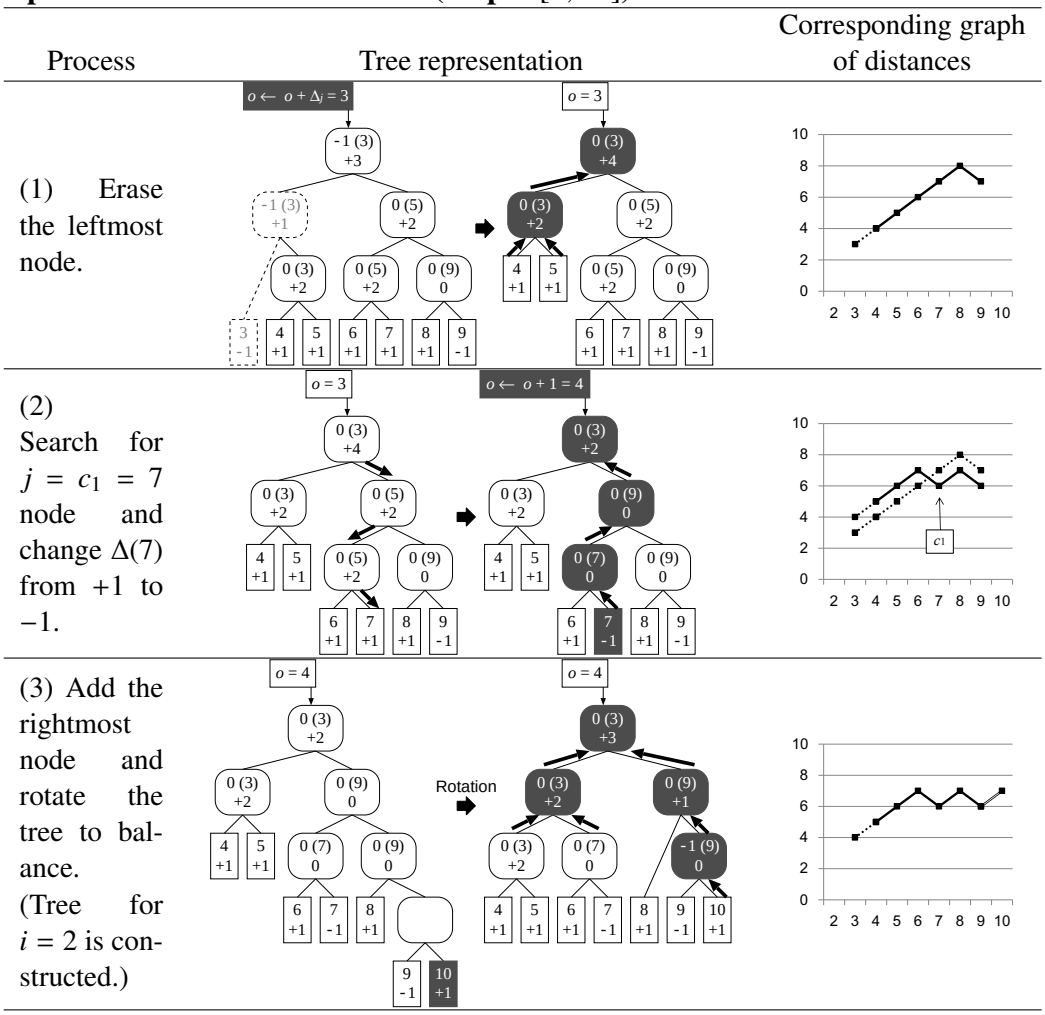


Figure 4: Update procedure of *Tree-Search* in the example of Table 2. The tree is updated from i to $i + 1$: (1) remove the leftmost node and change the value of baseline by $o \leftarrow o + \Delta(j)$ for $j = b_i$, (2) search for the change point c_i and change the value of $\Delta(j)$ from +1 to −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.

Algorithm Array+Base-Search(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 $D[j] \leftarrow D[j] + 1$; $\forall j < c_i$; $\quad D[j] \leftarrow D[j] - 1$, $\forall j \geq c_i$.

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 $D[e] \leftarrow H_{t[i..e]}(t_{<e>}) \leq H_p(t_{<e>})$? $D[e] - 1$: $D[e] + 1$.

22 **if** $D[e] \leq D[j^*]$ **then** $j^* \leftarrow e$.

23 **if** $o + D[j^*] \leq k$ **then** output $t[i..j^*]$.

Figure 5: Algorithm *Array+Base-Search*.

Proof of Theorem 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. □

The following corollary is obvious.

Corollary 1. *Array+Base-Search 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 2. *Let t consist of characters chosen randomly according to some distribution over Σ. Then Array+Base-Search 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 [20]. 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 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 $q \geq \log |p| \cdot c_o$ 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.

