Dynamic Edit Distance Table under a General Weighted Cost Function

Heikki Hyyrö[1], Kazuyuki Narisawa[2,3], and Shunsuke Inenaga[4]

[1]Department of Computer Sciences, University of Tampere, Finland
heikki.hyyro@cs.uta.fi
[2]Department of Informatics, Kyushu University, Japan
[3]Japan Society for the Promotion of Science (JSPS)
k-nari@i.kyushu-u.ac.jp
[4]Graduate School of Information Sci. and Electrical Eng., Kyushu University, Japan
inenaga@c.csce.kyushu-u.ac.jp

1 Introduction

String comparison is a fundamental task in theoretical computer science, with applications in e.g., spelling correction and computational biology. *Edit distance* is a classic similarity measure between two given strings A and B. It is the minimum total cost for transforming A into B, or vice versa, using three types of edit operations: single-character insertions, deletions, and/or substitutions.

Landau et al. [1] introduced the problem of left incremental edit distance computation: Given a solution for the edit distance between A and B, the task is to compute a solution for the edit distance between A and B', where $B' = bB$. The alternative problem in which B and B' are interchanged is called the left decremental edit distance computation. Applications of left incremental/decremental edit distance computation include cyclic string comparison and computing approximate periods (see [1–3] for more).

Let m and n be the lengths of A and B, respectively. The basic dynamic programming method for the above problem requires $\Theta(mn)$ time per added/deleted character in front of B. For a unit edit cost function (the insertion, deletion, and substitution costs are all 1), Landau et al. [1] presented a fairly complicated $O(k)$-time algorithm, where k is an error threshold with $1 \leq k \leq \max\{m, n\}$. When k is not specified, then the algorithm takes $O(m + n)$ time. Other $O(m + n)$-time solutions for the unit cost function were presented in [2–4].

This paper deals with a more general, weighted edit cost function: we allow the edit cost function to have arbitrary non-negative integer costs. Schmidt [2] presented a complicated $O(n \log m)$ time solution per added/deleted character for a general cost function. In this paper, we present a simple $O(\min\{c(m + n), mn\})$-time algorithm for the same problem, where c is the maximum weight in the cost function. This translates into $O(m+n)$ time under constant weights. Our algorithm uses a *difference table*, a representation of a dynamic programming table proposed by Kim and Park [3]. We also show that the algorithm of Kim and Park is not easily applicable to the case of general weights.

We report some preliminary experimental results which show advantages of our algorithm over a basic dynamic programming method for general edit cost functions and the Kim-Park algorithm for the unit cost function.

2 Preliminaries

Let Σ be a finite *alphabet*. An element of Σ is called a *character* and that of Σ^* is called a *string*. The empty string is denoted by ε. For any string $A = a_1 a_2 \cdots a_m$, let $A[i:j] = a_i \cdots a_j$ for $1 \leq i \leq j \leq m$. For convenience, let $A[i:j] = \varepsilon$ if $i > j$.

For any string $A = a_1 a_2 \cdots a_m$, we define the three editing operations:

1. Insert character b after position i of A, where $i = 0$ means inserting at front.
2. Delete character a_i from position i of A.
3. Substitute character b for character a_i at position i of A.

The above operations can be represented as pairs (ε, b), (a_i, ε), and (a_i, b), respectively. Each (x, y) has a positive cost function $\delta(x, y)$. That is, $\delta : (\{\varepsilon\} \times \Sigma) \cup (\Sigma \times \{\varepsilon\}) \cup (\Sigma \times \Sigma) \rightarrow \mathcal{N}$, where $\mathcal{N}$ denotes the set of non-negative integers. For any $a, b \in \Sigma$, we assume $\delta(a, b) = 0$ if $a = b$, and $\delta(a, b) > 0$ otherwise.

The *edit distance* of between strings A and B under cost function δ is the minimum total cost of editing operations under δ that transform A into B, or vice versa. Such an edit distance between A and B under δ is denoted by $ed_\delta(A, B)$.

The fundamental solution for $ed_\delta(A, B)$ is to compute a dynamic programming table D of size $(m+1) \times (n+1)$ s.t. $D[i, j] = ed_\delta(A[1:i], B[1:j])$ for $0 \leq i \leq m$ and $0 \leq j \leq m$, using the well-known recurrence (1) shown below.

$$\begin{aligned} D[i,0] &= \textstyle\sum_{h=1}^{i} \delta(a_h, \varepsilon) \text{ for } 0 \leq i \leq m, \\ D[0,j] &= \textstyle\sum_{h=1}^{j} \delta(\varepsilon, b_h) \text{ for } 0 \leq j \leq n, \text{ and} \\ D[i,j] &= \min\{D[i,j-1] + \delta(\varepsilon, b_j), D[i-1,j] + \delta(a_i, \varepsilon), \\ &\qquad D[i-1,j-1] + \delta(a_i, b_j)\}, \text{ for } 1 \leq i \leq m \text{ and } 1 \leq j \leq n. \end{aligned} \tag{1}$$

As seen above, for a given D-table for A and $B[1:j]$, we are able to compute $ed_\delta(A, B[1:j+1])$ and $ed_\delta(A, B[1:j-1])$ in $O(m)$ time. This paper deals with the symmetric problem of *left incremental (resp. decremental) edit distance computation*: Given a representation of $ed_\delta(A, B[j:n])$ for strings A and B, compute a representation of $ed_\delta(A, B[j-1:n])$ (resp. $ed_\delta(A, B[j+1:n])$).

