View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

ScienceDirect

Journal of Discrete Algorithms 5 (2007) 380–392

JOURNAL OF DISCRETE ALGORITHMS

www.elsevier.com/locate/jda

Edit distance with move operations

Dana Shapira *, James A. Storer

Computer Science Department, Brandeis University, Waltham, MA, USA

Received 22 December 2004; accepted 10 January 2005

Available online 3 May 2006

Abstract

The traditional edit-distance problem is to find the minimum number of insert-character and delete-character (and sometimes change character) operations required to transform one string into another. Here we consider the more general problem of a string represented by a singly linked list (one character per node) and being able to apply these operations to the pointer associated with a vertex as well as the character associated with the vertex. That is, in O(1) time, not only can characters be inserted or deleted, but substrings can be moved or deleted. We limit our attention to the ability to move substrings and leave substring deletions for future research. Note that O(1) time substring move operation implies O(1) substring exchange operation as well, a form of transformation that has been of interest in molecular biology. We show that this problem is NP-complete, and that a "recursive" sequence of moves can be simulated with at most a constant factor increase by a non-recursive sequence. Although a greedy algorithm is known to have poor (a polynomial factor) worst case performance, we present a polynomial time greedy algorithm for non-recursive moves which on a subclass of instances of a problem of size n achieves an approximation factor to optimal of at most $O(\log n)$. The development of this greedy algorithm shows how to reduce moves of substrings to moves of characters, and how to convert moves of characters to only inserts and deletes of characters.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Edit-distance; NP-complete; Bin packing problem; Longest common substring (LCS)

1. Introduction

The traditional edit-distance problem is to find the minimum number of insert-character and delete-character operations required to transform a string S of length n to a string T of length m. Sometimes the costs of inserts and deletes may differ, and change-character operations may have a different cost from a delete plus an insert. Here we restrict our attention to just the insert-character, delete-character, and move operations where all have the same unit cost, although we believe that much of what we present can be generalized to non-uniform costs.

It is well known how to solve the edit distance problem in $O(n \cdot m)$ using dynamic-programming (see for example the book of Storer [16] or Gusfield [8]) for a presentation of the algorithm and references). If the whole matrix is kept for trace back to find the optimal alignment, the space complexity is $O(n \cdot m)$, too. If only the value of the edit distance is needed, only one row of the matrix is necessary, and the space complexity is $O(m)$. Using the "Four Russians" algorithm, slight improvements to the $O(n \cdot m)$ time are given in the paper of Masek and Paterson [12].

* Corresponding author.
E-mail addresses: shapird@cs.brandeis.edu (D. Shapira), storer@cs.brandeis.edu (J.A. Storer).

1570-8667/$ – see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.jda.2005.01.010

In addition to the insert character and delete character operations, we allow move operations that transfer a substring from one location in S to another at a constant cost. One way to model the move operation is by viewing strings as singly-linked lists (one character per vertex), and allowing operations to apply not only to the character associated with a vertex but also to the pointer associated with it. To define the problem, we can assume that special characters # and $ are first added to S to form the string #$S$$, and the problem is to transform #$S$$ to #$T$$ with the stipulation that # and $ cannot be involved in any operation, although #'s pointer might be changed ($'s pointer is always *nil*). That is, # defines the list head, the process must produce a list that goes from # to $, and the characters stored at the vertices traversed are the transformed string, which must be equal to T (all vertices unreachable from # after the transformation is complete are considered deleted). For simplicity, we assume that all operations (insert-character, insert-pointer, delete-character, and delete-pointer) have the same unit cost. In terms of what can be done in O(1) time, what has been gained with the addition of the insert-pointer and delete pointer operations is the ability to:

1. Move a substring in O(1) time.
2. Delete a substring in O(1) time.

We limit our attention to the ability to move substrings and leave substring deletions for future research. Note that O(1) time substring move operations imply O(1) substring exchange operations as well, a form of transformation that has been of interest in molecular biology. Moves can perform transformations in O(1) time that could not be done in O(1) time in the standard edit distance model. For example, let $S = a^n b^m c^n d^m e^n$ and $T = a^n d^m c^n b^m e^n$, where $m \ll n$ but m is not a constant. The usual edit distance between S and T is O(m), as we would like to swap every b with every d. Using the new model the edit distance is reduced to O(1), by changing O(1) pointers.

The main results presented here are:

- The problem of edit distance with moves is NP-complete.
- A "recursive" sequence of moves can be simulated with at most a constant factor increase by a non-recursive sequence.
- A polynomial time greedy algorithm for non-recursive moves which on a subclass of instances of a problem of size n achieves an approximation factor to optimal at most O($\log n$).

Our development of this greedy algorithm shows how to rename substrings of the given strings into *meta-characters* and how to reduce moves of substrings to moves of these meta-characters, and how to convert moves of the meta-characters to only inserts and deletes of characters.

1.1. Previous work

Kececioglu and Sankoff [10] and Bafna and Pevzner [1] consider the reversal model, where the two given strings have no repeated characters, and which takes a substring of unrestricted size and replaces it by its reverse in one operation. Move operations can be simulated with O(1) reversal operations. For example, let $S = ABCD$. We wish to transform S to $ACBD$. Instead of moving C to the left of B, we can reverse BC and then reverse each of B and C separately. However, a reversal cannot in general be simulated by O(1) moves (so the reversal model is more powerful than the move model).

Muthukrishnan and Sahinalp [14] consider approximate nearest neighbors and sequence comparison with "block operations", and without "recursion"; related experiments are presented in Muthukrishnan and Sahinalp [15]. They employ an approximate distance preserving embedding of strings into vector spaces based on a hierarchical parsing for an algorithm that achieves an approximation factor of O($\log n \log^* n$) ($\log^* n$ is the number of times log function is applied to n to produce a constant). With block operations, they include not only moves and deletes, but also copies and reversals. The presence of copy and reversal operations changes the problem greatly. Not only are reversals more powerful than moves, but copies allow one to do in O(1) cost something that is not possible in O(1) time under normal assumptions about the manipulation of lists. They show NP-completeness and give a close to log factor approximation algorithm for related problems, but their construction is much more complex than presented here and does not seem to directly apply to this more simple model of just deletes and moves.