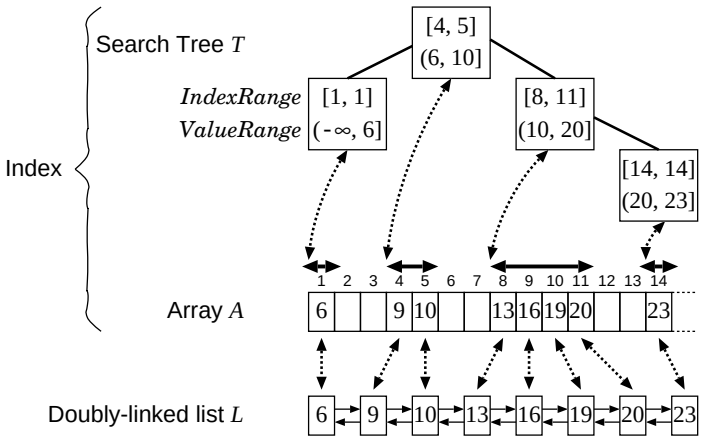


Figure 6: Data structure of LsAT: A doubly-linked list L supported by an array A and a search tree T. Each array cell stores a list element as a pointer, and each tree node stores a connected array range by both array indices and values.

4.3. Algorithm List+Base-Search

4.3.1. Outline

Next we show the second algorithm *List+Base-Search*, whose average-case time complexity is $O(|t| + |p|)$: the same as that in *Array+Base-Search*, and worst-case complexity is $O(|t| \log k + |p|)$: the same as that in Ukkonen's *Tree-Search*.

Ukkonen's *Tree-Search* algorithm is advantageous to the *Array-Search* version because the procedure for each i can be done in $O(\log k)$ that is faster than $O(k)$ in the latter. However, the tree structure requires $O(\log k)$ time for updating even if $c_i \notin scope(i + 1)$ ((1) and (3) in Fig. 4). Therefore, instead of a tree, we propose to use a data structure of a doubly-linked list supported by an array and a search tree, called *LsAT* in this paper.

We show an outline of LsAT. The list L, called the *candidate list*, efficiently holds the current j^* for i and all of the candidates of j^* for the future $i+1, i+2, \ldots$ In Fig. 6, 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. 6, there are six holes at array indices 2, 3, 6,

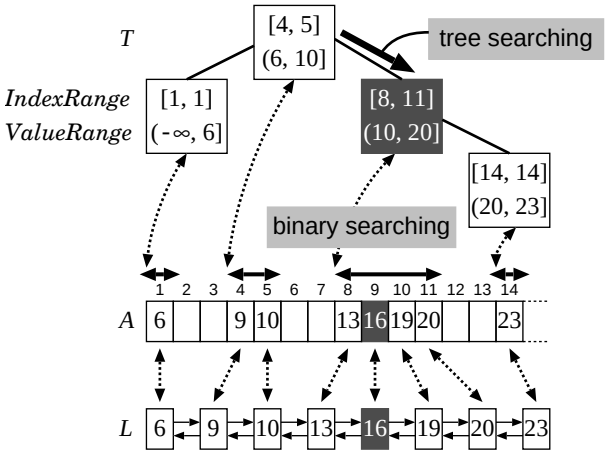


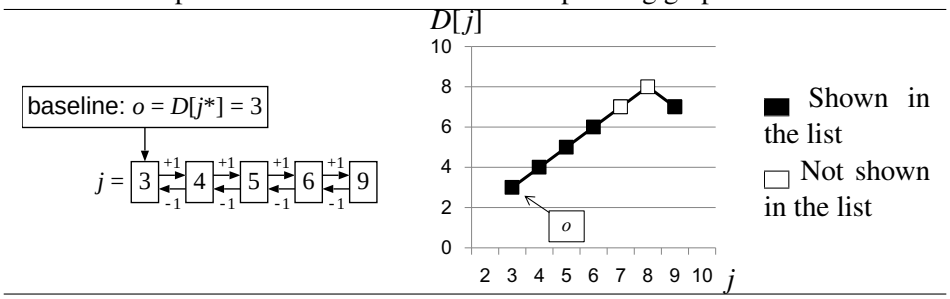
Figure 7: Searching for a change point in an LsAT (example of finding $c_i = 15$ or the next larger element). First we search T for the node including c_i by comparing *ValueRange* and then search *IndexRange* of A with binary searching. Finally we reach the corresponding element in L.

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*[3]). 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. 6, 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 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 2, we can show that the cost of necessary operations for the case of $c_i \notin scope(i + 1)$ is less than $O(\log k)$ on average.

[3] *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. 8 and 11.