3 The Kim-Park Algorithm

A *unit cost function* δ_1 is s.t. $\delta_1(\varepsilon, b) = 1$ for any $b \in \Sigma$, $\delta_1(a, \varepsilon) = 1$ for any $a \in \Sigma$, and $\delta_1(a, b) = 1$ for any $a \neq b$. The unit cost edit distance is known as the *Levenshtein edit distance* or *edit distance*. This section briefly recalls the algorithm of Kim and Park [3] that solves the problem in $O(m+n)$ time for δ_1.

3.1 Solution for the Unit Cost Function

Essentially the same techniques can be used to solve both the left incremental and decremental problems; as in [3], we concentrate on the decremental problem.

Let D denote the D-table for A and B, and D' denote the D-table for A and $B' = B[2:n]$. We find it convenient to use 1-based column indices with D'. Now column 1 acts as the left boundary column with values $D[i,1] = \sum_{h=1}^{i} \delta(a_h, \varepsilon)$, and columns $j = 2 \ldots n$ obey recurrence (1) in normal fashion. Now $D'[i,j] = ed_\delta(A[1:i], B[2:j])$ and cell (i,j) corresponds to a_i and b_j in both D and D'.

Kim and Park use a difference representation (the DR-table) of the D-table, where each position (i,j) has two fields such that $DR[i,j].U = D[i,j] - D[i-1,j]$ and $DR[i,j].L = D[i,j] - D[i,j-1]$. $DR[i,j].U$ is the difference to the upper neighbor and $DR[i,j].L$ is the difference to the left neighbor when row indices grow downwards and column indices towards right.

Let DR' denote the DR-table of strings A and B'. In what follows, we recall how the Kim-Park algorithm computes the DR'-table from the DR-table.

The Kim-Park algorithm is essentially based on the change table Ch, which is defined under our indexing convention[1] as $Ch[i,j] = D'[i,j] - D[i,j]$.

Lemma 1 ([3]). *For the unit cost function δ_1, each $Ch[i,j]$ is -1, 0, or 1.*

Lemma 2 ([3]). *For any $0 \le i \le m$, let $f(i) = \min\{j \mid Ch[i,j] = -1\}$ if such j exists, and let $f(i) = n$ otherwise. Then, $Ch[i,j'] = -1$ for $f(i) \le j' < n$ and $f(i) \ge f(i-1)$ for $1 \le i \le m$. Also, for any $0 \le j \le n$, let $g(j) = \min\{i \mid Ch[i,j] = 1\}$ if such i exists, and let $g(j) = m+1$ otherwise. Then, $Ch[i',j] = 1$ for $g(j) \le i' \le m$ and $g(j) \ge g(j-1)$ for $1 \le j < n$.*

$Ch[i,j]$ is said to be *affected* if $Ch[i-1,j-1]$, $Ch[i-1,j]$, and $Ch[i,j-1]$ are not of the same value. $DR'[i,j]$ is also said to be affected if $Ch[i,j]$ is affected.

Lemma 3 ([3]). *If $DR'[i,j]$ is not affected, then $DR'[i,j] = DR[i,j]$. If $DR'[i,j]$ is affected, then $DR'[i,j].U = DR[i,j].U - Ch[i,j] + Ch[i-1,j]$ and $DR'[i,j].L = DR[i,j].L - Ch[i,j] + Ch[i,j-1]$.*

By Lemmas 1 and 2, there are $O(m+n)$ affected entries in Ch, and these entries are categorized into two types: (-1)-boundaries and 1-boundaries. Consider the four neighbors $Ch[i-1,j-1]$, $Ch[i-1,j]$, $Ch[i,j-1]$ and $Ch[i,j]$. Among these, the upper-right entry $Ch[i-1,j]$ belongs to the (-1)-boundary if and only if $Ch[i-1,j] = -1$ and at least one of the other three entries is not -1. In similar fashion, the lower-left entry $Ch[i,j-1]$ belongs to the 1-boundary if and only if $Ch[i,j-1] = 1$ and at least one of the other three entries is not 1.

The Kim-Park algorithm scans the (-1)- and 1-boundaries of Ch and computes the affected entries in DR' using Lemma 3.

Theorem 1 ([3]). *The algorithm of Kim and Park [3] transforms DR to DR' in $O(m+n)$ time for δ_1.*

[1] [3] used 0-based indexing with D' and defined $Ch[i,j] = D'[i,j] - D[i,j+1]$.