Cormode and Muthukrishnan [5] consider the same problem of moves only as considered here, that is, the standard edit distance problem augmented with the substring move operation. They present a $O(n \log n)$ time algorithm that yields an approximation factor of $O(\log n \log^* n)$. Their result employs an approximately distance preserving embedding of strings into the L_1 vector space using a simplified parsing technique they call *Edit Sensitive Parsing*. They state as an open problem whether their $O(\log n \log^* n)$ factor approximation can be improved.

Cormode, Paterson, Sahinalp and Vishkin consider in [4] the problem of minimizing the communication involved in the exchange of similar documents. They minimize the total number of bits they exchange by measuring the distance between these documents. They consider the Hamming distance, the Levenshtein distance and introduce a new one called the *LZ distance*. Their LZ metric considers substring copying and substring deletions together with the usual single character operations. The main difference of their approach and ours is that they are interested in the minimum number of operations required to produce one string from another, starting from an empty string by allowing substring copies from the source string. The construction of the new string is done by character insertion, character exchange, character deletion, and substring deletion of a repeated substring, and substring copies. We perform the operations on a given string in order to attain the other using only substring moves, character insertions and character deletions.

Ergun and Sahinalp [6] study the string similarity problem and provide a constant factor approximation to the edit distance problem with character insertions, arbitrary block deletions and copies. Their results rely critically on the presence of the copy operation. They consider two frameworks: external transformations and internal transformations. The external transformation is to construct one string from an initially empty string where the source of all operations is the other string. The internal transformation constructs one string by iteratively applying edit operations on the other string (the work to be presented here corresponds to internal transformations, since moves are not allowed in their external transformation framework). Although one can use moves in the internal transformations their construction relies on copies, character insertions, and a specific delete operation (used as a final operation).

Chrobak et al. [3] consider the *Minimum Common String Partition* (*MCSP*) problem, that receives two input strings and tries to minimize the partition of the strings into the same collection of substrings. They refer to several versions of this problem by limiting the number of times each character can occur in both input strings, and study the approximation of a greedy algorithm for MCSP that at each step extracts a longest common substring from the given strings. In the case of 2-MCSP, where each character can occur at most twice in each input string the approximation of 3 is shown. For 4-MCSP the approximation ratio of the greedy algorithm is at least $\Omega(\log n)$, and for the general problem they present an approximation ratio between $\Omega(n^{0.43})$ and $O(n^{0.69})$.

Bafna and Pevzner [2] refer to moves as *transpositions*; this is motivated by observing that if $S = uvwx$ is transformed by moving substring w to $T = uwvx$, then the effect is to have exchanged the two substrings w and v. For the case that S is a permutation of the integers 1 through n, they give a 1.5 approximation algorithm for the minimum number of transpositions needed to transform S to a different permutation T. Although related to the problem considered here, the restriction that all characters are distinct greatly changes the problem.

Lopresti and Tompkins [11] consider a model in which two strings S and T are compared by extracting collections of substrings and placing them into the corresponding order. Tichy [17] looks for a minimal covering set of T with respect to a source S such that every character that also appears in S is included in exactly one substring move; unlike our model, one substring move can be used to cover more than one substring in T. Thus, S is constructed by using the copy operation of substrings of the minimal covering set. Hannenhalli [9] studies the minimum number of rearrangements events required to transform one genome into another; a particular kind of rearrangement called a *translocation* is considered, where a prefix or suffix of one chromosome is swapped with the prefix or suffix of the other chromosome, and a polynomial algorithm is presented which computes the shortest sequence of translocations transforming one genome into another.

1.2. Formalism

We now give a formal description of the three operations *insert*, *delete*, and *move*. Let Σ denote a finite alphabet. For a character $\sigma \in \Sigma$, a string $S = s_1 \cdots s_n$ and a position $1 \leqslant p \leqslant n$, the operation *insert*$(\sigma, p)$ inserts the character σ to the pth position of S. After performing this operation, S becomes $s_1 \cdots s_{p-1} \sigma s_p \cdots s_n$. The operation *delete*$(p)$ deletes the character which occurs at the pth position of S, and returns the character which was deleted, i.e., S becomes $s_1 \cdots s_{p-1} s_{p+1} \cdots s_n$ and it returns the character s_p.

Consider the strings $S = abcdef$ and $T = adebcf$. Transforming S into T should move either the string bc from position 2 to position 4 or move de from position 4 to position 2. To eliminate ambiguity we define the move operation as the second choice. Given two distinct positions $1 \leqslant p_1 < p_2 \leqslant n$ and a length $1 \leqslant \ell \leqslant n - p_1 + 1$, $move(\ell, p_1, p_2)$ moves the string at position p_2 of length ℓ to position p_1. After performing the move operation, S becomes $s_1 \cdots s_{p_1-1} s_{p_2} \cdots s_{p_2+\ell-1} s_{p_1} \cdots s_{p_2-1} s_{p_2+\ell} \cdots s_n$. Alternatively, we sometimes write $move(str, p_1, p_2)$, where str specifies the string which is moved, rather than its length.