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



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

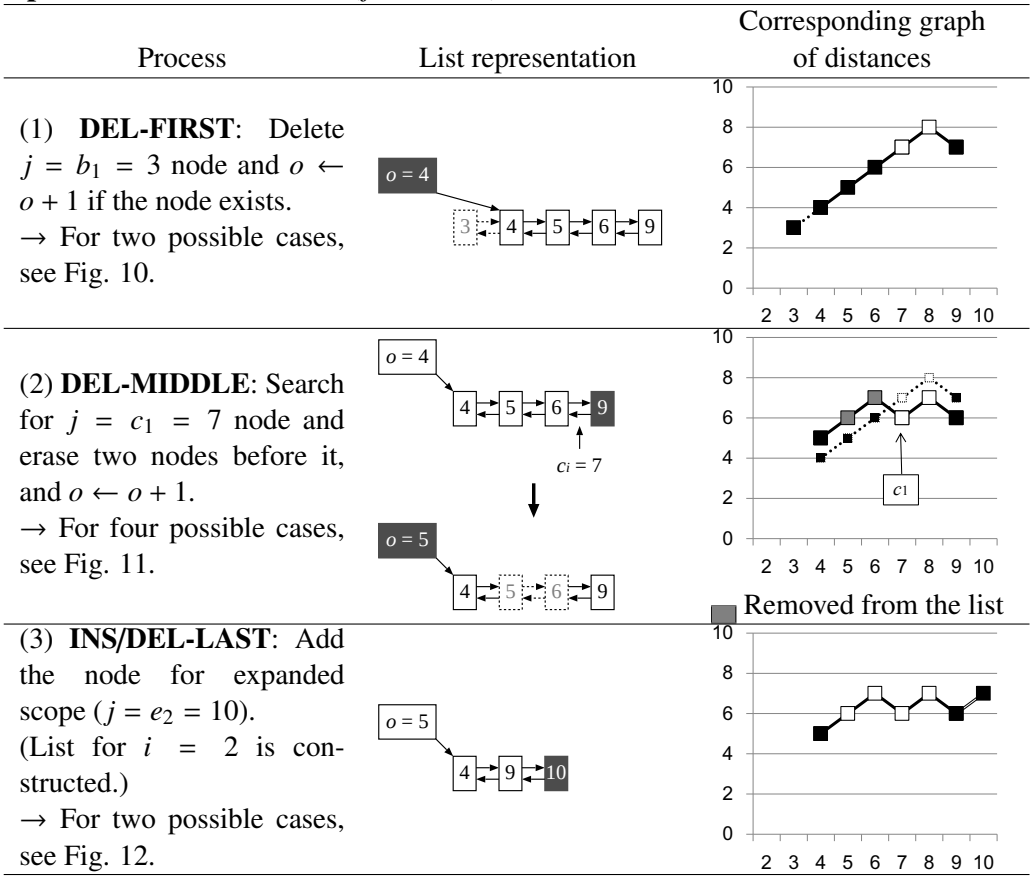
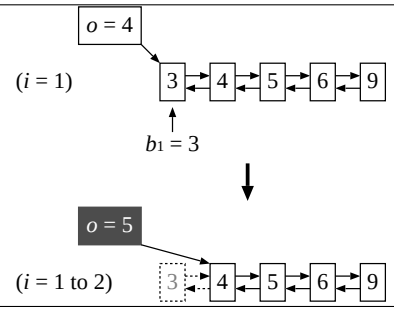