D

		a	c	a	a	a	a	a
	0	5	10	15	20	25	30	35
a	1	0	5	10	15	20	25	30
b	2	1	5	10	15	20	25	30
b	3	2	6	10	15	20	25	30
b	4	3	7	11	15	20	25	30
b	5	4	8	12	16	20	25	30
c	6	5	4	9	14	19	24	29
a	7	6	5	4	9	14	19	24

D′

		c	a	a	a	a	a
	0	5	10	15	20	25	30
a	1	5	5	10	15	20	25
b	2	6	6	10	15	20	25
b	3	7	7	11	15	20	25
b	4	8	8	12	16	20	25
b	5	9	9	13	17	21	25
c	6	5	10	14	18	22	26
a	7	6	5	10	14	18	22

Ch

		c	a	a	a	a	a
	-5	-5	-5	-5	-5	-5	-5
a	1	0	-5	-5	-5	-5	-5
b	1	1	-4	-5	-5	-5	-5
b	1	1	-3	-4	-5	-5	-5
b	1	1	-3	-3	-4	-5	-5
b	1	1	-3	-3	-3	-4	-5
c	1	1	1	0	-1	-2	-3
a	1	1	1	1	0	-1	-2

Fig. 1. From left to right, D, D' and Ch tables for strings $A =$ `abbbbca` and $B =$ `acaaaaa`, with cost function $\delta(\varepsilon, b) = 5$ for any character b, $\delta(a, \varepsilon) = 1$ for any character a, and $\delta(a, b) = 5$ for any characters $a \neq b$.

3.2 Exponential Lower Bound for a General Cost Function

As became evident in the preceding section, the primary principle of the Kim-Park algorithm could be frased as "trace the x-boundary in Ch for each possible boundary-type x". Here we consider how a direct application of this principle would to a general weighted function δ. An important fact is that now Lemma 1 does not hold. See Fig. 1 that illustrates D, D' and Ch for $A =$ `abbbbca` and $B =$ `acaaaaa`, with $\delta(\varepsilon, b) = 5$ for any $b \in \Sigma$, $\delta(a, \varepsilon) = 1$ for any $a \in \Sigma$, and $\delta(a, b) = 5$ for any $a \neq b$. The entries of the Ch-table have seven different values -5, -4, -3, -2, -1, 0, and 1.

Even if we leave aside the non-trivial question of how to define the possible boundary types under general costs, the feasibility of "tracing each possible x-boundary" seems to depend heavily on the number of different values in Ch.

It was shown in [5] that the number of different values in Ch is constant if each edit operation cost is a constant rational number. Let us now consider an integer cost function that may have an exponential edit cost w.r.t. the length of a given string. We may obtain the following negative result. The proof is omitted due to lack of space.

Theorem 2. *Let $A = a_1 a_2 \cdots a_m$ and $B = b_1 b_2 \cdots b_{m+1}$ be strings such that $a_i \neq a_{i'}$ for any $1 \leq i \neq i' \leq$, $b_j \neq b_{j'}$ for any $1 \leq j \neq j' \leq m+1$, and $a_i \neq b_j$ for any $1 \leq i \leq m$ and $1 \leq j \leq m+1$. Let $\delta(\varepsilon, \sigma) = \delta(\sigma, \varepsilon) = (m-1)2^m - 1$ for any $\sigma \in \Sigma$, $\delta(a_i, b_j) = 2^m$ if $i \neq j$, and $\delta(a_i, b_j) = 2^{i-1}$ if $i = j$. Then, Ch-tables under δ can have $\Omega(2^m)$ different values.*

Due to the above theorem, a natural extension of the Kim-Park algorithm might need to check if there is an x-boundary in the Ch-table for exponentially many different values x. Note e.g. from Fig. 1 that different boundaries do not all begin from column 1, and now the boundaries may also be non-contiguous, making their tracing more difficult. Also note the input strings may define which of the $\Omega(2^m)$ values actually appear within the $O(mn)$ entries of the Ch-table.

In part due to these difficulties, we propose in the next section an algorithm that discards the notion of tracing boundaries.

4 A Simple Algorithm for a General Cost Function

As became evident in the preceding discussion, the essential question in incremental/decremental edit distance computation is: Which entries in DR do we need to change in order to transform DR into DR'? The algorithm of Kim and Park finds such changed entries by traversing the affected entries of the Ch-table.

We ignore the Ch-table and concentrate only on the difference table DR (and its updated version DR'). Recurrence (1) showed how to compute the value $D[i,j]$ when the three neighboring values $D[i,j-1]$, $D[i-1,j]$ and $D[i-1,j-1]$ are known. Consider Fig. 2, where the values of $D[i,j-1]$, $D[i-1,j]$ and $D[i,j]$ are represented by using $D[i-1,j-i] = d$ as a base value. Now $DR[i-1,j].L = x$ and $DR[i,j-1].U = y$.

	$j-1$	j
$i-1$	d	$d+x$
i	$d+y$	$d+z$

Fig. 2. Illustration of computing $DR[i,j]$.