In the following section we show that computing the optimal edit-distance with moves is NP-complete (substring deletions are not needed for this construction, move-string operations suffice to imply NP-completeness). In Section 3 we simplify the problem of finding an approximation algorithm by showing that the elimination of recursive moves cannot change the edit distance by more than a constant factor. In Section 4 we present a greedy algorithm that works by repeatedly replacing a given number of copies of a longest common substring of S and T by a new character, and then Section 5 shows how a reduction to the standard edit distance algorithm can be used. Although a greedy algorithm is known to have poor (a polynomial factor) worst case performance [3], Sections 6 and 7 show that on a subclass of problems this greedy algorithm gives a log factor worst-case approximation to optimal. Section 8 mentions some areas of future research.

2. NP-completeness of edit distance with moves

Theorem 1. *Given two strings S and T and an integer $m \in N$, using only the three unit-cost operations insert character, delete character, and move substring, it is NP-complete to determine if S can be converted to T with cost $\leqslant m$ using the above operations.*

Proof. Since a non-deterministic algorithm need only guess the operations and check in polynomial time that S is converted to T with cost $\leqslant m$, the problem is in NP. □

We employ a transformation from the bin-packing problem, which is:

Given a bin capacity B, a finite set of integers $X = \{x_1, \ldots, x_n\}$ where $x_i \leqslant B$, and a positive integer k, the *bin packing* problem is to determine if there is a partition of X into disjoint sets $X_1, \ldots, X_k$, such that the sum of the items in each X_i is exactly B. The *bin-packing* problem is NP-complete in the strong sense (e.g., see Garey and Johnson [7]); that is, even if numbers in the statement of the problem instance are encoded in unary notion (a string of n 1's representing the number n), it is still a NP-complete problem.

Given an instance B, $X = \{x_1, \ldots, x_n\}$, k of the bin-packing problem, let a, #, and \$ denote three distinct characters, and let:

$$S = \$^k \prod_{i=1}^{n} \#a^{x_i}$$

$$T = \left(a^B \$\right)^k \#^n$$

$$m = n$$

Since the bin-packing problem is NP-complete in the strong sense, we can assume that the lengths of S and T are polynomial in the statement of the bin-packing problem.

Claim. *S can be converted to T with a cost $\leqslant m$ if and only if there is a partition of X into disjoint sets $X_1, \ldots, X_k$ such that the sum of the items in each X_i is B.*

For the *if* portion of the proof, suppose that there is a partition of X into disjoint sets $X_1, \ldots, X_k$, such that the sum of the items in each X_i is B. Then S can be transformed to T by, for each item $x_i \in X_j$, $1 \leqslant j \leqslant k$, moving the corresponding a's to between the \$ of the corresponding bin. That is, perform $move(a^{x_i}, k + \sum_{\ell=1}^{i-1} (x_\ell + 1), (j-1)B)$, for a total of $\sum_{j=1}^{k} \sum_{x_i \in X_j} 1 = \sum_{i=1}^{n} 1 = n$ operations.

For the *only if* portion of the proof, we use a sequence of lemmas to show that if the edit distance with moves between S and T is $\leqslant n$, then there is a bin packing in k bins of size B of the items of X.

Lemma 1. *The operations that were executed in order to convert S into T could be attained by using only move operations.*

Proof. The number of occurrences of each character in S is equal to the number of its occurrences in T. Therefore, if x is either \$ or # or a, then each $insert(x, p_1)$ operation will cause a $x \leftarrow delete(p_2)$ operation of the character x and vice versa. Thus every insert or delete operation could be performed by using the operation $move(x, p_2, p_1)$. □

We say that a pattern P is *involved* in some set of operations, A, if at least one of A's operations moves or deletes at least one of P's characters, or if one of its operations puts another string between P's characters.

Lemma 2. *Each occurrence of the string #a in S is involved in at least one operation during the conversion of S into T.*

Proof. Define $P = \#a$. The a occurs on the right side of the # sign in P. No a occurs on the right side of # in T. As P occurs in S, we must move or delete either the # sign or the a, or both during our conversion of S into T. □

We say that a subpattern P of S is *"straightened-up"* by some set of operations, if after using these operations, its characters occur in the same order as in T.

Lemma 3. *By performing one operation we cannot "straighten up" more than one occurrence of the pattern #a.*

Proof. Using Lemma 1 we can relate only to move operations. By moving any other string into the string $\#a$, the # still remains on the left side of the a. Thus, performing an operation on characters that do not involve the current occurrence of the string $\#a$, cannot straighten it up. In order to affect more than one occurrence of the string $\#a$ we should include characters from more than one occurrence. Suppose we performed $move(Q, p_1, p_2)$, for some pattern Q such that Q consists of more than one occurrence of the string $\#a$. We show that at least one more operation should be applied in order to straighten up these occurrences. As Q should include more than one occurrence of the string $\#a$, it should include the pattern $a\#$. If Q contains more than one # sign then Q contains the string $\#a^{x_i}\#$ for some $1 \leqslant i < n$. Thus, Q contains the string $\#a$ and by using Lemma 2, at least one other operation is needed in order to move the # sign to the right side of the a's. If the pattern $a\#a$ is a subpattern of Q, then at least one a is on the right side of the # sign, so again more than one operation is required. Thus Q is of the form $a^y\#$ for $1 \leqslant y \leqslant x_{i-1}$. As we performed $move(Q, p_1, p_2)$, the remaining # is still left on the left side of the a's of the other block, so still one operation did not straighten up the two occurrences of the string $\#a$. □

Lemma 4. *Each occurrence of the string #a in S is involved in exactly one move operation during the conversion of S into T, and these are the only operations that were executed.*

Proof. The pattern $\#a$ occurs n times in S. Using Lemma 2, we already know that each occurrence of the string $\#a$ must be involved in at least one operation. By Lemma 3 we know that by performing one operation we can "straighten up" a maximum of one occurrence of the string $\#a$. The string S was converted into T by performing at most n operations. Therefore, each occurrence of the string $\#a$ is involved in exactly one operation, and no other operations were executed. □

Conclusion 1. *No \$ sign was moved during the conversion of S into T.*

Lemma 5. *Given the two strings S and T. In order to convert S into T by using only n operations, we only moved the blocks a^{x_i} $\forall 1 \leqslant i \leqslant n$.*

Proof. Using Lemma 4, we already know that the string #a is involved in exactly one *move* operation, and only such operations were done in order to convert S into T.