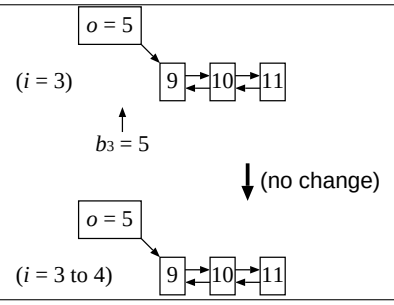
Figure 8: Update procedure of LsAT in the example of Table 2. The list L in i starts with j^* (the largest position attaining the minimum distance o) and then j^2 (the largest positron attaining the distance $o + 1$) and so on, within $scope(i)$. L is updated for i to $i + 1$ as follows: (1) if b_i exists 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: detailed in Fig. 11), 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 List+Base-Search(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 ← 1.
2       b ← |p| − k; e ← |p| + k.
3       Calculate distance d_q(t[1..j], p) for all j ∈ [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, ..., 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 ≥ 2
7       while i < |t| − q + 1 do
8           Calculate the change point c_i.
9           i ← i + 1.
10          b ← b + 1; e ← e + 1.
11          if c ∈ [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 ≤ k then output t[i..j*], where j* is the 1st element of L.
```

Figure 9: Algorithm *List+Base-Search*. Detailed procedures of **DEL-FIRST**, **DEL-MIDDLE** and **INS/DEL-LAST** are shown in Appendix.

Process	How to update doubly-linked list nodes
Reconnect the head to the second cell, delete the first cell and then increase the value of o by 1.	

(1-a) In the case that the node to be deleted is in list L (e.g., in updating from $i = 1$ to 2 in our example)

Do nothing	

(1-b) In the case that the node to be deleted is not in list L (e.g., in updating from $i = 3$ to 4 in our example)

Figure 10: Case study of **DEL-FIRST** step in Algorithm *List+Base-Search* (corresponding to step (1) in Fig. 8). 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.

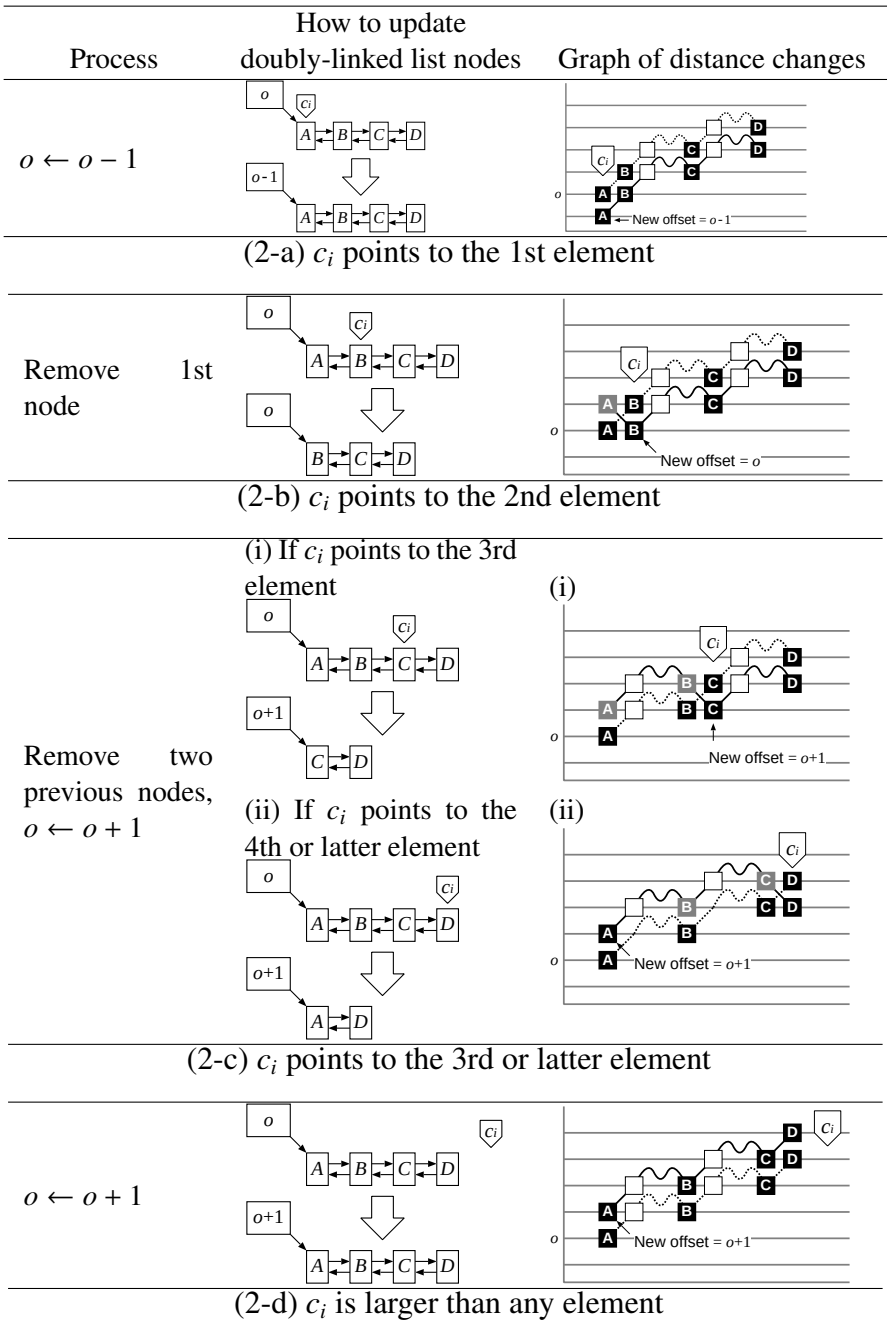
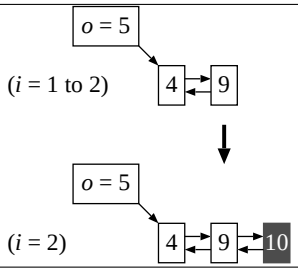
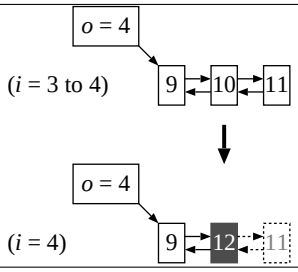


Figure 11: Case study of **DEL-MIDDLE** in Algorithm *List+Base-Search* according to the position of change point c_i (corresponding to step (2) in Fig. 8). From top to bottom: (2-a) in case when c_i points to the 1st element, (2-b) the 2nd, (2-c) the 3rd or latter, (2-d) greater than the last.

Process	How to update doubly-linked list nodes
Insert a new node to the last of L.	

(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 2).

Remove the last node and replace the new last node with $j = e_{i+1}$.	

(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 2).

Figure 12: Case study of **INS/DEL-LAST** step in Algorithm *List+Base-Search* (corresponding to step (3) in Fig. 8). 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.

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