Computing $DR[i,j]$ consists of computing $DR[i,j].U = d+z-(d+x) = z-x$ and $DR[i,j].L = d+z-(d+y) = z-y$. If we assume that $DR[i-1,j].L = x$ and $DR[i,j-1].U = y$ are already known, then the only missing value is z. Based on recurrence (1), the relationship between the values in Fig. 2 fulfills the condition $d+z = \min\{d+y+\delta(\varepsilon,b_j), d+x+\delta(a_i,\varepsilon), d+\delta(a_i,b_j)\}$. Since d appears in each choice within the min-clause, we may drop it from both sides. Now $z = \min\{y+\delta(\varepsilon,b_j), x+\delta(a_i,\varepsilon), \delta(a_i,b_j)\}$. This leads directly into the following recurrence (2) for the entry $DR[i,j]$.

$$
\begin{aligned}
&DR[i,0].U = \delta(a_i,\varepsilon) \text{ for every } 1 \le i \le m,\\
&DR[0,j].L = \delta(\varepsilon,b_j) \text{ for every } 1 \le j \le n, \text{ and}\\
&DR[i,j].U = z - DR[i-1,j].L \text{ and } DR[i,j].L = z - DR[i,j-1].U, \text{where}\\
&z = \min\{DR[i-1,j].L + \delta(\varepsilon,b_j), DR[i,j-1].U + \delta(a_i,\varepsilon), \delta(a_i,b_j)\}, \text{ for}\\
&\quad \text{every } 1 \le i \le m \text{ and every } 1 \le j \le n.
\end{aligned}
\tag{2}
$$

To avoid references to the Ch-table, we use the following alternative to decide if $DR'[i,j]$ is affected, that is, if it is possible that $DR'[i,j] \neq DR[i,j]$. The next lemmas follows directly from recurrence (2).

Lemma 4. *For $1 \le i \le m$ and $1 \le j \le n$, the entry $DR'[i,j]$ is affected if and only if $DR'[i-1,j].L \neq DR[i-1,j].L$ or $DR'[i,j-1].U \neq DR[i,j-1].U$.*

Our algorithm for transforming DR into DR' uses Lemma 4 to keep track of which entries in DR may become different. All such affected entries are recomputed using recurrence (2).

The columns $j = 1 \ldots n$ of DR' are processed one column at a time in the order of increasing j. During the computation we maintain a $prev\Delta$-table as follows: When starting to process column j, the table $prev\Delta$ contains the row numbers i for which $DR'[i,j-1].U \neq DR[i,j-1].U$. These row numbers are recorded in increasing order.

The column $j = 1$ of DR is a special case. It is transformed directly into the first boundary column of DR' by setting $DR'[i, 1].U = \delta(a_i, \varepsilon)$ for $i = 1 \ldots m$. For simplicity we add each $i = 1 \ldots m$, into $prev\Delta$ even if $DR'[i, 1].U = DR[i, 1].U$.

Each column $j = 2 \ldots n$ is processed according to Lemma 4 that states that the value $DR'[i, j]$ needs to be computed (ie. its value may change from $DR[i, j]$) only if $DR'[i, j-1].U \neq DR[i, j-1].U$ or $DR'[i-1, j].L \neq DR[i-1, j].L$.

The entries $DR'[i, j]$ are recomputed for all i that appear in $prev\Delta$, in the order of increasing row indices i. This handles all entries in column j where the first condition, $DR'[i, j-1].U \neq DR[i, j-1].U$, of Lemma 4 is true.

The second condition, $DR'[i-1, j].L \neq DR[i-1, j].L$, corresponds to recomputed and consequently changed values in the currently processed column j. This is easily checked during the computation as we proceed along increasing row indices i: Whenever we recompute the entry $DR'[i, j]$, that is, recompute $DR'[i, j].U$ and $DR'[i, j].L$, we also check whether $DR'[i, j].L \neq DR[i, j].L$. If this condition is true, then the next-row entry $DR'[i+1, j]$ is affected and will be recomputed next. This ensures that also all entries $DR'[i, j]$, for which $DR'[i-1, j].L \neq DR[i-1, j].L$ holds, will be recomputed in column j.

In order to prepare the table $prev\Delta$ for the next column $j + 1$, we record each row index i where $DR'[i, j].U \neq DR[i, j].U$ into a second table $curr\Delta$. This is done whenever an entry $DR'[i, j]$ has been computed. When we later move from column j to column $j + 1$, the roles of the tables $prev\Delta$ and $curr\Delta$ are interchanged. Hence the affected row indices recorded into $curr\Delta$ in column j will be read from $prev\Delta$ in column $j + 1$, and the new affected row indices in column $j + 1$ will be recorded to $curr\Delta$, which previously acted as $prev\Delta$ in column j and was holding the affected values for column $j - 1$.

The above-described steps are implemented by Algorithm 1. Let us present the following clarifying comments on the pseudocode of the algorithm:

- In the pseudocode we do not use the separate notation DR' to refer to the transformed version of DR.
- In the pseudocode, the tables $prev\Delta$ and $curr\Delta$ are indexed starting from 1. The variables $prevIdx$ and $currIdx$, respectively, denote the current positions in these tables.
- The end of the table $prev\Delta$ is marked by inserting a sentinel value $m + 1$ as the last value in the table. Also the loop on lines 1-2 does this (and instead leaves out the first row 1, as the computation in any case starts by using the row index $i = 1$).
- Lines 6-8 compute the updated values $DR'[i, j].L$ and $DR'[i, j].U$ into the variables $new.L$ and $new.U$ according to recurrence (2).
- Line 9 stores the old values $DR[i, j].L$ and $DR[i, j].U$ into the variables $old.L$ and $old.U$
- Lines 13-18 check the condition $DR'[i-1, j].L \neq DR[i-1, j].L$ of Lemma 4 in the following way: Line 15 increments the current row index i as if the condition would be true and $i+1$ would be the next row to process. Line 16 checks if this condition was not true. If it was not, the repeat-until reads still unused row indices from the $prev\Delta$-table until either one which is at least

