Journal of Discrete Algorithms 34 (2015) 2–17

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Dynamic edit distance table under a general weighted cost function

Heikki Hyyrö [a,*], Kazuyuki Narisawa [b], Shunsuke Inenaga [c]

[a] *School of Information Sciences, University of Tampere, Finland*
[b] *Graduate School of Information Sciences, Tohoku University, Japan*
[c] *Department of Informatics, Kyushu University, Japan*

A R T I C L E I N F O

Article history:
Available online 27 May 2015

Keywords:
Edit distance
General cost functions
Dynamic programming
Incremental string comparison

A B S T R A C T

We discuss the problem of edit distance computation under a dynamic setting, where one of the two compared strings may be modified by single-character edit operations and we wish to keep the edit distance information between the strings up-to-date. A previous algorithm by Kim and Park (2004) [6] solves a more limited problem where modifications can be done only at the ends of the strings (so-called decremental or incremental edits) and the edit operations have (essentially) unit costs. If the lengths of the two strings are m and n, their algorithm requires $O(m+n)$ time per modification. We propose a simple and practical algorithm that (1) allows arbitrary non-negative costs for the edit operations and (2) allows the modifications to be done at arbitrary positions. If the latter string is modified at position $j^\star$, our algorithm requires $O(\min\{rc(m+n), mn\})$ time, where $r = \min\{j^\star, n - j^\star + 1\}$ and c is the maximum edit operation cost. This equals $O(m+n)$ in the simple decremental/incremental unit cost case. Our experiments indicate that the algorithm performs much faster than the theoretical worst-case time limit $O(mn)$ in the general case with arbitrary edit costs and modification positions. The main practical limitation of the algorithm is its $\Theta(mn)$ memory requirement for storing the edit distance information.

1. Introduction

1.1. String comparison with edit distance

String comparison is a fundamental task in theoretical computer science, with important applications to, e.g., spelling correction, file comparison, and bioinformatics [2,5]. A natural and classical way of comparing a string A of length m with another string B of length n is to calculate the minimum cost of transforming A into B (or vice versa), which is called the *edit distance* between A and B. The most basic version of the edit distance allows the following three types of edit operations: inserting a character to an arbitrary position in A (insertion), deleting a character from an arbitrary position in A (deletion), and replacing a character c at an arbitrary position in A with another character c' with $c' \neq c$ (substitution), and each edit operation is assigned to a positive number called an edit cost. Assuming $n \geq m$, there is a well-known dynamic programming algorithm which computes the edit distance between A and B in $\Theta(nm)$ time using $\Theta(m)$ space [16]. Given

[*] Corresponding author.
E-mail addresses: heikki.hyyro@uta.fi (H. Hyyrö), narisawa@ecei.tohoku.ac.jp (K. Narisawa), inenaga@inf.kyushu-u.ac.jp (S. Inenaga).

http://dx.doi.org/10.1016/j.jda.2015.05.007
1570-8667/© 2015 Elsevier B.V. All rights reserved.

a new character b appended to B, this algorithm is able to compute the edit distance between A and Bb in $\Theta(m)$ time, which we call *right incremental* edit distance computation. The method also supports the inverse operation of removing the last character from B, which we call *right decremental* distance computation, in $\Theta(m)$ time.

1.2. Incremental/decremental edit distance computation

Landau et al. [9] introduced the problem of *incremental* edit distance computation: Given a representation of the solution for the edit distance between A and B, the task is to efficiently support both *left incremental* and *right incremental* distance computation. Left incremental computation refers to computing edit distance between A and bB, that is, updating the distance information after a new character b has been prepended to B.

In similar manner, the inverse problem of *decremental* edit distance computation is to efficiently support both left decremental and right decremental distance computation, which refer to updating the distance information after the first or last character from B has been removed.

Incremental/decremental edit distance computation is useful for cyclic string comparison and computing approximate periods (see [9,13,6] for details). Another application example is computing edit distances for a sliding window.

If we apply the basic dynamic programming algorithm of [16] to the incremental/decremental edit distance problem, it requires $\Theta(mn)$ time per added/deleted character at the beginning of B. For the unit edit cost function where the costs for insertion, deletion, and substitution are all 1, Landau et al. [9] presented an involved $O(k)$-time algorithm, where k is an error threshold with $1 \le k \le \max\{m, n\}$. When k is not specified, the algorithm takes $O(m+n)$ time. Kim and Park [6] gave a simpler $O(m+n)$-time solution to the same problem under the unit cost function [6].

The main emphasis of this paper is to deal with a general weighted edit cost function: we allow the edit cost function to have arbitrary non-negative integer costs. We present a simple $O(\min\{c(m+n), mn\})$-time algorithm for incremental/decremental weighted edit distance computation, where c is the maximum weight in the cost function. This translates into $O(m+n)$ time under constant weights.

In similar fashion to the Kim and Park [6] algorithm (KP) for incremental/decremental computation under unit cost operations, our algorithm represents the dynamic programming matrix as a *difference table*. We show that the KP algorithm is not easily applicable to the case of general weights and then propose a new algorithm that is simpler than KP but supports arbitrary weights. Our experiments indicate that our algorithm is substantially faster in practice than KP when unit cost weights are used. Under general weights, our algorithm clearly outperforms the alternative of using basic weighted dynamic programming.

Part of these results appeared in the preliminary version of this paper [4].

1.3. Interactive edit distance computation

We also consider the following even more general problem: Assume we have a representation of the solution to the edit distance between two strings A and B of lengths m and n, respectively. Then, given any position $j^\star$ in B, compute the edit distance between (1) A and the string obtained by deleting the character at position $j^\star$ in B, (2) A and the string obtained by inserting a new character immediately before position $j^\star$, or (3) A and the string obtained by substituting a new character for the character at position $j^\star$ in B. We call this the problem of *interactive edit distance computation*. An example of applications is a “real-time” UNIX diff-style [11] facility which enables one to constantly see the current differences between two different versions of program source code files that one is editing.

Our algorithm can readily deal also with interactive distance computation, working in $O(\min\{cr(m+n), mn\})$ time per each operation, where $r = \min\{j^\star, n - j^\star + 1\}$ is the distance from the edited column to either end of the dynamic programming table. We will also show that the bound is tight, namely, for any $j^\star$ and c, there is an instance for which recomputing the difference table requires $\Omega(\min\{cr(m+n), mn\})$ time.

Our incremental/decremental algorithm, and hence its extension to the interactive edit distance problem (denoted `diff`), uses $O(nm)$ space. To see a time-space trade-off, we compare our algorithm with Hirschberg's $O(m)$-space algorithm [3] (denoted `linear`) for the interactive edit distance computation, which is based on the standard right incremental edit distance computation algorithm. Our experiments show that our method `diff` is faster than `linear` in most cases, whilst using more space.

The UNIX diff command is based on the greedy algorithm by Myers and Miller [11] for the unit cost function. Since their algorithm fills the values of the dynamic programming table in ascending order according to the number of errors, it has to fill only a small portion of the dynamic programming table if the compared strings are similar. Zhang et al. [17] proposed a version of the greedy algorithm for general weighted costs. We implemented a greedy algorithm similar to that of Zhang et al. [17] (denoted `greedy`), and compared it with our algorithm `diff`. Our experiments show that `diff` is faster than `greedy` in most cases, whilst using more space.

1.4. Related work

1.4.1. Dist tables

A dist table $dist(A, B')$ for strings A and B' stores the edit distances between the whole A and every substring of B'. Landau [8] posed a question of how efficiently two dist tables can be merged, i.e., given dist tables $dist(A, B')$ and $dist(A, B'')$

for strings A, B' and B'', how efficiently one can compute dist table $dist(A, B)$ for A and $B = B'B''$? It is known that $dist(A, B)$ can be obtained in $O(nm)$ or $O(m^{1.5})$ time from $dist(A, B')$ and $dist(A, B'')$, where m is the length of A and n is the length of B [8,14].

With relation to the dist tables, we also mention that Schmidt [13] gave an algorithm based on dist tables that allows general integer weights and is able to solve the *left* incremental problem in $O(n \log m)$ time per added/deleted character, assuming $m \le n$.

All the above mentioned algorithms for interactive edit distance computation (`diff`, `linear`, and `greedy`) require quadratic time in the worst case for each operation. We remark that all these algorithms store a representation of the edit distances between all prefixes of A and all prefixes of B. Thus, `diff`, `linear`, and `greedy` have more information about string A, while dist tables have more information about string B. Therefore the algorithms we concentrate on this paper and the ones based on dist tables are not directly comparable with each other.