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 $scope(i)$ 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 $scope(i)$ attaining $D[j^*], D[j^*]+1, D[j^*]+2, \ldots$ in order to hold the future candidates of j^*.

Definition 4. *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 rth element of L, denoted by j^r, satisfies $D[j^r] - D[j^*] = r - 1$, and

(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. 8.

The algorithm is shown in Fig. 9. In the algorithm, the fundamental tree operations are **DEL-FIRST**, **DEL-MIDDLE** and **INS/DEL-LAST** as shown in Fig. 8. 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. 10, 11 and 12). Detailed operations for the tree are shown in Appendix.

4.3.2. Complexity analysis

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

Theorem 2. *Let α be the probability that the event $c_i \in scope(i+1)$ happens. Then List+Base-Search runs in $O(\alpha|t| \log k + |t| + |p|)$ time.*

The complexity is better than that in *Array+Base-Search* ($O(\alpha|t|k + |t| + |p|)$ in Theorem 1). From Theorem 2, it is obvious that the algorithm works in $O(|t| \log k + |p|)$ time in the worst case.

Proof of Theorem 2. Assume that, if $c_i \notin scope(i+1)$ 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 $c_i \in scope(i+1)$ and the other for $c_i \notin scope(i+1)$, 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 $c_i \in scope(i+1)$, can be done in $O(\log k)$ time, and
- **DEL-FIRST** and **DEL/INS-LAST**, both of which are needed even if $c_i \notin scope(i+1)$, can be done in $O(\log k)$ time in probability β and $O(1)$ in probability $1 - \beta$.

First we show 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.

Next we analyze the complexities of **DEL-FIRST** and **DEL/INS-LAST**. We show that **DEL-FIRST** can be done in $O(\log k)$ time in probability β ($\beta \leq \alpha$) and

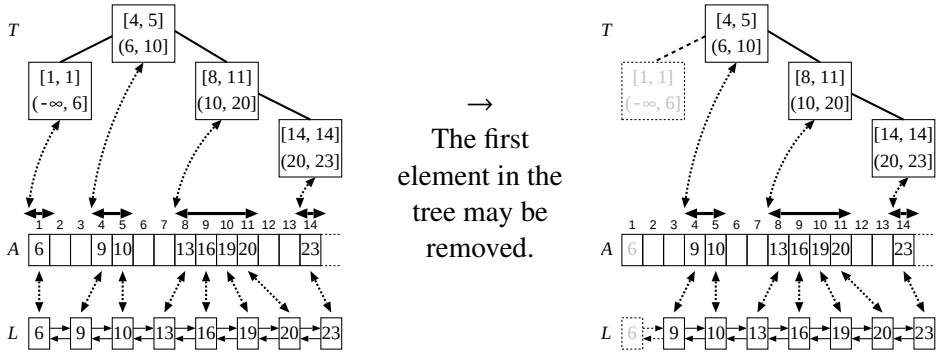


Figure 13: An example of removing the leftmost node of T by **DEL-FIRST**

$O(1)$ 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. 13). 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).

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 is, when $c_i \in scope(i+1)$ 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. □

Under our assumption $k = \Theta(|p|)$, this complexity becomes $O(\alpha|t| \log |p| + |p|)$. In a way similar to that for *Array+Base-Search* (Corollaries 1 and 2), we obtain the following two corollaries.

Corollary 3. *List+Base-Search runs in $O(|t| + |p|)$ time if $\alpha = O(1/|t|)$ or $\alpha = O(1/\log |p|)$ provided that $k = \Theta(|p|)$.*

Corollary 4. *Let t consist of characters chosen randomly according to some distribution over Σ. Then Array+Base-Search 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 *List+Base-Search* for random t (Corollary 4) is the same as that of *Array+Base-Search* (Corollary 2). Note that, in addition to the worst-case time complexity, *List+Base-Search* is also advantageous to *Array+Base-Search* 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 *Array+Base-Search*.