Algorithm 1: Generalized traversal of affected entries

```
1  for  i ← 1 to m do
2      prevΔ[i] ← i + 1;   DR[i, 1].U ← δ(a_i, ε)
3  i ← 1;   j ← 1;   DR[0, j].L ← δ(ε, b_j);   currIdx ← 1;   prevIdx ← 1
4  while  i ≤ m and j ≤ n do
5      while  i ≤ m do
6          x ← DR[i − 1, j].L;   y ← DR[i, j − 1].U
7          z ← min{x + δ(a_i, ε),  y + δ(ε, b_j),  δ(a_i, b_j)}
8          new.L ← z − y;   new.U ← z − x
9          old.L ← DR[i, j].L;   old.U ← DR[i, j].U
10         DR[i, j].L ← new.L;   DR[i, j].U ← new.U
11         if  old.U ≠ new.U then
12             currΔ[currIdx] ← i;   currIdx ← currIdx + 1
13         i ← i + 1
14         if  old.L = new.L then
15             now = i
16             repeat
17                 i ← prevΔ[prevIdx];   prevIdx ← prevIdx + 1
18             until  i ≥ now
19     currΔ[currIdx] ← m + 1
20     Interchange the roles of the tables currΔ and prevΔ
21     currIdx ← 1;   i ← prevΔ[1];   prevIdx ← 2;   j ← j + 1
```

$i + 1$ is found (and it becomes the next row to process) or $prev\Delta$ becomes fully processed (sentinel $m + 1$ was read).
- Line 19 adds the end sentinel $m + 1$ to the table $curr\Delta$.
- Line 20 corresponds in practice to e.g. swapping two pointers that point to the Δ-tables.
- Line 21 already reads the first row value $i = prev\Delta[1]$ for column $j + 1$. Therefore $prevIdx$ becomes 2.
- The main loop of line 4 stops either when line 21 sets $i = m + 1$, which means that the $prev\Delta$-table for the current column was empty, or when the last column n has been processed.

Let $\#_j$ denote the number of actually changed entries in column j. That is, $\#_j = |\{i \ : \ DR'[i, j] \neq DR[i, j]\}|$.

Theorem 3. *Algorithm 1 recomputes a total of $\Theta(m)$ entries in columns $j = 1 \ldots 2$ and a total of $O(\sum_{j=2}^{n} \#_j)$ entries in columns $j = 3 \ldots n$.*

Proof. The case for columns $j = 1 \ldots 2$ follows directly from how the m entries $DR'[i, j].U$ in column 1 are recomputed (ie. reset) and how the $prev\Delta$-table is initialized with $\Theta(m)$ row indices prior to processing column 2.

When the algorithm starts to process a column $j > 2$, the $prev\Delta$-table contains $O(\#_{j-1})$ row indices (and one sentinel). Hence in column j at least these $O(\#_{j-1})$ entries will be recomputed. In addition to these, further entries $DR'[i, j]$ will be recomputed only if the entry $DR'[i-1, j]$ was recomputed and it became

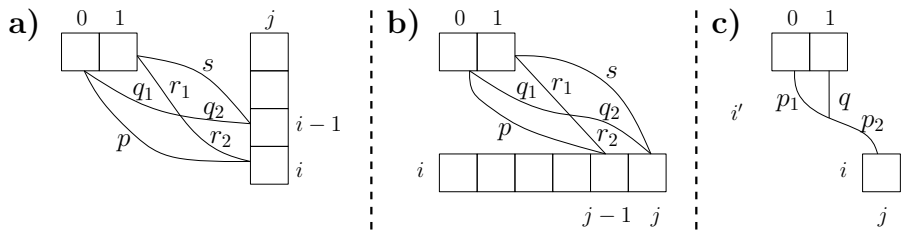

Fig. 3. Illustration of crossing paths in proof of Theorem 4.

different than $DR[i-1,j]$. The number of such entries is at most $\#_j$. No other cells are recomputed in column j. Hence the total number of cells recomputed in any column $j > 2$ is at most $O(\#_{j-1}+\#_j)$. Therefore the total number of entries recomputed in columns $j = 3 \ldots n$ is $O(\sum_{j=3}^{n}(\#_{j-1} + \#_j)) = O(\sum_{j=2}^{n} \#_j)$. □

Theorem 3 states that Algorithm 1 makes the minimum possible work in columns $j > 2$, as clearly any algorithm that transforms DR into DR' must change at least $\#_j$ values in column j. The columns $j = 1$ and $j = 2$ possibly involve up to $\Theta(m)$ unnecessary work due to how the boundary column $j = 1$ and the $prev\Delta$-table are first initialized. Minor modifications and a further preprocessing stage would allow us to make the algorithm completely minimal, ie. to recompute only $O(\#_1 + \#_2)$ entries in the first two columns. We omit this consideration for now, as one of the main goals of this paper is to propose an algorithm that is both general and practical. The current form of Algorithm 1 seems to fulfill this goal well. The pseudocode is compact but nevertheless already provides such a detailed description of the algorithm that it is very straight-forward to compose a working implementation in real code, even if one has little background knowledge. We believe that our Algorithm 1 is not only more general than the previous algorithm of Kim and Park; it also seems even simpler in terms of implementing and understanding all steps of the algorithm. These are valuable qualities in practice.