1.4.2. Semi-local string comparison

Given two strings A and B of length m and n respectively, the *semi-local edit distance problem* is to compute the edit distances between (1) the whole A and every substring of B, (2) every prefix of A and every suffix of B, and (3) every suffix of A and every prefix of B. This is a generalization of a dist table which only deals with (1). For the unit cost function, Tiskin [15] showed how to compute an $O(nm)$-space data structure for the semi-local edit distance problem in $O(nm)$ time. We denote this data structure by $slsc(A, B)$. For the unit cost function, his data structure can deal with the incremental edit distance computation in $O(m)$ time. Also, given $slsc(A, B')$ and $slsc(A, B'')$ for string A of length m, and strings B' and B'' of total length n, $slsc(A, B)$ can be computed in $O(m \log(\min\{m, n\}))$ time. His data structure can be extended to any *uniformly* weighted cost function which assigns the following weights: w_M (≥ 0) to every pair of matching characters, w_X ($< w_M$) to every pair of mismatching characters (corresponding to substitution), and w_G ($< w_X/2$) to every pair of a character and a gap (corresponding to deletion or insertion). For any uniformly weighted cost function satisfying the above conditions, his data structure occupies $O(d^2nm)$ space, can be constructed in $O(d^2nm)$ time, and solves the incremental edit distance computation in $O(dm)$ time, where $d = w_M - 2w_G$. Concatenating two representations of the semi-local string comparison takes $O(dm \log(\min\{dn, dm\}))$ time. Sakai [12] showed a generalized data structure to Tiskin's, which deals with a non-uniformly weighted cost function δ' returning an arbitrary integer and requires $O(q^2nm)$ space, where $q = \max_{(a,b)\in\Sigma\times\Sigma}(\max\{0, \delta'(a, b) - \delta'(a, \varepsilon) - \delta'(\varepsilon, b)\})$. Moreover, using the canonical substring decomposition [15,12] of strings, it is possible to construct a data structure of $O(q^2nm)$ space which represents $slsc(A, B_\ell)$ for any string A of length n and any length-ℓ substring B_ℓ of any string B of m. Once this data structure has been constructed, the weighted edit distance between each length-k substring A_k of A and B_ℓ can be computed in $O(qk)$ time. In addition, it is possible to extend the $O(q^2nm)$-size data structure so that it is able to compute $slsc(A_k, B_\ell)$ in $O(q(k + \ell)\log^2(q(k + \ell)))$ time. The above-mentioned scheme is so powerful that slight modifications to it would lead to a time-efficient solution to the interactive edit distance problem with general edit costs. However, the crucial drawback of this approach is the $O(q^2nm)$ space complexity. As will be seen in our experimental results, memory consumption seems to be the main concern already for our $O(nm)$-space method. This implies that trading more space for possible speed-up is not worth while.

2. Preliminaries

Let Σ be an *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 $A = a_1a_2 \cdots a_m$, let $A[i : j] = a_i \cdots a_j$ for $1 \le i \le j \le m$. For convenience, let $A[i : j] = \varepsilon$ if $i > j$.

For any string $A = a_1a_2 \cdots a_m$, we define the three edit 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 a_i by character b at position i of A.

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

The *edit distance* 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 $ed_\delta(A, B)$.

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 $a \ne b$. The unit cost edit distance is known as the *Levenshtein edit distance*.

Let m and n be the lengths of A and B, respectively. 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 $0 \le i \le m$ and $0 \le j \le n$, using the well-known recurrence (1) shown below.

$$D[i, 0] = \sum_{h=1}^{i} \delta(a_h, \varepsilon) \text{ for } 0 \le i \le m,$$

$$D[0,j] = \sum_{h=1}^{j} \delta(\varepsilon, b_h) \text{ for } 0 \le j \le n, \text{ and}$$
$$D[i,j] = \min\{D[i,j-1] + \delta(\varepsilon, b_j), D[i-1,j] + \delta(a_i, \varepsilon),$$
$$D[i-1,j-1] + \delta(a_i, b_j)\}, \text{ for } 1 \le i \le m \text{ and } 1 \le j \le n. \qquad (1)$$

Consider a situation where the table D has been computed for strings $A = A[1:m]$ and $B = B[1:n]$ and B is then modified at some position j'. Since each value $D[i,j] = ed_\delta(A[1:i], B[1:j])$ in the recurrence depends on prior values of form $D[i^-, j^-] = ed_\delta(A[1:i^-], B[1:j^-])$, where $0 \le i^- \le i$ and $0 \le j^- \le j$, the existing values in the D-table will remain valid within the columns $j = 1, \dots, j' - 1$. The values in columns $j = j', \dots n$ need to be recomputed, which takes $O((n - j')m)$ time. This implies that the basic dynamic programming recurrence is efficient for updating D only when the modification to B is done near its end position n.

3. The Kim–Park algorithm

Our algorithm is based on a similar representation of the dynamic programming matrix as the algorithm of Kim and Park [6] (KP) that solves the incremental/decremental problem in $O(m+n)$ time under the unit cost function δ_1. Therefore we begin by reviewing KP. We follow the example of the original work [6] and concentrate on the left decremental problem. Essentially the same technique can be used to solve the left incremental problem, and the right incremental/decremental cases can be handled by basic dynamic programming.

3.1. Left decremental computation under the unit cost function

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 $j = 2 \dots n$ obey recurrence (1) in normal fashion. Furthermore $D'[i,j] = ed_\delta(A[1:i], B[2:j])$ and cell (i,j) corresponds to a_i and b_j in both D and D'.

KP uses 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 KP algorithm computes the DR'-table from the DR-table.

KP is essentially based on a so-called change table Ch, which is defined under our indexing convention[1] as $Ch[i,j] = D'[i,j] - D[i,j]$.

Lemma 1. *(See [6].) For the unit cost function δ_1, each $Ch[i,j]$ is -1, 0, or 1.*

Lemma 2. *(See [6].) 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 $1 \le j < n$.*

Lemmas 1 and 2 say that each the Ch-table contains values -1, 0 and 1 in such manner that -1's appear contiguously in the upper-right part of the table, 1's appear contiguously in the lower-left part, and 0's appear contiguously in between. A consequence is that the values in any given row or column of Ch appear in non-increasing order: first 1's, then 0's and last -1's.

$Ch[i,j]$ 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. *(See [6].) 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.

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

[1] [6] 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 $A = \texttt{abbbbca}$ and $B = \texttt{acaaaaa}$, 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 $a \neq b$.

Theorem 4. *(See [6].) The algorithm of Kim and Park [6] transforms DR to DR′ in $O(m+n)$ time for δ_1.*

3.2. Exponential lower bound for a general cost function

As became evident in the preceding section, the primary principle of the KP algorithm could be phrased as "trace the x-boundary in Ch for each possible boundary-type x". Here we consider how a direct naive application of this principle would work 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 $A = \texttt{abbbbca}$ and $B = \texttt{acaaaaa}$, 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 [10] 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. The following lemma shows a negative result such that the Ch-table can in principle contain exponentially many different values for such an edit cost function.

Theorem 5. *Let $A = a_1a_2\cdots a_m$ and $B = b_1b_2\cdots b_{m+1}$ be strings such that $a_i \neq a_{i'}$ for any $1 \leq i \neq i' \leq m$, $b_j \neq b_{j'}$ 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.*

Proof. Let D and D' be DP tables for $ed_\delta(A, B)$ and $ed_\delta(A, B[2:m+1])$, respectively.

If $m \geq 3$, it holds that

$$\delta(\varepsilon, \sigma) + \delta(\sigma, \varepsilon) - \sum_{i=1}^{m} \delta(a_i, b_{i+1}) = 2(m-1)2^m - 2 - m2^m = (m-2)2^m - 2 > 0 \tag{2}$$

which implies that replacing a_i with b_{i+1} for all $1 \leq i \leq m$ gives the minimum total edit cost for A and B', that is, $D'[m, m+1] = m2^m$.