5. Experiment

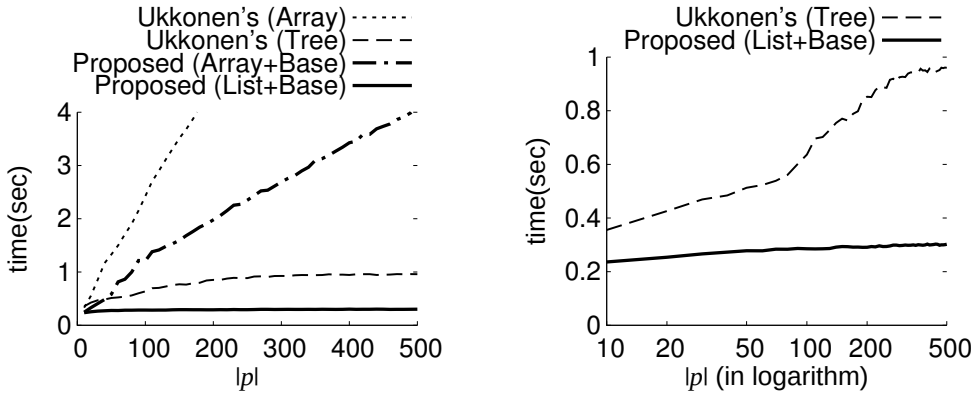
We verified in an experiment the theoretical computational complexities of the proposed two algorithms *Array+Base-Search* and *List+Base-Search*, and Ukkonen's two original algorithms *Array-Search* and *Tree-Search*.

5.1. Settings

The experiment was carried out in the following way:

- We composed a text string t of length 100,000 by randomly choosing the characters from Σ ($|\Sigma| = 20$) with equal probability. For each of $|p| \in \{10, 20, 30, \ldots, 500\}$, we generated 100 patterns p by taking substrings of length $|p|$ from t starting from random positions.
- We enumerated 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, we measured only the time consumed both for construction of the suffix tree of p and for similar substring searching. We averaged the time for 100 trials in each $|p|$.

We used the red-black tree [19] for realizing the search trees in *List+Base-Search* and *Tree-Search*. 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.



(a) Four algorithms (b) *List+Base-Search* and *Tree-Search*

Figure 14: Computation time of similar substring searching ($|t| = 100,000, |\Sigma| = 20, q = 5$)

5.2. Result

The result is shown in Fig. 14. Only $|p|$ is changed. As seen in Fig. 14(a), the proposed algorithms *Array+Base-Search* and *List+Base-Search* improved Ukkonen's corresponding algorithms *Array-Search* and *Tree-Search*, respectively.

Let us discuss the complexity of those four algorithms in the setting of this experiment. We examined the complexity in $|p|$. Noticing $|t|$ $(= 100,000) \gg |p|$ (≤ 500) and $|t|$ is constant, we may omit $|t|$ and $|p|$ from 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 *Array-Search*: $O(|t|k) = O(|t||p|)$,
- Ukkonen's *Tree-Search*: $O(|t| \log k + |p|) = O(|t| \log |p| + |p|) = O(|t| \log |p|)$,
- proposed *Array+Base-Search*: $O(|t|+|p|) = O(|t|) = O(1)$, if $q \geq 2 \log_{1/r_{max}} |p|$, and
- proposed *List+Base-Search*: $O(|t|+|p|) = O(|t|) = O(1)$, if $q \geq \log_{1/r_{max}}(|p| \log |p|)$.

The conditions for the last two algorithms are come from Corollaries 2 and 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. 14, we can see that the results are consistent with their theoretical complexities derived above except for *Array+Base-Search*: linear, logarithmic

and constant in $|p|$ for *Array-Search*, *Tree-Search* and *List+Base-Search*, respectively. A possible explanation for *Array+Base-Search* 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 *List+Base-Search* is sufficiently satisfied by $5 \geq \log_{20}(500 \log 500) = 2.7$.

