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

ARTICLE INFO

Article history:
Available online 27 May 2015

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

ABSTRACT

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/

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 $1 \leq k \leq \max\{m, n\}$. 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 $dist(A, B'')$