If $m \geq 2$, it holds that

$$\delta(\varepsilon, \sigma) + \delta(\sigma, \varepsilon) - \sum_{i=1}^{m} \delta(a_i, b_i) = 2(m-1)2^m - 2 - 2^m + 1 = 2(m-1)2^m - 1 > 0 \tag{3}$$

which implies that replacing a_i with b_i for all $1 \leq i \leq m$ gives the minimum total edit cost for A and $B[1:m]$, that is, $D[m, m] = 2^m - 1$. Also, if $m \geq 2$, then $\delta(\varepsilon, b_j) = (m-1)2^m - 1 \geq 2^m - 1 = D[m, m]$ for any j. Moreover, $\delta(a_i, b_j) = 2^m$ for any $i \neq j$. Hence we have

$$D[m, m+1] = 2^m - 1 + \delta(\varepsilon, b_{m+1}) = 2^m - 1 + (m-1)2^m - 1 = m2^m - 2.$$

Thus $Ch[m, m+1] = D'[m, m+1] - D[m, m+1] = 2$.

For any set $U = \{j_1, j_2, \ldots, j_{|U|}\} \subseteq \{1, 2, \ldots, m\}$, let B_U be a string such that

$$B_U[j] = \begin{cases} B[j] & \text{if } j \notin U, \\ a_j & \text{if } j \in U. \end{cases}$$

Let D_U and D'_U be DP tables for $ed_\delta(A, B_U)$ and $ed_\delta(A, B_U[2:m+1])$, respectively. Also, let $Ch_U[i, j] = D'_U[i, j] - D_U[i, j]$. It follows from Equation (2) that $D'_U[m, m+1] = D[m, m+1] = m2^m$. On the other hand, we have that

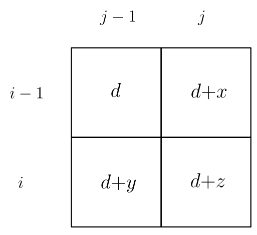

Fig. 2. Illustration of computing $DR[i, j]$.

$$D_U[m, m+1] = D[m, m+1] - \sum_{j \in U} 2^{j-1}$$

by Equation (3) and $\delta(a_i, a_i) = 0$ for any $1 \le i \le m$. Therefore we obtain

$$Ch_U[m, m+1] = Ch[m, m+1] - \sum_{j \in U} 2^{j-1}.$$

Since there are $O(2^m)$ such subsets U of $\{1, 2, \dots, m\}$, the number of possible different values in the Ch-tables is $\Omega(2^m)$. $\square$

Due to the above theorem, a naive extension of KP 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 under a general cost function

As became evident in the preceding discussion, the essential question after B has been modified is as follows: Which entries in DR do we need to change? KP 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. At the same time we also move to consider a more general setting, where B has been modified at some position $j^\star$. Let $B^\star$ denote B after the modification, ℓ denote the length difference between $B^\star$ and B, and $DR^\star$ denote a version of DR that has been preprocessed as follows in accordance to the modification: If character at position $j^\star$ was removed, then $DR^\star$ is DR where column $j^\star$ has been removed (and $\ell = -1$). If a character was inserted before position $j^\star$, then $DR^\star$ is DR after a copy of column $j^\star - 1$ has been inserted directly after column $j^\star - 1$ (and $\ell = 1$). In the case of substitution, $DR^\star$ equals DR (and $\ell = 0$). Note that we use normal 0-based indexing with the table $DR^\star$. Initially, for all rows i, $DR^\star[i, j] = DR[i, j]$ for $0 \le j < j^\star$ and $DR^\star[i, j+\ell] = DR[i, j]$ for $j^\star < j \le n$. Preprocessing DR into $DR^\star$ and modifying B into $B^\star$ takes $O(m+n)$ time.

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 $DR[i, j-1].U = y$.

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 (4) for the entry $DR[i, j]$.

$$\begin{aligned}
&DR[i, 0].U = \delta(a_i, \varepsilon) \text{ for every } 1 \le i \le m,\\
&DR[0, j].L = \delta(\varepsilon, b_j) \text{ for every } 1 \le j \le n, \text{ and}\\
&DR[i, j].U = z - DR[i-1, j].L \text{ and } DR[i, j].L = z - DR[i, j-1].U, \text{ where}\\
&z = \min\{DR[i-1, j].L + \delta(\varepsilon, b_j), DR[i, j-1].U + \delta(a_i, \varepsilon), \delta(a_i, b_j)\}, \text{ for}\\
&\quad \text{every } 1 \le i \le m \text{ and every } 1 \le j \le n.
\end{aligned} \tag{4}$$

We use the following lemma to decide which cells $DR^\star[i, j]$ should be recomputed. It follows directly from recurrence (4).

Lemma 6. *Assume that the values $DR^\star[i', j']$ are correct for all cells where $i' < i$ or $j' < j$. The entry $DR^\star[i, j]$ needs to be recomputed if and only if $DR^\star[i-1, j].L \ne DR[i-1, j-\ell].L$ or $DR^\star[i, j-1].U \ne DR[i, j-1-\ell].U$.*

It is obvious that columns $j < j^\star$ have remained correct when position $j^\star$ has been modified. Our algorithm processes the columns $j = j^\star, \ldots, n$ of $DR^\star$ one column at a time in the order of increasing j. Each column $j = j^\star \ldots n + \ell$ is processed according to Lemma 6 that states that the value $DR^\star[i, j]$ may need to be recomputed (i.e. its value may change from $DR[i, j - \ell]$) only if $DR^\star[i, j-1].U \neq DR[i, j-1-\ell].U$ or $DR^\star[i-1, j].L \neq DR[i-1, j-\ell].L$. To help keep track of such cells, 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 $DR^\star[i, j-1].U \neq DR[i, j-1-\ell].U$. These row numbers are recorded in increasing order. The entries $DR^\star[i, j]$ 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, $DR^\star[i, j-1].U \neq DR[i, j-1-\ell].U$, of Lemma 6 is true.

For simplicity, our strategy is to always recompute all values in the first processed column $j^\star$ of $DR^\star$. This is achieved by initializing the $prev\Delta$-table to contain all $i = 1 \ldots m$, regardless of whether $DR^\star[i, j^\star].U = DR[i, j^\star - \ell].U$. In the remaining columns $prev\Delta$ will hold only indices that truly are known to require recomputation.

The second condition, $DR^\star[i-1, j].L \neq DR[i-1, j-\ell].L$, 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 $DR^\star[i, j]$, that is, recompute $DR^\star[i, j].U$ and $DR^\star[i, j].L$, we also check whether $DR^\star[i, j].L \neq DR[i, j-\ell].L$. If this condition is true, then the next-row entry $DR^\star[i+1, j]$ is affected and will be recomputed next. This ensures that also all entries $DR^\star[i, j]$, for which $DR^\star[i-1, j].L \neq DR[i-1, j-\ell].L$ 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 $DR^\star[i, j].U \neq DR[i, j - \ell].U$ into a second table $curr\Delta$. This is done whenever an entry $DR^\star[i, j]$ 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:

- The pseudocode does not show the $O(m+n)$ preprocessing of B or $DR^\star$.
- 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 $DR^\star[i, j].L$ and $DR^\star[i, j].U$ into the variables $new.L$ and $new.U$ according to recurrence (4).
- Line 9 stores the old values $DR[i, j-\ell].L$ and $DR[i, j-\ell].U$, which up to this point still were in cell $DR^\star[i, j]$, into the variables $old.L$ and $old.U$.
- Lines 13–18 check the condition $DR^\star[i-1, j].L \neq DR[i-1, j-\ell].L$ of Lemma 6 in the following way: Line 13 increments the current row index i as if the condition would be true and $i+1$ would be the next row to process. Line 14 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 $i+1$ 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, $\#_j = |\{i \ : \ DR^\star[i, j] \neq DR[i, j-\ell]\}|$.

Theorem 7. *Algorithm 1 recomputes a total of $\Theta(m)$ entries in column $j = j^\star$ and a total of $O(\sum_{j=j^\star+1}^{n} \#_j)$ entries in columns $j = j^\star + 1, \ldots, n$.*

Proof. The case for column $j^\star$ follows directly from how we recompute all m entries by initializing $prev\Delta$ with values $i = 1, \ldots, m$.