6. Conclusion

We have shown that the similar substring searching in the q-gram distance can be made in $O(|t| + |p|)$ time, co-linear in text length and pattern length, on average under a mild condition. The proposed algorithms exploit the fact that similar strings are often found at very close positions if the starting positions of them are close. If the problem is really the case, we are able to avoid many steps for searching. The average-case linearity of the time complexity is guaranteed by assuming that the alphabet size or the value of q is large compared with the pattern length. From the viewpoint of approximation of similar strings in the edit distance, the larger value of q is, the greater the effect is [3]. Therefore the proposed algorithm increased the availability more than so far.

In the future work, we will consider relaxing the current condition that each character of t is distributed independently and identically. In addition, we will confirm the practical efficiency of the proposed algorithm experimentally by applying these algorithms.

Appendix

A. Detailed procedures and time complexities of DEL-FIRST, DEL-MIDDLE and INS/DEL-LAST in algorithm *List+Base-Search*

In this appendix we show the detailed procedures of **DEL-FIRST**, **DEL-MIDDLE** and **INS/DEL-LAST** in the algorithm *List+Base-Search* (Section 4.3) together with their time complexities.

A.1. Operation DEL-FIRST

In operation **DEL-FIRST**, as shown in Fig. 13 (Section 4.3.2), 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 4.3.2, 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 $\beta|t|O(\log k) + (1-\beta)|t|O(1) = O(\alpha|t| \log k + |t|)$.

A.2. Operation **DEL-MIDDLE**

In operation **DEL-MIDDLE**, we remove at most two successive elements from L (Fig. 11 in Section 4.3.1). First we just erase the corresponding cells in A. Then we arrange the nodes of T (Fig. A.1). 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 $c_i \in scope(i+1)$, the time complexity for **DEL-MIDDLE** in total is $\alpha|t| \cdot O(\log k) = O(\alpha|t| \log k)$.

A.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 (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 $1-\beta$.

References

[1] E. Ukkonen, Approximate string-matching with q-grams and maximal matches, Theor. Comput. Sci. 92 (1992) 191–211.

[2] A. M. Sokolov, Vector representations for efficient comparison and search for similar strings, Cybern. and Syst. Analysis 43 (4) (2007) 484–498.

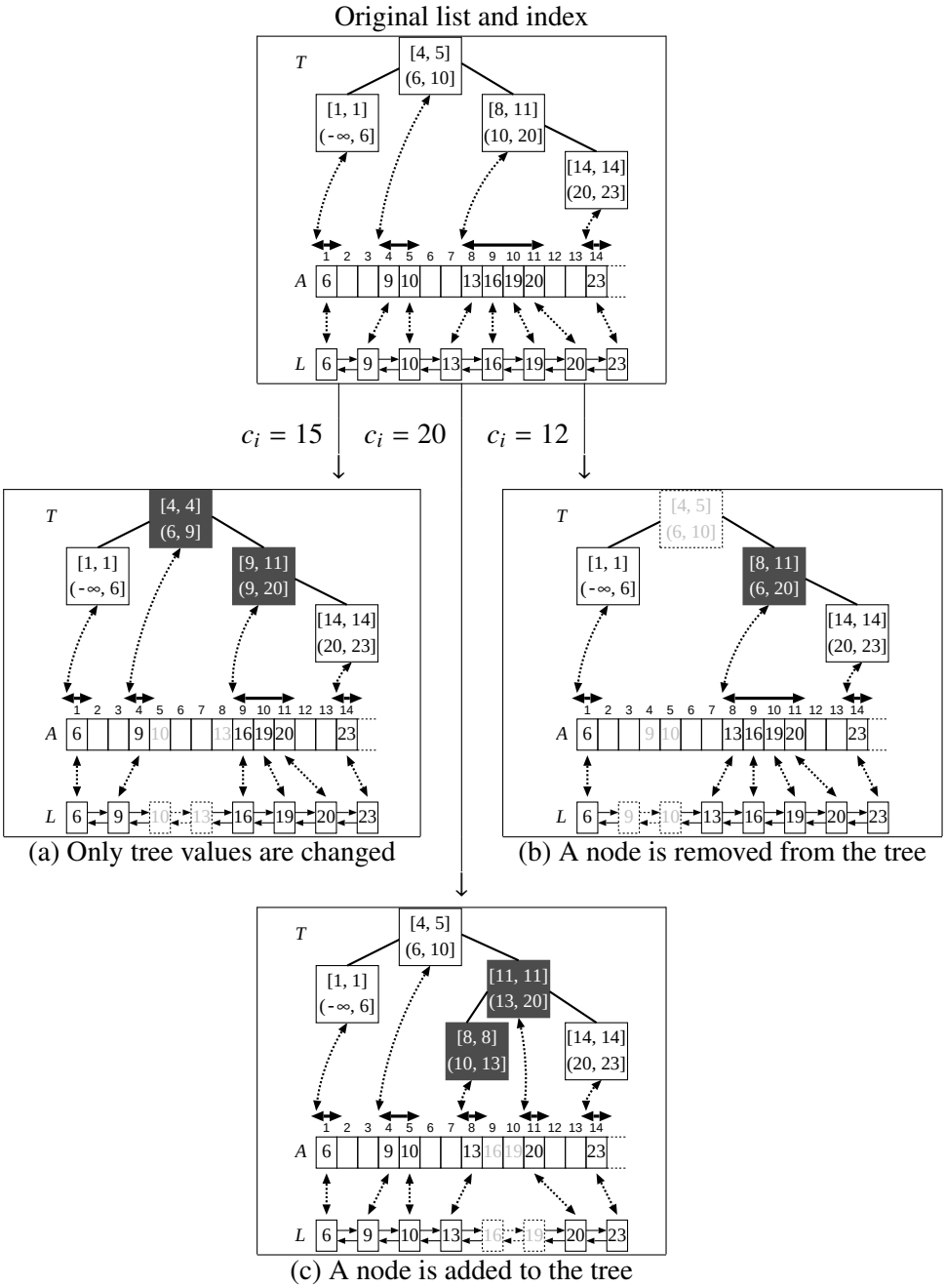


Figure A.1: Case study of removing two elements in LsAT by **DEL-MIDDLE**. There are three cases of (a) only node values are changed in T, (b) one or two nodes are removed from T, and (c) a node is added to T. These processes can be done in $O(\log k)$ time in T and $O(1)$ time in A and L.