Corollary 1. *Algorithm 1 transforms DR into DR' in $O(m+n)$ time under the unit cost function δ_1.*

Proof. It follows from Theorems 1 and 3 and the preceding discussion that the work is at most $O(m + n) + \Theta(m) = O(m + n)$. □

Theorem 4. *Let c be the highest weight in the used cost function δ, that is, $c = \max\{\delta(a, b) \ : \ a, b \in \Sigma \cup \{\varepsilon\}\}$. Then $\sum_{j=1}^{n} \#_j = O(c(m + n))$.*

Proof. We analyse the tables DR', DR and Ch in similar fashion as Schmidt in the proof of Theorem 6.1 in [2]. The basis is to consider table D as a weighted grid graph that has a horizontal edge with weight $\delta(\varepsilon, b_j)$ from $D[i, j-1]$ to $D[i, j]$ for $i = 0 \ldots m$ and $j = 1 \ldots n$, a vertical edge with weight $\delta(a_i, \varepsilon)$ from $D[i-1, j]$ to $D[i, j]$ for $i = 1 \ldots m$ and $j = 0 \ldots n$, and a diagonal edge with weight $\delta(a_i, b_j)$ for

$i = 1 \ldots m$ and $j = 1 \ldots n$. Now the edit distance $D[i,j] = ed_\delta(A[1:i], B[1:j])$ is equal to the cost of the cheapest weighted path from $D[0,0]$ to $D[i,j]$.

Let us consider the rows i in column j where $DR'[i,j].U \neq DR[i,j].U$. Since $D'[i,j] = D[i,j] + Ch[i,j]$, we have that $DR'[i,j].U = D'[i,j] - D'[i-1,j] = D[i,j] + Ch[i,j] - D[i-1,j] - Ch[i-1,j] = DR[i,j].U + Ch[i,j] - Ch[i-1,j]$. That is, $DR'[i,j].U \neq DR[i,j].U$ if and only if $Ch[i,j] \neq Ch[i-1,j]$.

Figure 3a depicts minimum cost paths corresponding to the distances $D[i,j] = p$, $D[i-1,j] = q_1 + q_2$, $D'[i,j] = r_1 + r_2$ and $D'[i-1,j] = s$. The path from $D[0,0]$ to $D[i-1,j]$ must cross with the path from $D[0,1]$ to $D[i,j]$. In Figure 3a, the crossing point divides these paths into the subpaths q_1, q_1, r_1 and r_2.

Each path and subpath has a minimal cost, and so the inequalities $p \leq q_1 + r_2$ and $s \leq r_1 + q_2$ hold. Hence $D[i,j] + D'[i-1,j] = p + s \leq q_1 + r_2 + r_1 + q_2 = D[i-1,j] + D'[i,j]$. This leads into the inequality $D'[i-1,j] - D[i-1,j] \leq D'[i,j] - D[i,j]$, that is, $Ch[i-1,j] \leq Ch[i,j]$. Since we deal with integers, $Ch[i,j] \neq Ch[i-1,j]$ iff $Ch[i,j] \geq Ch[i-1,j] + 1$. Note that the $Ch[i,j]$ values are non-decreasing with growing i, and the minimum increment is 1.

Now consider the possible range of values for $Ch[i,j]$ when $i \geq 1$ and $j \geq 1$. The value $D[i,j]$ can never be larger than the alternative of first going to $D[0,1]$ along the edge with weight $\delta(\varepsilon, b_2)$ and then following the minimal path of cost $D'[i,j]$. That is, $Ch[i,j] \geq -\delta(\varepsilon, b_2) \geq c$, where c is the maximum weight in δ. On the other hand, the value $D'[i,j]$ can never be worse than the alternative of going directly down until the path corresponding to $D[i,j]$ is reached in some point $D[i',1]$, and then following that path to the end. This is depicted in Figure 3c so that $D[i,j] = p_1 + p_2$ and q is the cost of the direct downward path from $D[0,1]$ up to the point of crossing $D[i',1]$. The paths (with costs) p_1 and q have the same cost $\sum_{h=1}^{i'-1} \delta(a_h, \varepsilon)$ up to row $i'-1$. It is not difficult to show that $q + p_2 \leq p_1 + p_2 + \min\{\delta(\varepsilon, b_1), \delta(a_{i'}, b_1) - \delta(a_{i'}, \varepsilon)\}$, which means that $Ch[i,j] \leq \min\{\delta(\varepsilon, b_1), \delta(a_{i'}, b_1) - \delta(a_{i'}, \varepsilon)\} \leq c$.

Since $-c \leq Ch[i,j] \leq c$ and the Ch-values are non-decreasing with increments ≥ 1, column j may contain at most $O(c)$ different rows i where $Ch[i,j] \neq Ch[i-1,j]$, that is, at most $O(c)$ different rows i where $DR'[i,j].U \neq DR[i,j].U$.