When the algorithm starts to process a column $j > j^\star$, 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 $DR^\star[i, j]$ will be recomputed only if the entry $DR^\star[i-1, j]$ was recomputed and it became different than $DR[i-1, j-\ell]$. 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 $j = j^\star + 1, \ldots, n$ is $O(\sum_{j=j^\star+1}^{n} (\#_{j-1} + \#_j)) = O(\sum_{j=j^\star+1}^{n} \#_j)$. $\square$

Algorithm 1: Generalized traversal of affected entries.

```
for $i \leftarrow 1$ to $m$ do
    $prev\Delta[i] \leftarrow i+1$;
$i \leftarrow 1$; $j \leftarrow j^\star$; $currIdx \leftarrow 1$; $prevIdx \leftarrow 1$
while $i \le m$ and $j \le n+\ell$ do
    while $i \le m$ do
        $x \leftarrow DR^\star[i-1,j].L$; $y \leftarrow DR^\star[i,j-1].U$
        $z \leftarrow \min\{x+\delta(a_i,\varepsilon),\ y+\delta(\varepsilon,b^\star{}_j),\ \delta(a_i,b^\star{}_j)\}$
        $new.L \leftarrow z-y$; $new.U \leftarrow z-x$
        $old.L \leftarrow DR^\star[i,j].L$; $old.U \leftarrow DR^\star[i,j].U$
        $DR^\star[i,j].L \leftarrow new.L$; $DR^\star[i,j].U \leftarrow new.U$
        if $old.U \ne new.U$ then
            $curr\Delta[currIdx] \leftarrow i$; $currIdx \leftarrow currIdx+1$
        $i \leftarrow i+1$
        if $old.L = new.L$ then
            $now = i$
            repeat
                $i \leftarrow prev\Delta[prevIdx]$; $prevIdx \leftarrow prevIdx+1$
            until $i \ge now$;
    $curr\Delta[currIdx] \leftarrow m+1$
    Interchange the roles of the tables $curr\Delta$ and $prev\Delta$
    $currIdx \leftarrow 1$; $i \leftarrow prev\Delta[1]$; $prevIdx \leftarrow 2$; $j \leftarrow j+1$
```

Theorem 7 states that Algorithm 1 makes the minimum possible work in columns $j > j^\star$, as clearly any algorithm that transforms DR into $DR^\star$ must change at least $\#_j$ values in column j. The column $j = j^\star$ possibly involves up to $\Theta(m)$ unnecessary work as we in any case recompute all values in that column. This work however could also be seen as being part of $O(m+n)$ preprocessing of $DR^\star$ and $B^\star$.

We note that even though the pseudocode of Algorithm 1 is compact, it already largely corresponds to a practical implementation. It should be straight-forward to compose a working implementation in real code, even if one has little background knowledge. We believe that one strength of our algorithm is its relative simplicity. Our view is that the algorithm is considerably simpler to understand and implement than KP even though our algorithm solves a more general problem. This is a valuable quality when considering adopting the algorithm to practical use.

Corollary 1. *Algorithm 1 solves the incremental/decremental problem in $O(m+n)$ time under the unit cost function δ_1.*

Proof. The bound $O(m+n)$ for left incremental/decremental case follows from Theorems 4 and 7, as in that case transforming DR to $DR^\star$ essentially corresponds to transforming DR to DR'. The bound $O(m+n)$ for right incremental/decremental follows from how in that case $j^\star = n$ or $j^\star = n+1$ (the latter correspond to appending to the end), and hence only a constant number of size-$O(m)$ columns will be processed. $\square$

Theorem 8. *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 $\sum_{j=1}^{n} \#_j = O(c(m+n))$ in the left incremental/decremental problem.*

Proof. In this case $DR^\star$ essentially corresponds to DR'. We analyze the tables DR', DR and Ch in similar fashion as Schmidt in the proof of Theorem 6.1 in [13]. 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 $DR'[i,j].U \ne DR[i,j].U$. 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, $DR'[i,j].U \ne DR[i,j].U$ if and only if $Ch[i,j] \ne Ch[i-1,j]$.

Fig. 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 Fig. 3a, the crossing point divides these paths into the subpaths q_1, q_2, r_1 and r_2.

Each path and subpath has a minimal cost, and so the inequalities $p \le q_1+r_2$ and $s \le r_1+q_2$ hold. Hence $D[i,j]+D'[i-1,j] = p+s \le q_1+r_2+r_1+q_2 = D[i-1,j]+D'[i,j]$. This leads into the inequality $D'[i-1,j]-D[i-1,j] \le D'[i,j]-D[i,j]$, that is, $Ch[i-1,j] \le Ch[i,j]$. Since we deal with integers, $Ch[i,j] \ne Ch[i-1,j]$ iff $Ch[i,j] \ge Ch[i-1,j]+1$. 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 $i \ge 1$ and $j \ge 1$. 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, $Ch[i,j] \ge -\delta(\varepsilon,b_2) \ge c$, 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 Fig. 3c so that $D[i,j] = p_1+p_2$ and q is the cost of the

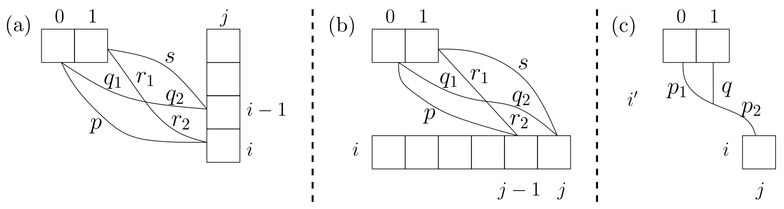

Fig. 3. Illustration of crossing paths in proof of Theorem 8.

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 $q + p_2 \le p_1 + p_2 + \min\{\delta(\varepsilon, b_1), \delta(a_{i'}, b_1) - \delta(a_{i'}, \varepsilon)\}$, which means that $Ch[i, j] \le \min\{\delta(\varepsilon, b_1), \delta(a_{i'}, b_1) - \delta(a_{i'}, \varepsilon)\} \le c$.

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

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

The end result is that columns $j = 1 \ldots n$ contain $O(cn)$ points (i, j) where $DR'[i, j].U \ne DR[i, j].U$, and rows $i = 1 \ldots m$ contain $O(cm)$ points (i, j) where $DR'[i, j].L \ne DR[i, j].L$. 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. *In the incremental/decremental problem, 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 for left incremental/decremental problem follow from Theorems 7 and 8. The right incremental/decremental problem clearly fits within the same bounds as it involves processing a constant number of columns in $O(m)$ time. □

The following theorem gives an upper-bound of our algorithm for the interactive edit distance computation problem, where string B is edited at position $j^\star$.

Theorem 9. *In the interactive edit distance computation problem with an arbitrary modification position* $j^\star$ *and an arbitrary integer cost function* δ *whose maximum weight is c, Algorithm 1 transforms DR into DR** *in* $O(\min\{rc(m + n), mn\})$ *time, where* $r = \min\{j^\star, n - j^\star + 1\}$.

Proof. The bound is immediate when $r = n - j^\star + 1$, as in that case r is the number of columns that the algorithm processes, taking at most $O(rm + n)$ time. When $r = j^\star$, we note that editing position $j^\star$ in a purely left incremental/decremental fashion (first remove the first $j^\star$ characters of B, then prepend the first $j^\star + \ell$ characters of $B^\star$) takes at most $O(rc(m + n))$ time. Since Algorithm 1 recomputes a minimal number of entries in $DR^\star$, it will not recompute more entries than the at most $O(rc(m + n))$ entries that the separate applications of decremental/incremental computation recompute (both approaches essentially eventually transform DR into $DR^\star$). Since Algorithm 1 also never makes more than $O(mn)$ work, the overall bound is $O(\min\{rc(m + n), mn\})$. □

The next theorem shows that the $O(\min\{rc(m + n), mn\})$-bound of Theorem 9 is indeed tight, as long as we maintain the difference tables.

Theorem 10. *For any positive integers m, n and any choice of the modification position* $j^\star$*, there exist strings of lengths m, n each and a cost function* δ *with maximum cost c such that Algorithm 1 requires* $\Omega(\min\{rc(m + n), mn\})$ *work to transform DR into DR***, where* $r = \min\{j^\star, n - j^\star + 1\}$.

Proof. As was claimed in the proof of Theorem 9, the bound is immediate when $r = n - j^\star + 1$. In what follows, we consider the case where $r = j^\star$. Since $n - j^\star + 1 = O(n)$, the $\Omega(j^\star c(m + n))$-bound is of interest only if $j^\star = o(n)$. Hence we suppose $j^\star = o(n)$ in the sequel.