Suppose we performed $move(Q, p_1, p_2)$ for some substring Q of S. ($\Longrightarrow \# \in Q$ or $a \in Q$ (or both).) If Q contains the pattern #a as its subpattern, using Lemma 2, this would follow at least another operation, thus performing more than one operation for this occurrence, which contradicts Lemma 4.

Suppose the pattern a# occurs in Q. As a and # are not adjacent in T, we should need at least another operation in order to move a \$ sign between them. This contradicts Conclusion 1.

Suppose $Q = \#$. Therefore, we have not touched the a's which occur right after this #. By Conclusion 1 the \$ signs were not moved. Thus, these a's remain on the right side of the \$ sign. This would require at least one more operation for this occurrence, which contradicts Lemma 4.

Suppose $Q = a^y$ and that $y \neq x_i, \forall 1 \leqslant i \leqslant n$. Since for every i, $1 \leqslant i \leqslant n$, $y < x_i \leqslant B$, then $x_i - y > 1$ a's precede or follow the pattern a^y in S. By Conclusion 1 the \$ signs were not moved. Thus, these a's remain on the right side of \$ sign. This would require at least one more operation for this occurrence, which contradicts Lemma 4.

$\Longrightarrow Q = a^{x_i}$ and we only performed the operation $move(a^{x_i}, p_1, p_2)$. Using Lemma 2 this was done for every $1 \leqslant i \leqslant n$. □

Lemma 6. *Let us assume that the edit distance of S and T is less than or equal to n. Then there is a bin packing in k bins of size B of the items in X.*

Proof. Using Lemma 5, the only operations executed are $move(a^{x_i}, p_1, p_2)$, $\forall 1 \leqslant i \leqslant n$. As the \$ signs occur on the left side of the a's in S, and there are B a's preceding each \$ in T, we should move B a's in front of each \$, by using only movements of x_i a's. This corresponds to a packing of these x_i's into that bin. □

3. Recursive moves

In this section we simplify the problem of finding an approximation algorithm by showing that the elimination of *recursive* moves cannot change the edit distance by more than a constant factor. For simplicity we refer only to move operations, which is justified in Section 6.

A sequence $A = a_1, a_2, \ldots, a_r$ of legal move operations produces a division, $\widehat{A}$, of the string S into blocks of characters. More formally, the first move operation in A of the form $move(i, j, \ell)$ defines a partition of $S = s_1 \cdots s_n$ into four (possibly empty) blocks: $[s_1 \cdots s_{i-1}]$, $[s_j \cdots s_{j+\ell-1}]$, $[s_i \cdots s_{j-1}]$ and $[s_{j+\ell} \cdots s_n]$.

Every other move operation of A refines this partition by adding at most 3 blocks (two blocks at the source location and one at its destination). Note that any subsequence of A defines a partition of S, and if $A_1 \subseteq A_2$ are two subsequences of A, then the partition of S, $\widehat{A_2}$, defined by A_2 is a refinement of the partition $\widehat{A_1}$, defined by S_1.

Definition. A sequence of move operations applied to a string is *recursive* if it contains a move operation which moves a substring whose characters do not occur consecutively in the original string, or moves a substring in between characters which have already been moved.

For example, if S is the string $abcde$ and the character b is moved to obtain the string $T = acdbe$, then moving the substring dbe or ac are both considered as recursive moves.

The following example shows us that performing recursive moves can reduce the cost of the edit-distance. Let $S = xababycdcdz$ and $T = xcddcyabbaz$. If we do not allow recursive moves the minimum cost of converting S into T is 5 operations ($Move(cd, 7, 2)$, $Move(d, 10, 4)$, $Move(c, 10, 5)$, $Move(y, 10, 6)$ and $Move(b, 10, 9)$). The indices are given according to the new positions after previous moves were done. By allowing recursive moves we can reduce the cost to 4 operations ($Move(b, 5, 4)$, $Move(d, 10, 9)$, $Move(cddc, 7, 2)$ and $Move(y, 10, 6)$).

Theorem 2. *If there is a recursive sequence of n moves that converts S into T, then there is a non-recursive sequence of no more than $3n$ moves that convert S to T.*

Proof. When recursive moves are allowed, the first move creates 4 blocks and every other move creates at most 3 more blocks. Thus, the number of blocks which result from a series R of n recursive moves is bounded by $3n + 1$.

Given the mapping which specifies the order of the corresponding blocks in T, one could easily replace R with a series R' of $3n+1$ (non-recursive) moves which scans the mapped blocks in T from left to right and for each scanned block in T moves the corresponding block in S to its correct position. □

A worst case example. We first show that the bound of 3 is tight, and then modify the construction for a finite alphabet. The following example builds two strings S and T of n phrases such that the non-recursive edit distance of S and T is three times the recursive edit distance of these strings. For simplicity, because each character will not occur more than once in both S and T, for this example we use the notation $move(b)$ for moving a substring b of S, without stating its source and destination locations. Let $S_1 = abcde$ and $T_1 = adbce$. Suppose S_1 is a substring of S, not aligned with the substring T_1 of T, and therefore must be moved. The recursive edit is 2 ($move(d)$ and $move(adbce)$) and the non-recursive edit is 4 ($move(a)$, $move(d)$, $move(bc)$ and $move(e)$).