In similar fashion we may show each row i contains at most $O(c)$ columns j where $DR'[i,j].L \neq DR[i,j].L$. As seen by comparing Figures 3a and 3b, the underlying cases are very similar. We omit further details due to lack of space.

The end result is that columns $j = 1 \ldots n$ contain $O(cn)$ points (i,j) where $DR'[i,j].U \neq DR[i,j].U$, and rows $i = 1 \ldots m$ contain $O(cm)$ points (i,j) where $DR'[i,j].L \neq DR[i,j].L$. Since an entry $DR'[i,j]$ is affected only in the preceding types of points, we may conclude that $\sum_{j=1}^{n} \#_j = O(c(m+n))$. □

Corollary 2. *Algorithm 1 transforms DR into DR' in $O(\min\{c(m+n), mn\})$ time under an arbitrary cost function δ whose maximum weight is c, and in $O(m+n)$ time under a cost function δ with constant (but arbitrary) weights.*

Proof. The $O(mn)$ bound is due to the fact that Algorithm 1 recomputes each of the $O(mn)$ entries at most once. The other bounds follow from Theorems 3 and 4. □

5 Experiments

In all experiments of this section, we computed a representation of D for A and $B[j:n]$ for each $j = n, n-1, \ldots, 1$, where the length of both A and B was n. All the experiments were conducted on a CentOS Linux desktop computer with two 3GHz dual core Xeon processors and 16GB memory.

5.1 Random Data

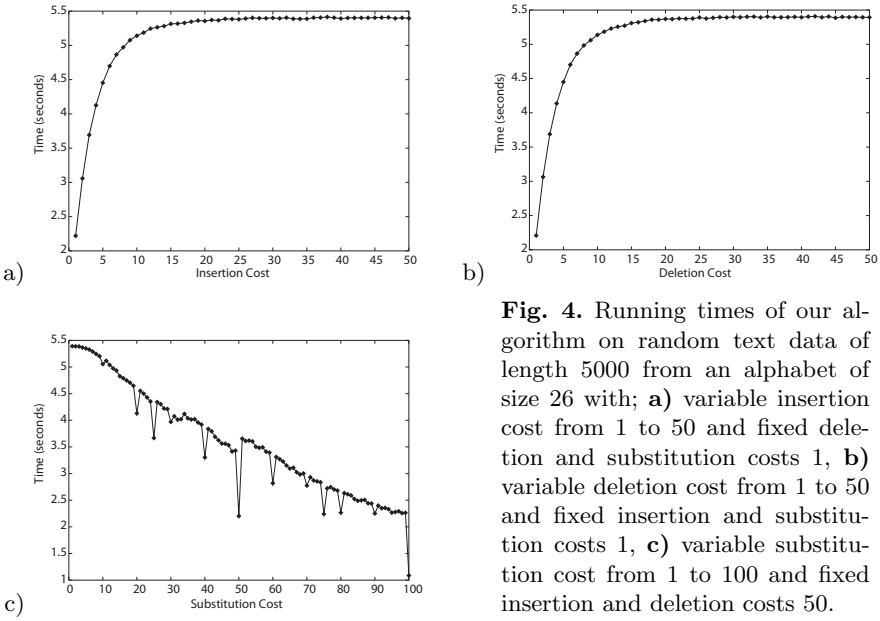

Fig. 4. Running times of our algorithm on random text data of length 5000 from an alphabet of size 26 with; **a)** variable insertion cost from 1 to 50 and fixed deletion and substitution costs 1, **b)** variable deletion cost from 1 to 50 and fixed insertion and substitution costs 1, **c)** variable substitution cost from 1 to 100 and fixed insertion and deletion costs 50.

First we performed some experiments to investigate the performance of our algorithm under various edit operation costs. The running times shown in this section are average times for 10 runs with randomly generated string pairs.

Figures 4a-4c show the running times of our algorithm for random texts of length 5000 with an alphabet of size 26.

In the test of Fig. 4a, deletion and substitution costs were fixed to 1 and the insertion cost varied from 1 to 50. Fig. 4b is otherwise similar, but now the insertion and substitution costs were fixed to 1 and the deletion cost varied from 1 to 50. From these it is evident how our algorithm becomes slower as the insertion or deletion cost becomes larger.

Fig. 4c corresponds to a test where the insertion and deletion costs were fixed to 50 and the substitution cost varied from 1 to 100.

Next we performed experiments to compare running times of our algorithm with the naive method and the Kim-Park algorithm [3] on random text data,

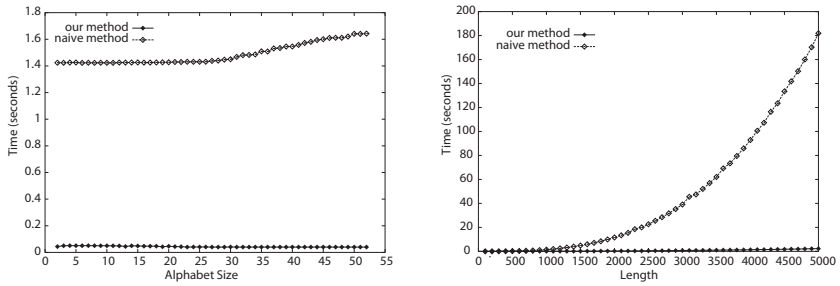

Fig. 5. Running times of our algorithm and the naive method on random text data with; (left) fixed length 1000 and variable alphabet sizes from 2 to 52, (right) fixed alphabet size 26 and variable lengths from 100 to 5000.