For simplicity, we show an $\Omega(j^\star cn)$ bound. To avoid reverting to the general $O(mn)$ bound, we assume $j^\star c = o(m)$. Let $A = a_1 \cdots a_m$ be a string of length $m > (j^\star - 1)(c - 2)$ whose prefix of length $(j^\star - 1)(c - 2)$ consists of $(j^\star - 1)(c - 2)$ distinct characters, i.e., $a_i \ne a_h$ for any $1 \le i \ne h \le (j^\star - 1)(c - 2)$. Let $B = b_1 \cdots b_n$ be a string of length n such that $b_i \ne b_j$ for any $1 \le i \ne j \le j^\star$, and $b_p = b_{j^\star}$ for all $j^\star < p \le n$. We use the cost function δ which is defined as follows: For convenience, we assume the maximum weight c is even. For each $1 \le j \le j^\star$ and each $1 \le k < c/2$, let $\delta(a_{(j-1)(c-2)+2k-1}, b_j) = c/2 - k$. We

DP		u	v	x	x	x	
	0	5	10	15	20	25	
a	5	4	9	14	19	24	-1
b	10	9	14	19	24	29	5
c	15	13	18	23	28	33	4
d	20	18	23	28	33	38	5
e	25	22	27	32	37	42	4
f	30	27	32	37	42	47	5
g	35	31	36	41	46	51	4
h	40	36	41	46	51	56	5
i	45	41	40	45	50	55	-1
j	50	46	45	50	55	60	5
k	55	51	49	54	59	64	4
l	60	56	54	59	64	69	5
m	65	61	58	63	68	73	4
n	70	66	63	68	73	78	5
o	75	71	67	72	77	82	4
p	80	76	72	77	82	87	5
r	85	81	77	82	87	92	5
r	90	86	82	87	92	97	5
r	95	91	87	92	97	102	5

DP*		u	v	w	x	x	
	0	5	10	15	20	25	
a	5	4	9	11	16	21	-4
b	10	9	14	10	15	20	-1
c	15	13	18	15	20	25	5
d	20	18	23	19	24	29	4
e	25	22	27	24	29	34	5
f	30	27	32	28	33	38	4
g	35	31	36	33	38	43	5
h	40	36	41	37	42	47	4
i	45	41	40	42	47	52	5
j	50	46	45	41	46	51	-1
k	55	51	49	46	51	56	5
l	60	56	54	50	55	60	4
m	65	61	58	55	60	65	5
n	70	66	63	59	64	69	4
o	75	71	67	64	69	74	5
p	80	76	72	68	73	78	4
r	85	81	77	73	78	83	5
r	90	86	82	78	83	88	5
r	95	91	87	83	88	93	5

Fig. 4. A concrete example of Theorem 10, where $j^\star = 3$ and $c = 10$. The cost function δ is defined as follows: $\delta(\mathtt{a},\mathtt{u}) = \delta(\mathtt{i},\mathtt{v}) = 4$, $\delta(\mathtt{c},\mathtt{u}) = \delta(\mathtt{k},\mathtt{v}) = 3$, $\delta(\mathtt{e},\mathtt{u}) = \delta(\mathtt{m},\mathtt{v}) = 2$, $\delta(\mathtt{g},\mathtt{u}) = \delta(\mathtt{o},\mathtt{v}) = 1$, $\delta(s,\mathtt{w}) = 1$ for any $s \in \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{g},\mathtt{h},\mathtt{i},\mathtt{j},\mathtt{k},\mathtt{l},\mathtt{m},\mathtt{n},\mathtt{o},\mathtt{p}\}$, the substitution costs for any other character pairs are 10, and the insertion/deletions costs for all characters are 5. The left figure is the DP table for string $A = \mathtt{abcdefghijklmnoprrr}$ of length $19 > 16 = (j^\star - 1)(c - 2)$ and string $B = \mathtt{uvxxx}$ of length 6. The right figure is the DP table for string A and string $B' = \mathtt{uvwxx}$ which is obtained by replacing $\mathtt{x}$ at position $j^\star = 3$ of B with $\mathtt{w}$. The rightmost column of each DP table shows $DR[i,p].U$ and $DR^\star[i,p].U$ for all columns $p \geq j^\star = 3$, respectively. Observe that the 5's and 4's at rows 1 to 16 of DR are shifted down by one row in $DR^\star$. Also, note that the vertical difference values of DR and $DR^\star$ will stay the same no matter how many $\mathtt{x}$'s we append to the right end of B. Hence, $O(j^\star cn)$ entries need to be updated when transforming DR to $DR^\star$.

set the substitution costs for all the other character pairs to c, and the insertion/deletion costs for all the characters to $c/2$. See also Fig. 4 which shows a concrete example.

The key observation on this instance is the following: Whenever there is a match of characters with some cost less than $c/2$, a "disturbing" value that breaks the usual value difference $c/2$ is introduced. To see this more formally, let us consider an arbitrary fixed j with $1 \leq j \leq j^\star$. Since $c/2 - k < c/2$, replacing $a_{(j-1)(c-2)+2k-1}$ with b_j always gives the best score at the $((j-1)(c-2)+2k-1, j)$ entry of the DP table for A and B. Also, for any $1 \leq k < c/2 - 1$, clearly $\delta(a_{(j-1)(c-2)+2k-1}, b_j) = c/2 - k > c/2 - k - 1 = \delta(a_{(j-1)(c-2)+2k+1}, b_j)$. Hence, the score at the $((j-1)(c-2)+2k+1, j)$ entry is not dominated by the score at the $((j-1)(c-2)+2k-1, j)$ entry. Therefore, column $j^\star$ of the DP table for A and B contains $(j^\star - 1)(c/2 - 1)$ different disturbed values. Moreover, by the definition of the cost function δ, each of the $(j^\star - 1)(c/2 - 1)$ different disturbed values propagates to the same row of the remaining $n - j^\star$ columns, increasing by c at each column onwards. Consequently, we have that for any $j^\star \leq p \leq n$,

- $DR[i,p].U = c$ for any even row i with $2 \leq i \leq (j^\star - 1)(c-2)$, and
- $DR[i,p].U = c - 1$ for any odd row $i = (j-1)(c-2) + 2k - 1$ with $1 \leq j \leq j^\star$ and $1 < k < c/2$.

Now, we replace the $j^\star$th character $b_{j^\star}$ of string B with b' which satisfies $\delta(a_i, b') = 1$ for all $1 \leq i \leq (j^\star - 1)(c-2)$. This shifts the disturbed value differences down just by one row at column $j^\star$ in the DP table, and the usual value difference $c/2$ down just by one row at column $j^\star$ in the DP table as well. Consequently, we have that for any $j^\star \leq p \leq n$,

- $DR^\star[i,p].U = c - 1$ for any even row $i = (j-1)(c-2) + 2k$ with $1 \leq j \leq j^\star$ and $1 < k < c/2$, and
- $DR^\star[i,p].U = c$ for any odd row i with $3 \leq i \leq (j^\star - 1)(c-2) - 1$.

Since $j^\star = o(n)$, $n - j^\star = O(n)$. Thus, there are $O(j^\star cn)$ entries of which the values differ between DR and $DR^\star$. This completes the proof. $\square$

5. Experiments

We have performed experiments with two problems: incremental/decremental edit distance computation and interactive edit distance computation. For the sake of observation of speed and memory usage of algorithms on various settings, we use random and corpora data for variable text length, alphabet size, and edit operation costs. All the experiments were conducted on a CentOS Linux desktop computer with 2.80 GHz Intel(R) Xeon CPU processor and 48 GB memory.

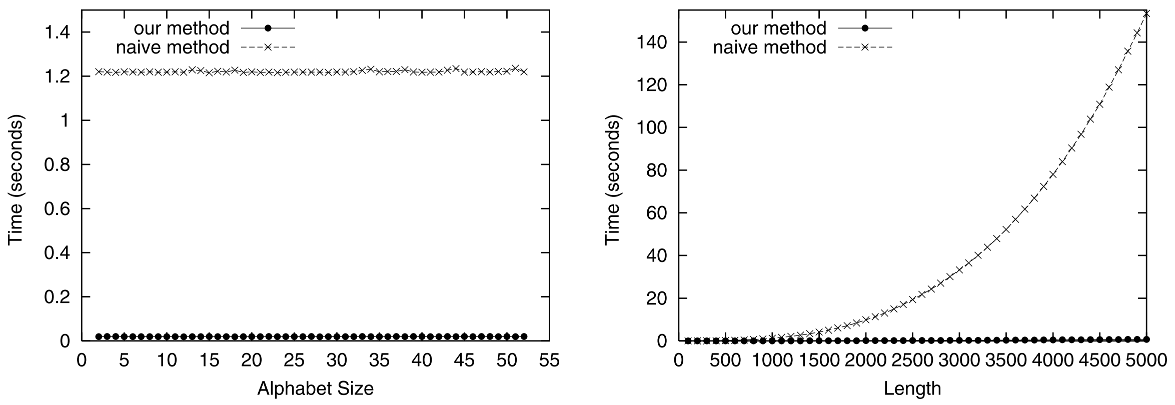

Fig. 5. Running times of our left incremental/decremental algorithm and the naive method on random text with; (left) fixed length 1000 and variable alphabet sizes from 2 to 52, (right) fixed alphabet size 26 and variable lengths from 100 to 5000.