Let $S_2 = ab_1b_2c_1c_2c_3de$ and $T_2 = adb_1c_2b_2c_1c_3e$, and suppose again S_2 must be moved in order to be aligned with T_2. Thus the recursive edit of S_2 and T_2 is 3 ($move(d)$, $move(c_2)$ and $move(adb_1c_2b_2c_1c_3e)$) and the non-recursive edit is 7 ($move(a)$, $move(d)$, $move(b_1)$, $move(c_2)$, $move(b_2c_1)$, $move(c_3)$ and $move(e)$).

Suppose we continue taking out a mid part of a block (thus, breaking a continuous string into three parts), and putting it into another block, and again breaking a continuous string. This way we add three blocks by move operation. If S_i should be moved again, these previous move operations turn into recursive ones, and cannot be performed by the non-recursive algorithm. Unfortunately, the non-recursive algorithm must move each block separately to its final location. Since every recursive operation adds 3 more blocks, the non-recursive edit distance increases by three. By generalizing this process, let S_n and T_n be the source and destination string when performing such moves n times. The recursive edit distance of S_n and T_n is $n+1$ (including the move of the entire block). We have $3n$ blocks in addition to the one block we have started with, which require $1+3n$ non-recursive moves. If n is unbounded, $\lim_{n\to\infty} \frac{1+3n}{1+n} = 3$. We still have to define S and T so that S_n must be moved in S in order to be aligned with T_n in T. As we would like the other parts of S not to be moved instead of the substring S_n, moving them must cost more than moving S_n. In order to achieve this we construct $2n$ strings, $\{S_n^1, S_n^2, \ldots, S_n^n\}$ and $\{T_n^1, \ldots, T_n^n\}$, of the form of S_n and T_n, respectively. The strings S_n^i and T_n^i are constructed from the same characters but their characters differ from any other pair S_n^j and T_n^j ($i \neq j$). Let $S = S_n^1 \cdot S_n^2 \cdots S_n^n$ and $T = T_n^n \cdot T_n^{n-1} \cdots T_n^1$. As the blocks in T occur in the reversed order of the blocks of S, at least $n-1$ blocks should be moved. The recursive edit distance of S and T is $(n-1)\cdot(n+1)+n$, as we should use n operations in each of the $n-1$ blocks before or after they have moved, and n more operations in the block which stands still. The non-recursive edit distance of S and T is $(n-1)\cdot(1+3n)+n$, as it should perform $1+3n$ operations for the $n-1$ blocks which are moved, and exactly the same operations of the recursive algorithm in the block that stays still. If n goes to infinity then we obtain the factor of three by $\lim_{n\to\infty} \frac{(n-1)\cdot(3n+1)+n}{(n-1)\cdot(n+1)+n} = 3$.

We now use Fibonacci codes to change the above construction to work for a binary alphabet. Fibonacci codes have the property that there are no adjacent 1's in each code word except at its end, so that 11 acts as a comma between consecutive code words. More formally, binary Fibonacci numbers are defined as follows: Let F_j be the jth Fibonacci number, $F_0 = 0$, $F_1 = 1$, and $F_j = F_{j-1} + F_{j-2}$ for $j > 1$. Any integer i can be represented by the binary string $I = I_1 I_2 \cdots I_r$, with I_j equal to either 0 or 1, where $i = \sum_{j=1}^{r} I_j F_{j+1}$. One can uniquely express any integer in this form so that there are no adjacent 1's in I, i.e., $I_j = 1$ implies that $I_{j-1} = 0$ for $j = 2, \ldots, r$. We then use the code that is obtained by replacing the trailing zeros in I by an additional 1, i.e., the code $\{11, 011, 0011, 1011, 00011, 10011, \ldots\}$. We now use the code $F_{n(i-1)+j}$ instead of the jth character of S_n^i in the definition above, and replace the same character in T_n^i by the same Fibonacci codeword. The factor of 3 between the non-recursive and recursive edit distance remains the same, since each move that crosses a Fibonacci codeword boundary, cannot include the suffix 11 of the codeword, thus preserving the same number of operations.

4. The greedy algorithm

In this section we present a polynomial time approximation algorithm for the minimum move edit-distance. It is a greedy method that reduces the two strings S and T to two other strings, so that we can perform the traditional edit distance algorithm on the new strings. Define $LCS(S, T)$ as the length of the longest common substring of the two strings S and T. For example: $LCS(abcd, edbc) = 2$, since bc is the longest common substring of S and T, and consists of 2 characters, but $LCS(abc, def) = 0$, since there is no common character of these two strings. The algorithm uses two procedures. The $ed(S, T)$ procedure computes the traditional edit distance of S and T by using

Stage 1: **while** ($|LCS(S,T)| > 1$) {
$P \leftarrow LCS(S,T)$
Let A be a new character, i.e., $A \notin \Sigma$.
Replace the same number of occurrences of P in S and in T by A.
$\Sigma \leftarrow \Sigma \cup \{A\}$
}
Stage 2: $d \leftarrow ed(S,T)$
Stage 3: $d \leftarrow check\_move(S,T)$
return d

Fig. 1. The greedy algorithm.

the dynamic programming method. The $check\_move(S,T)$ procedure checks whether we can reduce the edit-distance by using move operations instead of inserting and deleting the same character. The algorithm is given in Fig. 1.

In this section we explain the algorithm and in the following section we discuss the *check_move* procedure it uses. Consider the following example: let $\Sigma = \{a,b,c,d,e\}$, $S = cdeab$ and $T = abcde$. After the first stage of the greedy algorithm, $S' = AB$ and $T' = BA$, where $A = cde$ and $B = ab$. In the second stage, by performing the traditional edit distance on S' and T', we find that $d \leqslant 2$. The third stage does better by using *check_move* to determine that S can be converted to T by performing $A \leftarrow delete(1)$ and $insert(A,2)$ (which deletes and inserts the same character A), and therefore $d = 1$ since we can simply move the string cde to the end of S.