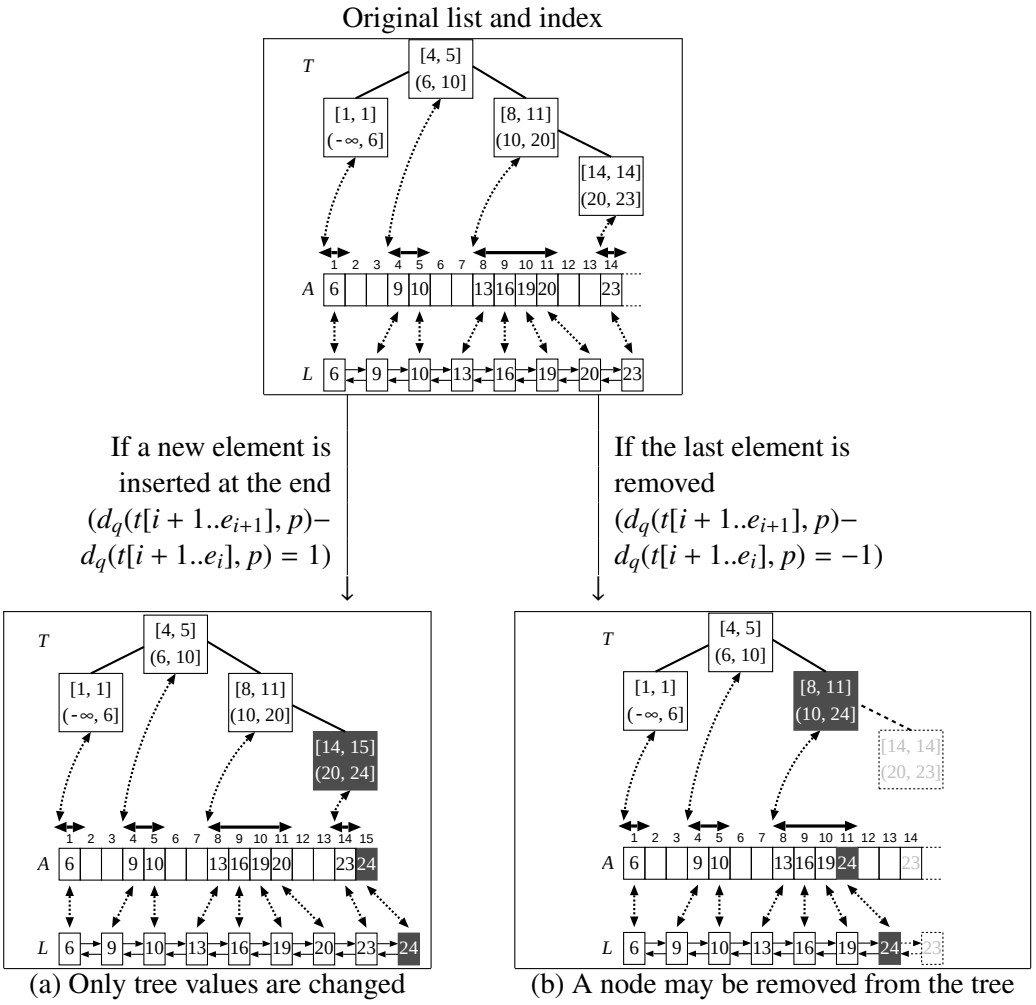


Figure A.2: Case study of **INS/DEL-LAST** in LsAT. (a) If we add an element at the end, we change the values of the rightmost node in T without adding or removing any node. (b) If we remove an element at the end, the rightmost node in T is removed if it holds only one list element, otherwise we have 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 $1 - \beta$.

[3] H. Hanada, A. Nakamura, M. Kudo, A practical comparison of edit distance approximation algorithms, in: Proc. 2011 IEEE Int. Conf. on Granul. Comput., 2011, pp. 231–236.

[4] Z. Bar-Yossef, T. S. Jayram, R. Krauthgamer, R. Kumar, Approximating edit distance efficiently, in: Proc. 45th Annu. IEEE Symp. on Found. Comput. Sci., 2004, pp. 550–559.

[5] R. Grossi, F. Luccio, Simple and efficient string matching with k mismatches, Inf. Process. Lett. 33 (3) (1989) 113–120.

[6] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, 1997.

[7] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (1) (2001) 31–88.

[8] B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, Ultrafast and memory-efficient alignment of short dna sequences to the human genome, Genome Biol. 10 (3) (2009) R25.1–10.

[9] H. Li, R. Durbin, Fast and accurate short read alignment with burrows-wheeler transform, Bioinforma. 25 (14) (2009) 1754–1760.

[10] E. Ukkonen, Finding approximate patterns in strings, J. Algorithms 6 (1) (1985) 132–137.

[11] G. M. Landau, U. Vishkin, Fast parallel and serial approximate string matching, J. Algorithms 10 (2) (1989) 157–169.

[12] D. E. Knuth, J. H. Morris Jr., V. R. Pratt, Fast pattern matching in strings, SIAM J. on Comput. 6 (2) (1977) 323–350.

[13] P. Weiner, Linear pattern matching algorithms, in: Proc. 14th IEEE Symp. on Switch. and Autom. Theory, 1973, pp. 1–11.

[14] P. H. Sellers, The Theory and Computation of Evolutionary Distances: Pattern Recognition, J. Algorithms 1 (4) (1980) 359–373.

[15] B. Melichar, String matching with k differences by finite automata, in: Proc. 13th Int. Conf. on Pattern Recognit., Vol. 2, 1996.

[16] S. Wu, U. Manber, Fast text searching: Allowing errors, Commun. ACM 35 (10) (1992) 83–91.

[17] W. I. Chang, E. L. Lawler, Sublinear approximate string matching and biological applications, Algorithmica 12 (4) (1994) 327–344.

[18] J. Bentley, Programming pearls: Algorithm design techniques, Commun. ACM 27 (9) (1984) 865–873.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Massachusetts Institute of Technology, 2001.

[20] G. Blom, L. Holst, D. Sandell, Problems and Snapshots from the World of Probability, Springer-Verlag, 1994, Ch. 2 “Basic probability theory I”, pp. 13–22.