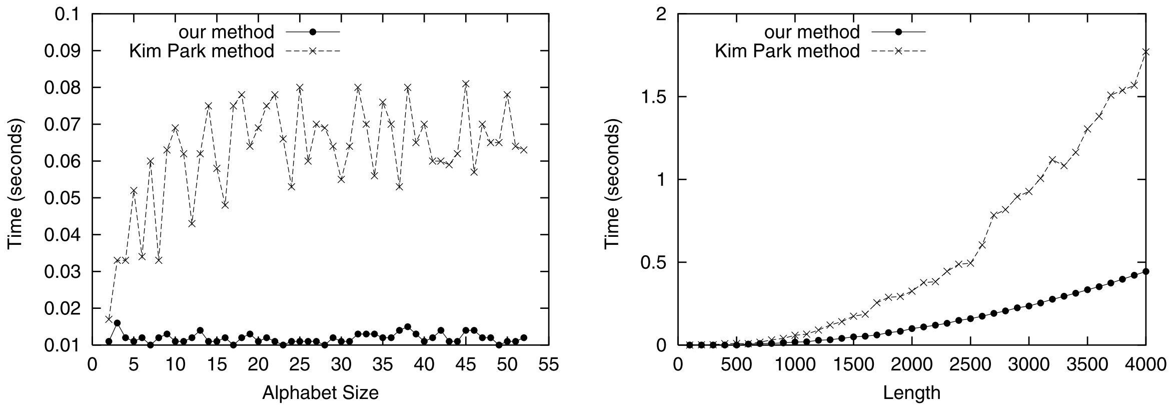

Fig. 6. Running times of our left incremental/decremental algorithm and the Kim and Park algorithm on random text with; (left) fixed length 1000 and variable alphabet sizes from 2 to 52, (right) fixed alphabet size 26 and variable lengths from 100 to 4000.

5.1. *Left incremental/decremental edit distance*

Right incremental/decremental computation can be handled in the optimal $\Theta(m)$ time even with the basic dynamic programming algorithm. Hence our tests in this section concentrate on the more interesting case of left incremental/decremental computation.

5.1.1. *Random data*

We performed experiments to compare running times of our algorithm with the basic (naive) dynamic programming method and the Kim–Park algorithm [6] (KP) on random text, varying the alphabet size and string length as parameters. Ours and KP 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 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 KP under the unit cost function on random text with; (left) fixed length 1000 and variable alphabet sizes from 2 to 52, (right) fixed alphabet size 26 and variable lengths from 100 to 4000. Our method clearly outperforms KP. We note that KP exhibits variance, probably due to the poor locality of its memory access patterns.

5.1.2. *Corpora data*

In this section, we show experimental results on real world texts 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 [1]. The tests concentrated on weighted cost functions and so KP was excluded.

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

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

Length	Proposal	Naive
1000	0.012	1.221
2000	0.096	9.820
3000	0.246	32.925
4000	0.457	77.936
5000	0.727	151.992

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

Length	Proposal	Naive
1000	0.012	1.220
2000	0.095	9.808
3000	0.240	32.928
4000	0.449	77.948
5000	0.717	152.328

Table 3
Cost function for the E. coli data.

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

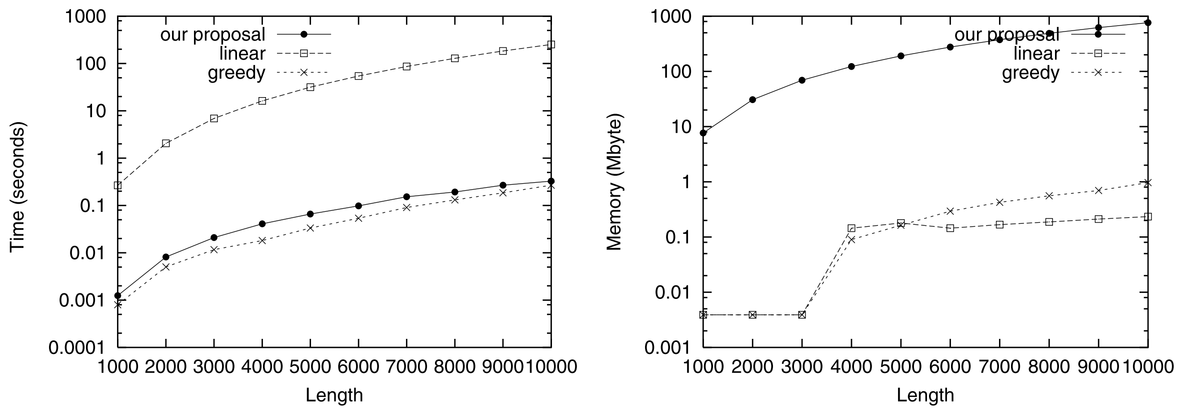

Fig. 7. Running times (left) and memory usages (right) for interactive edit distance computation. Randomly generated text with alphabet size 26, unit costs, 5% of modifications and length varying from 1000 to 10 000. The y-axes use logarithmic scale.

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 4 638 690. 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.

5.2. Interactive edit distance

Next, we performed some experiments to investigate the performance of out algorithm for the interactive edit distance problem on randomly generated text varying the length and the number of performed modification operations as parameters. Both the types (insertion, deletion, substitution) and positions of the performed modifications were selected randomly.

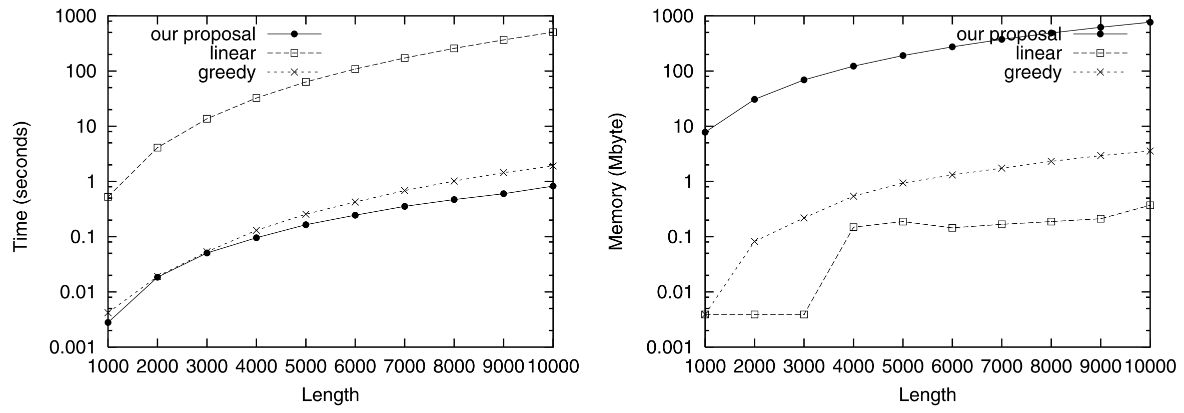

Fig. 8. Running times (left) and memory usages (right) for interactive edit distance computation. Randomly generated text with alphabet size 26, unit costs, 10% of modifications and length varying from 1000 to 10 000. The y-axes use logarithmic scale.