The greedy algorithm (Fig. 1) reduces the strings S and T to (possibly) shorter strings by replacing repeatedly a $LCS(S,T)$ by a new single character. For the worst-case performance bound that we will present, it will not matter whether the replacement step of the while loop replaces only one occurrence of $LCS(S,T)$ in S and T by A or replaces as many pairs as possible, so long as the total number of replacements of $LCS(S,T)$ by A in S is the same as the total number of replacements of $LCS(S,T)$ by A in T. It is perhaps natural to ask why we should obey the restriction that the number of replacements be equal. Let $greedy'$ denote an unrestricted version of the greedy algorithm that replaces the largest possible number of (non-overlapping) occurrences of $LCS(S,T)$ in S by A and the largest possible number of occurrences of $LCS(S,T)$ in T by A, even if the number of replacements made in S and T differ. Let opt denote the edit-distance returned by an optimal algorithm which allows move operations. Although $greedy'$ might perform better in some cases, the following lemma notes that in the worst case, $greedy'$ can perform very badly with respect to optimal.

The following example shows that For any $n > 0$, there are strings S and T of length n for which $\frac{greedy'}{opt} > n/2$. Let $\Sigma = \{a,b\}$. Let $S = (ab)^{2N}$ and let $T = (ab)^N (ba)^N$ strings of length $n = 4N$. The optimal edit distance is 2 ($insert(b, 2N+1)$, $b \leftarrow delete(4N)$), which is converted to a single move operation of the character b. By this version of the greedy algorithm S is reduced to the string AA and T is reduced to the string $A(ba)^N$, where $A = (ab)^N$. The edit distance is $2N+1$ which includes the operations: $A \leftarrow delete(2)$, $insert(b, 2N+2i-1)$ and $insert(a, 2N+2i)$ for i, $1 \leqslant i \leqslant N$. Now, $\frac{greedy'}{opt} = \frac{2N+1}{1} > 2N = \frac{n}{2}$.

In the above example, A occurs twice in T and only once in S, and the algorithm could not identify any similarity between A and $(ba)^{2n}$. The greedy algorithm, as presented in Fig. 1, overcomes this problem by replacing the same number of occurrences of the $LCS(S,T)$ in S and in T. However, it is known to have a polynomial factor worst case performance as well [3].

The greedy algorithm is based on the traditional edit distance algorithm. Therefore, it does not perform any recursive move, as every block participates in no more than one operation. However we have shown that by not allowing recursive moves we do not increase the number of move operations by more than a constant factor.

We now examine the running time of the greedy algorithm. The first stage can be done using a suffix trie in $O(\min(n,m) \cdot \max(n,m))$ processing time, where n and m are the length of the strings S and T, respectively. We construct the suffix trie, in $O(n+m)$ processing time, for the string $S\$T\#$, where \$ and # are two new symbols. We then traverse this suffix trie in post-order and label each vertex as to whether it has descendants in only S, in only T, or in both S and T (once you know this information for the children it is easily computed for the parent), to find the non-leaf vertex of lowest virtual depth that has both. This is repeated at most $\frac{\min(n,m)}{2}$ times, which happens when all the common substrings consist of 2 characters, and all characters participate in these common substrings. The second stage can be done in $O(n \cdot m)$ processing time, using the dynamic programming method. In the following section we

prove that the third stage can be done in $O(n+m)$ processing time. Therefore, the entire processing time of the greedy algorithm is $O(n \cdot m)$.

5. Identifying move operations

In the third stage of the greedy algorithm we are interested in identifying move operations which were done in the second stage. A move operation of a character σ is simply an insert and a delete operation of the same character σ. At first sight we might think that we cannot separate these two stages. It seems as if in every stage of the dynamic programming algorithm, after computing the cheapest operation of the current character, we must check if we could reduce the cost by combining it with an opposite operation of the same character and changing it to a move operation. We show that it is enough to identify move operations after computing the edit-distance, and we do not have to take it into account in the inner stages.

We use the following notation: Let P denote a way to convert a string S into a string T by using inserts and deletes. Let us denote by I_σ^P / D_σ^P the number of insertions/deletions of the character $\sigma \in \Sigma$ which were made when converting S into T using the path P. The edit-distance between the string S and T which is attained according to path P would be denoted by ed^P. The edit-distance between S and T, including move operations is denoted by edm^P, i.e., if P includes both $insert(\sigma)$ and $delete(\sigma)$ for any $\sigma \in \Sigma$, this would be calculated as one operation. If we are interested in the minimum edit-distance we use ed for the traditional edit distance and edm for the edit-distance with move operations.

Lemma 7. *For any two paths P and P' and $\sigma \in \Sigma$, $|I_\sigma^P - D_\sigma^P| = |I_\sigma^{P'} - D_\sigma^{P'}|$.*

Proof. Denote by n_σ^S and n_σ^T the number of appearances of a character σ in the strings S and T, respectively. For any path P which converts S into T $|I_\sigma^P - D_\sigma^P| = |n_\sigma^S - n_\sigma^T|$. □

Note that $\forall a, b \in \mathcal{N}$, $a + b = 2\min(a, b) + |a - b|$. Assuming that the cost of insert, delete and move operations are equal, we obtain:

$$ed^P = \sum_{\sigma\in\Sigma} \left(I_\sigma^P + D_\sigma^P\right) = \sum_{\sigma\in\Sigma} \left(2 \cdot \min\left(I_\sigma^P, D_\sigma^P\right) + \left|I_\sigma^P - D_\sigma^P\right|\right) \tag{1}$$

$$edm^P = \sum_{\sigma\in\Sigma} \left(I_\sigma^P + D_\sigma^P - \min\left(I_\sigma^P, D_\sigma^P\right)\right) \tag{2}$$

$$= \sum_{\sigma\in\Sigma} \left(\min\left(I_\sigma^P, D_\sigma^P\right) + \left|I_\sigma^P - D_\sigma^P\right|\right) \tag{3}$$