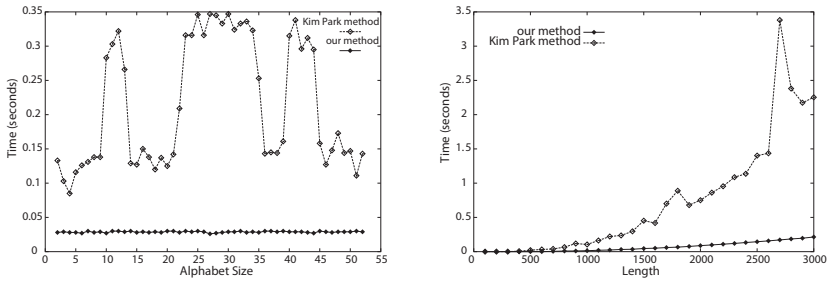

Fig. 6. Running times of our algorithm and the Kim and Park algorithm on random text data with; (left) fixed length 1000 and variable alphabet sizes from 2 to 52, (right) fixed alphabet size 26 and variable lengths from 100 to 3000.

varying the alphabet size and string length as parameters. Ours and the Kim-Park algorithm compute DR-tables, while the naive method computes D-tables.

Fig. 5 shows running times of our algorithm and the naive method on random text data with; (left) fixed length 1000 and variable alphabet sizes from 2 to 52, (right) fixed alphabet size 26 and variable lengths from 100 to 5000. In these experiments, the insertion, deletion, and substitution costs for our algorithm and the naive method were randomly selected to be 137, 116 and 242, respectively.

Fig. 6 shows running times of our algorithm and the Kim-Park algorithm under the unit cost function on random text data with; (left) fixed length 1000 and variable alphabet sizes from 2 to 52, (right) fixed alphabet size 26 and variable lengths from 100 to 3000. The Kim-Park algorithm has more variance, probably due to the poor locality of its memory access patterns.

5.2 Corpora Data

In this section, we show our experimental results on data from two corpora: one consists of English texts from Reuters-21578 text categorization test collection[2], and the other of biological data from the canterbury corpus [6].

[2] `http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html`

length	our method	naive method
1000	0.04	1.50
2000	0.27	12.0
3000	0.71	40.4
4000	1.36	97.1
5000	2.29	189

Table 1. Comparison of running times for the Reuters data (in seconds).

length	our method	naive method
1000	0.01	1.43
2000	0.09	11.5
3000	0.23	38.8
4000	0.43	92.8
5000	0.70	181

Table 2. Comparison of running times for the E.coli data (in seconds).

δ	ε	A	C	G	T
ε	-	3	3	3	3
A	3	0	2	1	2
C	3	2	0	2	1
G	3	1	2	0	2
T	3	2	1	2	0

Table 3. Cost function for the E.coli data.

Table 1 compares the running times of our algorithm and the naive method when processing English text. In this experiment, we used the same randomly selected insertion, deletion, and substitution costs which are 137, 116 and 242, respectively. For each length $l = 1000, 2000, 3000, 4000, 5000$, we randomly selected 10 files of length around l and performed left incremental edit distance computation between each possible file pair within the selected similar-length files. The table shows the average time in seconds over all computations with the given length.

Table 2 shows a similar comparison when processing DNA sequences from "E.coli", the complete genome of the E. Coli bacterium of length 4638690. For each length $l = 1000, 2000, 3000, 4000, 5000$, we randomly picked 10 substrings of length l and performed left incremental edit distance computation between each equal-length substring pair. In this experiment we used the cost function shown in Table 3, which was proposed in [7] for weighted edit distance computation between DNA sequences.

The difference between the highest costs 242 and 3 in these two experiments seemed to result in the difference of running times of our algorithm, since our algorithm runs in $O(\min\{c(m+n), mn\})$ time, where c is the highest cost used.

References

1. Landau, G.M., Myers, E.W., Schmidt, J.P.: Incremental string comparison. SIAM J. Comp. **27**(2) (1998) 557–582
2. Schmidt, J.P.: All highest scoring paths in weighted grid graphs and their application in finding all approximate repeats in strings. SIAM J. Comp. **27**(4) (1998) 972–992
3. Kim, S.R., Park, K.: A dynamic edit distance table. J. Disc. Algo. **2** (2004) 302–312
4. Hyyrö, H.: An efficient linear space algorithm for consecutive suffix alignment under edit distance. In: Proc. SPIRE'08. Volume 5280 of LNCS. (2008) 155–163
5. Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances. J. Comput. Syst. Sci. **20**(1) (1980) 18–31
6. Arnold, R., Bell, T.: A corpus for the evaluation of lossless compression algorithms. In: Proc. DCC'97. (1997) 201–210 `http://corpus.canterbury.ac.nz/`.
7. Kurtz, S.: Approximate string searching under weighted edit distance. In: Proc. 3rd South American Workshop on String Processing (WSP'96). (1996) 156–170