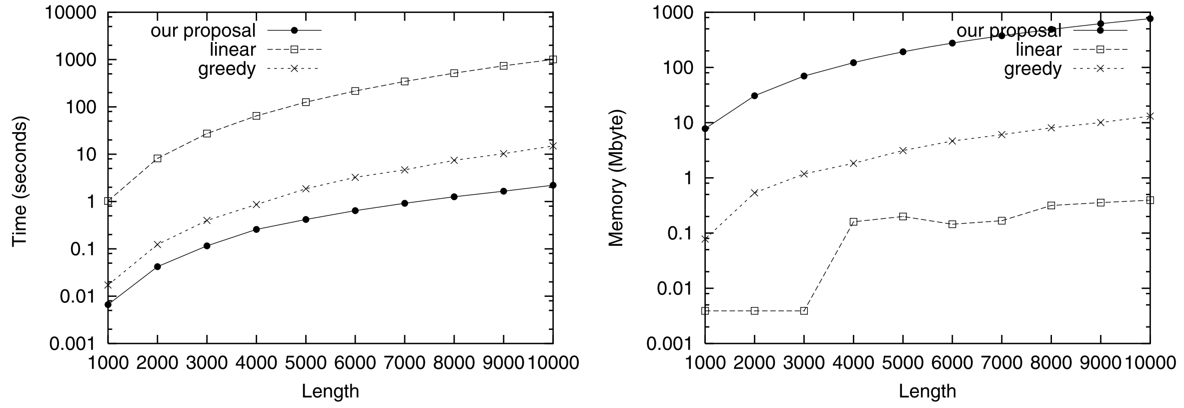

Fig. 9. Running times (left) and memory usages (right) for interactive edit distance computation. Randomly generated text with alphabet size 26, unit costs, 20% of modifications and length varying from 1000 to 10 000. The y-axes use logarithmic scale.

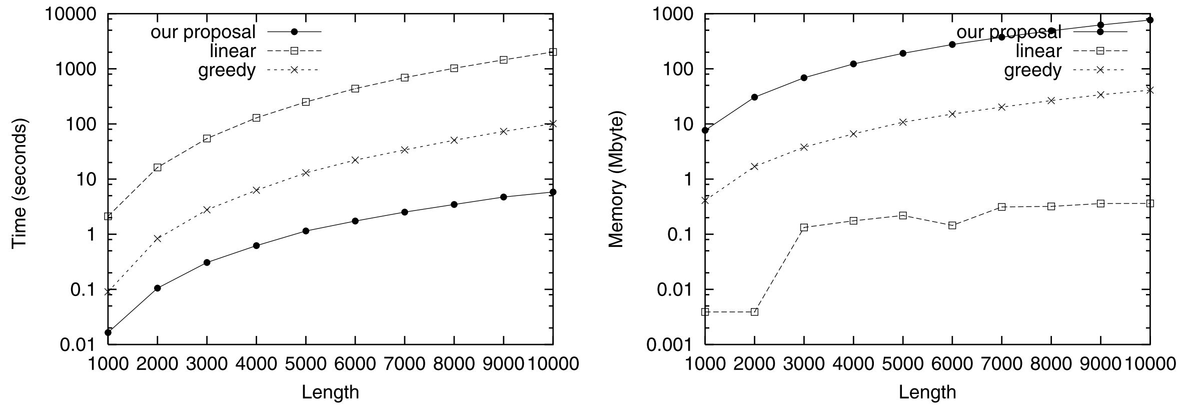

Fig. 10. Running times (left) and memory usages (right) for interactive edit distance computation. Randomly generated text with alphabet size 26, unit costs, 40% of modifications and length varying from 1000 to 10 000. The y-axes use logarithmic scale.

We compared our algorithm against the best existing alternatives known to us. In terms of space, we included Hirschberg's [3] space-efficient variant of dynamic programming that uses $O(m)$ space and $O(mn)$ time. In terms of time, we included a variant of a weighted greedy algorithm [17] that is similar to the implementation of UNIX diff [11].

Figs. 7–10 show the running times and memory usages of our algorithm, linear and greedy methods for randomly generated texts of variable length from 1000 to 10 000 with fixed alphabet size 26. The ratio of the number of performed modifications to the length of the text is 5% in Fig. 7, 10% in Fig. 8, 20% in Fig. 9 and 40% in Fig. 10. The greedy method is, as expected, fast when the number of performed modifications is relatively small (and hence the compared strings are very

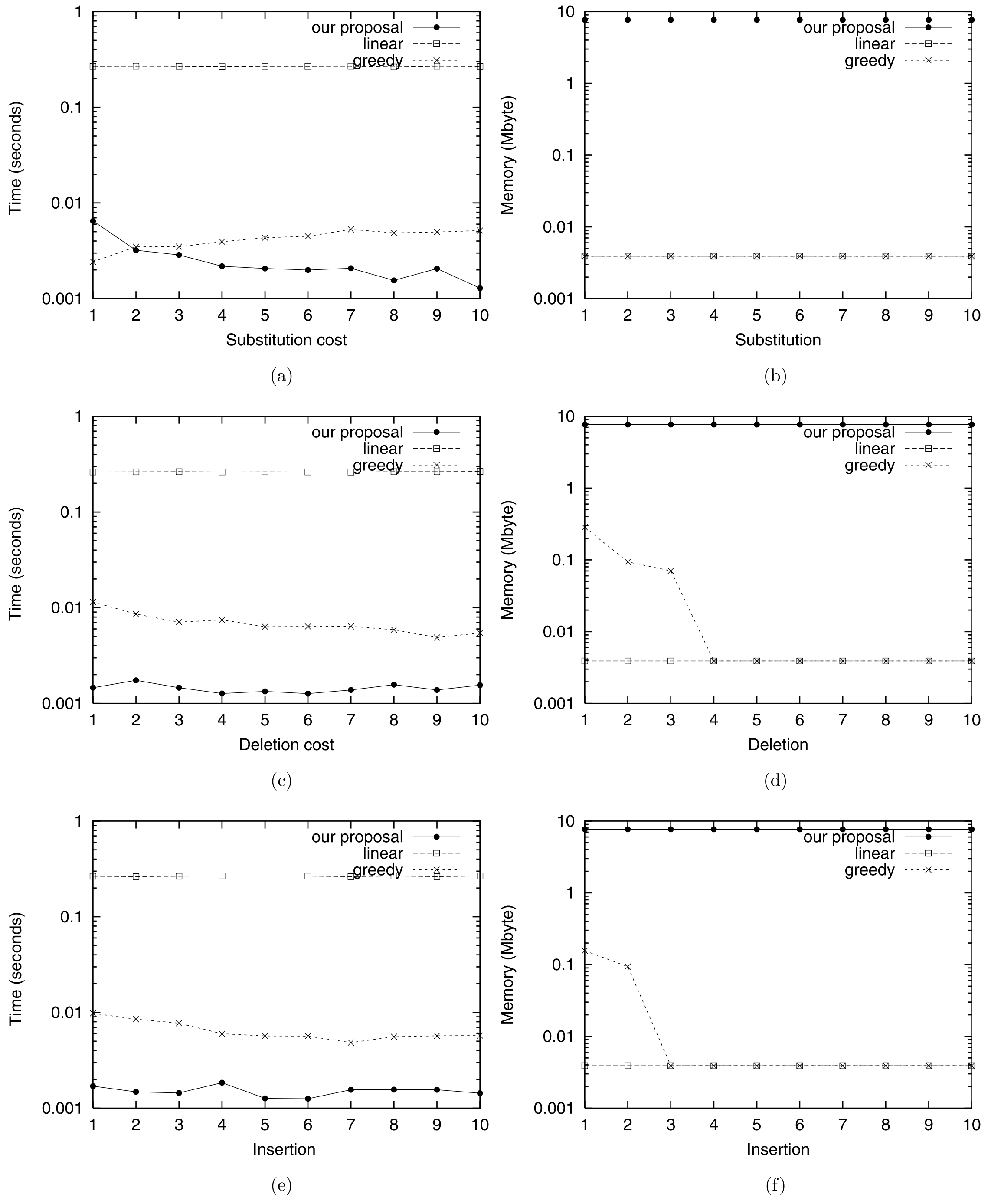

Fig. 11. Running times (a), (c), (e) and memory usages (b), (d), (f) for interactive edit distance computation. Randomly generated text with alphabet size 26, text length 1000 and 50 modifications; (a), (b) fixed insertion and deletion costs 10 and varying substitution cost from 1 to 10, (c), (d) fixed insertion and substitution costs 10 and varying deletion cost from 1 to 10, (e), (f) fixed deletion and substitution costs 10 and varying insertion cost from 1 to 10. The y-axes use logarithmic scale.

similar). Our algorithm becomes faster when the proportion of operations exceeds 10%. For example with 40% operations and text length 2000, our method is already over 10 times faster than the greedy method.