Lemma 8. *The minimal edit distance with move operations occurs in any optimal path of the traditional edit-distance.*

Proof. Suppose P and P' are two paths converting S into T, and that $ed^P < ed^{P'}$. By using Lemma 7 and Eq. (1) we find that

$$\sum_{\sigma\in\Sigma} \left(2 \cdot \min\left(I_\sigma^P, D_\sigma^P\right)\right) < \sum_{\sigma\in\Sigma} \left(2 \cdot \min\left(I_\sigma^{P'}, D_\sigma^{P'}\right)\right)$$

So by using Lemma 7 again and Eq. (3) we find that $edm^P < edm^{P'}$. □

Lemmas 7 and 8 show that after computing the edit-distance, we can take *any* optimal path which transfers S into T, and reduce the cost by exchanging inserts and deletes of the same character with one move operation. This can be done in $O(n + |\Sigma|)$ time, with the help of a $|\Sigma|$ size array.

6. Reduction to only move operations

Lemma 7 shows that every transformation of S into T must insert the missing characters and delete the extra ones. Therefore we perform a reduction to only moves, which makes the proof of the bound in the following section easier.

Given two strings S and T, we preprocess these strings and construct two new strings S' and T' as follows:

1. Let # and \$ be two new characters, i.e. #, \$ $\notin \Sigma$.
2. Define $\Sigma^1 = \{\sigma \in \Sigma\colon n_\sigma^S < n_\sigma^T\}$ and $\Sigma^2 = \{\sigma \in \Sigma\colon n_\sigma^T < n_\sigma^S\}$.
3. $$S' \leftarrow S \cdot \prod_{\sigma\in\Sigma^1} \left((\#\sigma)^{n_\sigma^T - n_\sigma^S}\right)\$^{\sum_{\sigma\in\Sigma^2}(n_\sigma^S - n_\sigma^T)}$$
$$T' \leftarrow T \cdot \#^{\sum_{\sigma\in\Sigma^1}(n_\sigma^T - n_\sigma^S)} \prod_{\sigma\in\Sigma^2} \left((\$\sigma)^{n_\sigma^S - n_\sigma^T}\right)$$

For example: If $S = abcab$ and $T = abcdc$ then after preprocessing these strings we get $S' = abcab\#d\#c\$\$$ and $T' = abcdc\#\#\$a\$b$. This way for every $\sigma \in \Sigma \cup \{\#, \$\} n_\sigma^{S'} = n_\sigma^{T'}$, and so we only have to use move operations.

Lemma 9. *If the costs assigned to each insert, delete and move operation are all equal then* $edm(S, T) = edm(S', T')$.

Proof. Suppose A is a sequence of move, insert, and delete operations that convert S into T by a minimum cost. The following operations convert S' into T'. Every insert operation is changed into a move operation of the appropriate character within the new characters into the same position in S. Every delete operation is a move operation of the character which is deleted to an appropriate position following a \$ sign. Every move operation remains unchanged. Therefore, $edm(S', T') \leqslant edm(S, T)$. As there are no two consequent # signs in S', but they must occur continuously in T', and as the \$ signs occur continuously in S' and must be separated by one character in T', there are at least $\sum_{\sigma\in\Sigma} |n_\sigma^T - n_\sigma^S|$ operations needed in order to locate the $\sum_{\sigma\in\Sigma} |n_\sigma^T - n_\sigma^S|$ # and \$ symbols at their final position. These operations do not influence the original characters of S and T. The number $\sum_{\sigma\in\Sigma} |n_\sigma^T - n_\sigma^S|$ is the number of insert and delete operations executed when converting S into T. If $edm(S', T') < edm(S, T)$ it means that some of the move operations required to convert S into T were not executed when converting S' into T'. Therefore, there is a cheaper way to convert S into T, which contradicts the minimalism of $edm(S, T)$. □

7. Bounds between optimal and greedy

Recall that any sequence of moves, performed by the greedy or optimal algorithm, produces a division of the string into blocks of characters, and both S and T contain the same blocks. We shall see that a bound on the number of greedy blocks, as a function of the optimal blocks, gives a bound on the number of move operations (Lemma 11). Unfortunately, the lower bound of the number of blocks is polynomial.

Chrobak et al. [3] consider the *Minimum Common String Partition* (*MCSP*) problem, and refer to several versions of this problem by limiting the number of times each character can occur in both input strings. They provide a lower bound of $\Omega(n^{1/\log_2 5})$ for the ratio of the number of blocks of greedy compared to optimal. Hence, in general, one cannot expect the greedy algorithm to perform well in the worst case for all inputs. Here, we introduce the notion of primary blocks and show that if all contain a constant number of greedy blocks, then a bound of $O(\log n)$ can be achieved.

Recall that in Step 1 of the greedy algorithm the LCS of the two strings is replaced by meta-characters over and over. These meta-characters form a partition of the original strings into pairs of greedy phrases. The greedy phrases

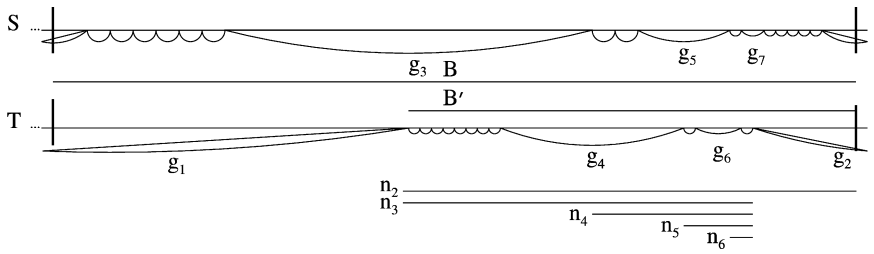

Fig. 2. Schematic view of Lemma 10's proof. The curve lines represent the greedy phrases of the optimal block given by the vertical lines.