To see how the weights affect to the performance of our algorithm and others, we conducted experiments varying insertion, deletion, and substitution costs. Figs. 11 and 12 show the running times and memory usages of our algorithm, linear and greedy methods for variable insertion, deletion, and substitution costs from 1 to 10. These experiments use randomly generated texts of fixed length 1000 with alphabet size 26. Fig. 11 shows results with 50 modifications and Fig. 12

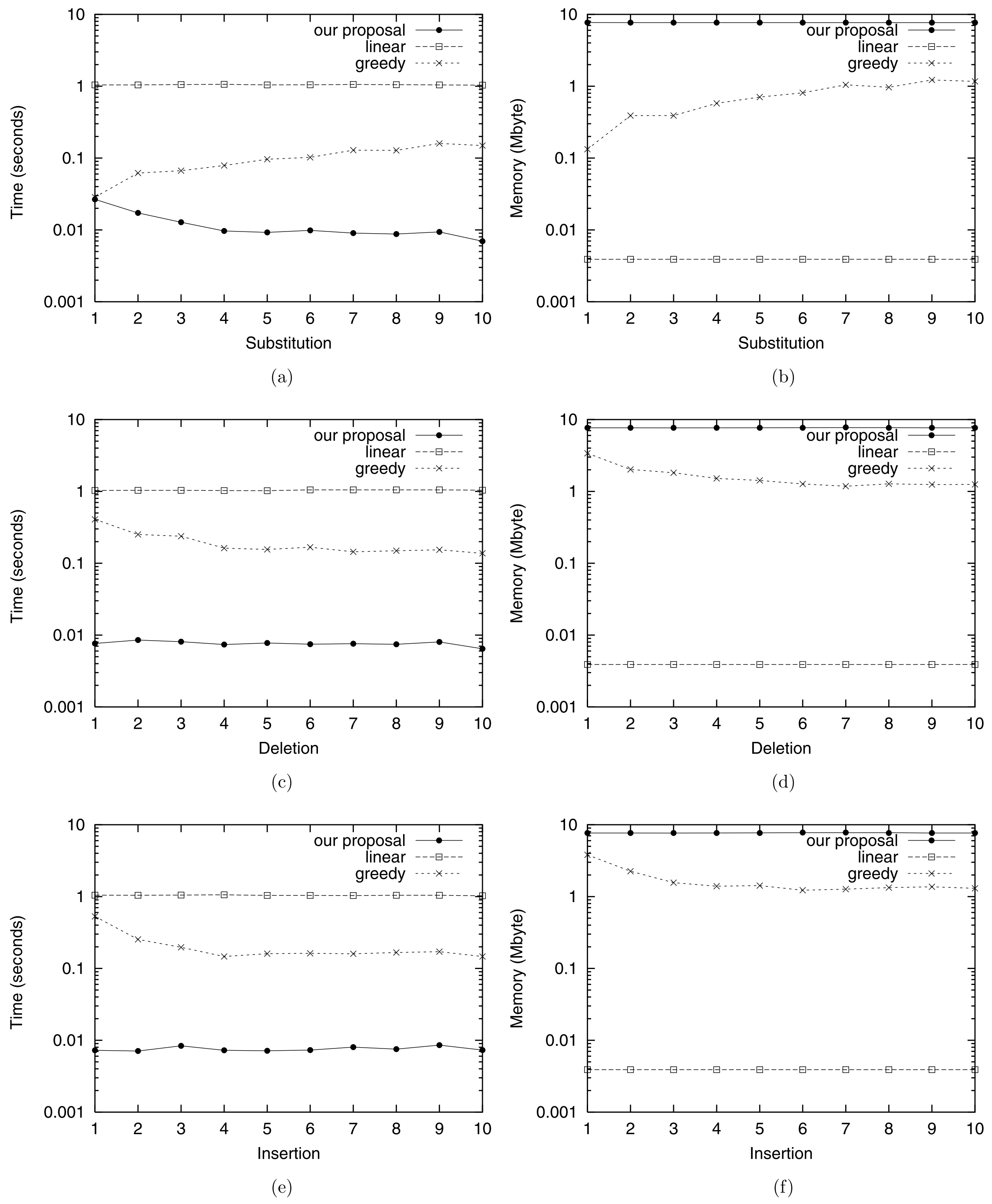

Fig. 12. Running times (a), (c), (e) and memory usages (b), (d), (f) for interactive edit distance computation. Random text with alphabet size 26, text length 1000 and 200 modifications; (a), (b) fixed insertion and deletion costs 10 and varying substitution cost from 1 to 10, (c), (d) fixed insertion and substitution costs 10 and varying deletion cost from 1 to 10, (e), (f) fixed deletion and substitution costs 10 and varying insertion cost from 1 to 10. The y-axes use logarithmic scale.

shows results with 200 modifications. Both the types (insertion, deletion, substitution) and positions of the performed modifications were selected randomly. In both Figs. 11 and 12, (a) and (b) show the results for varying substitution cost from 1 to 10, (c) and (d) show the results for varying deletion cost from 1 to 10, and (e) and (f) show the results for varying insertion cost from 1 to 10. In each case the other costs were fixed to be 10. Our algorithm was the fastest for almost all values of weights, with both 50 and 200 modifications. As expected, the memory usages of our algorithm are independent of the weights, but our algorithm uses much more space than the others.

6. Conclusions and further work

This paper proposed a simple but general algorithm for the problem of interactive edit distance computation, where we wish to update the edit distance information between two strings after one of them is modified at an arbitrary position $j^\star$. The algorithm takes $O(\min\{rc(m+n), mn\})$ time per modification, where m and n are the lengths of the two strings, $r = \min\{j^\star, n - j^\star + 1\}$ and c denotes the highest weight in the cost function.

Our algorithm solves the more limited incremental/decremental edit distance computation under a general weighted cost function in $O(\min\{c(m+n), mn\})$ time per added/deleted character. This leads to an $O(m+n)$-time solution under constant weights.

We also showed that a related algorithm of Kim and Park [6] does not extend well to the case of general weights; it might require even exponential time if an edit cost could be exponential w.r.t. the input string length.

We remark that our algorithm makes asymptotically minimum amount of work to maintain the difference representation of the dynamic programming table. Hence, a further improvement would require either a more efficient encoding of the dynamic programming table, or relaxing how much edit distance information to be maintained between the input strings.

Our algorithm was experimentally found to be faster than the main existing alternatives in many significant settings. A future work is to properly analyze the expected time complexity of our algorithm. The experimental results indicated that it can be much better than $O(mn)$ but not as good as $O(m+n)$ in the interactive edit distance computation setting.

References

[1] R. Arnold, T. Bell, A corpus for the evaluation of lossless compression algorithms, in: Proc. DCC'97, 1997, pp. 201–210, http://corpus.canterbury.ac.nz/.
[2] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[3] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Commun. ACM 18 (6) (1975) 341–343.
[4] H. Hyyrö, K. Narisawa, S. Inenaga, Dynamic edit distance table under a general weighted cost function, in: Proc. SOFSEM 2010, 2010, pp. 515–527.
[5] N.C. Jones, P.A. Pevzner, An Introduction to Bioinformatics Algorithms, MIT Press, 2004.
[6] S.-R. Kim, K. Park, A dynamic edit distance table, J. Discrete Algorithms 2 (2004) 302–312.
[7] S. Kurtz, Approximate string searching under weighted edit distance, in: Proc. 3rd South American Workshop on String Processing, WSP'96, 1996, pp. 156–170.
[8] G.M. Landau, Can dist tables be merged in linear time – an open problem, in: Proc. PSC 2006, 2006, p. 1.
[9] G.M. Landau, E.W. Myers, J.P. Schmidt, Incremental string comparison, SIAM J. Comput. 27 (2) (1998) 557–582.
[10] W.J. Masek, M. Paterson, A faster algorithm computing string edit distances, J. Comput. Syst. Sci. 20 (1) (1980) 18–31.
[11] W. Miller, E.W. Myers, A file comparison program, Softw. Pract. Exp. 15 (11) (1985) 1025–1040.
[12] Y. Sakai, An almost quadratic time algorithm for sparse spliced alignment, Theory Comput. Syst. 48 (1) (2011) 189–210.
[13] J.P. Schmidt, All highest scoring paths in weighted grid graphs and their application in finding all approximate repeats in strings, SIAM J. Comput. 27 (4) (1998) 972–992.
[14] A. Tiskin, Longest common subsequences in permutations and maximum cliques in circle graphs, in: Proc. CPM 2006, 2006, pp. 270–281.
[15] A. Tiskin, Semi-local string comparison: algorithmic techniques and applications, CoRR, arXiv:0707.3619, http://arxiv.org/abs/0707.3619.
[16] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.
[17] Z. Zhang, S. Schwartz, L. Wagner, W. Miller, A greedy algorithm for aligning DNA sequences, J. Comput. Biol. 7 (1–2) (2000) 203–214.