are the substrings in S and T corresponding to the meta-characters. That is, there is a one to one correspondence between greedy phrases of S and those of T. We now consider the case where the optimal and greedy parsing differ. We look at an optimal block in S and the corresponding optimal block in T, which contain several greedy phrases as illustrated in Fig. 2. (If there isn't such a block the greedy parsing is optimal.) Since the two optimal blocks are identical we can associate to each greedy phrase in S/T the substring which is formed out of the same characters and occurs in the corresponding location in the optimal phrase in T/S. We call this substring the *corresponding string* of the primary phrase. For simplicity the optimal blocks of S and T in Fig. 2 were moved to be aligned. Thus the corresponding string of a primary phrase in S/T is the substring which is exactly below/above it in the figure. For example, the greedy phrase g_3 in T corresponds to the substring in S starting from a suffix of g_2 and up to a prefix of g_4 (including the two phrases in between).

Definition. A greedy phrase of S/T is a *primary phrase* if it was chosen before any greedy phrase which is properly contained in the corresponding string in T/S. For example, the solid curve lines in Fig. 2 illustrate primary phrases, but the dotted curve lines illustrate non-primary phrases.

Recall that n is the number of characters in S/T. The following lemma gives a bound on the number of primary phrases.

Lemma 10. *There are at most* $\log n$ *primary phrases in one optimal phrase of* S *and its corresponding occurrence in* T.

Proof. Let B be an optimal phrase in S which contains the maximum number of greedy phrases. The reduction to only moves ensures that every phrase in S occurs also in T and vice versa. In particular any optimal phrase in S/T and any substring of this phrase occurs also in T/S. We prove that there are no more than $\log n$ primary phrases in B and its corresponding occurrence in T.

The greedy algorithm creates a non-increasing sequence of the lengths of the LCS's. Since B is a phrase which occurs in both S and T, we examine the point in time when the greedy algorithm has chosen other phrases. Let g_1 be the first greedy phrase chosen, which overlaps B in T. Without loss of generality, let us assume that it crosses the left boundary of B in T as seen in Fig. 2. As B was not chosen by the greedy algorithm, g_1 is at least as long as B. If g_1 crosses also the right boundary, then the substring B in T is contained in g_1, and there is only a single primary phrase in both occurrences of B in S and in T, since greedy phrases of B in S are not primary. Therefore, in the worst case g_1 crosses only one boundary of B.

Denote by B' the subblock of B in T eliminating the common characters of g_1. Since B' is a common substring of S and T and is still free to be chosen by the greedy algorithm, the next greedy phrase, g_2, must be at least as long as B'. If the third greedy phrase chosen, g_3, is also chosen in T then g_1,g_2 and g_3 contain B, and there are only 3 primary phrases in both occurrences of B in S and in T (since, again, the future greedy phrases of S are not primary). Thus, suppose that g_3 is chosen in S. In the worst case it either overlaps a suffix of g_1 or a prefix of g_2. (Otherwise, we again attain only three primary phrases.) In Fig. 2 g_3 overlaps (w.l.o.g.) a suffix of g_1.

In general we define a sequence $G = \{g_1, g_2, \ldots, g_k\}$ of primary blocks in S or in T, ordered by the time they were chosen by the greedy algorithm, and have the following properties: The first two greedy phrases, g_1 and g_2, are chosen as described above. For $i > 2$, g_i is a substring of B', which includes at least one more character of B' which is not already part of previous greedy blocks $\{g_1, \ldots, g_{i-1}\}$. Thus the g_i's form an increasing cover of B'. Without loss of generality we can assume that the primary blocks g_i are chosen from left to right. Note that the set G is finite since both S and T are finite strings, and that $|g_i| \geqslant |g_{i+1}|$.

If g_{i-1} and g_i ($i > 1$), are both chosen in S/T, then these two greedy blocks end the G sequence, i.e. $i = k$ (since g_i is the longest common substring of S and T, and these two occurrences in B' are still free to be chosen by the greedy algorithm).

We still have to prove that $k \leqslant \log n$. That is, that the longest sequence of alternating g_i's is less than or equal to $\log n$. For $i \geqslant 3$, define n_i to be the number of characters in B remaining after eliminating those that are common to $\{g_1, \ldots, g_{i-1}\}$ (see Fig. 2). Before choosing the block g_i, S and T share a common substring in B of length n_i. We have chosen g_i since $|g_i| \geqslant n_i$.

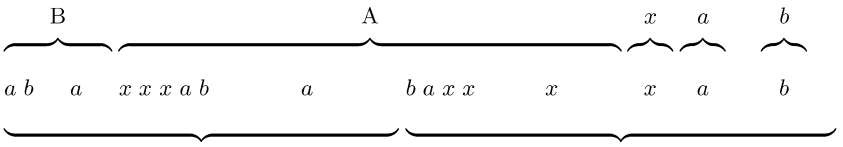

Fig. 3. The different parsing of the greedy and optimal algorithms.

Since the greedy blocks in S/T do not overlap, $|g_{i+1}| + |g_{i+3}| \leqslant n_i$. From the fact that $|g_{i+1}| \leqslant |g_i|$, we find that

$$2n_{i+3} \leqslant 2|g_{i+3}| \leqslant n_i \tag{4}$$

The longest sequence of $\{g_i\}$'s occurs when they are chosen alternately in S and in T. Using Eq. (4), after j stages: $n_i \geqslant 2n_{i+3} \geqslant 4n_{i+6} \geqslant \cdots \geqslant 2^j n_{i+3j}$. This alternating series terminates when the last block consists of exactly one character, i.e., when $n_{i+3j} = 1$. In particular, $n \geqslant |B| > n_2$, thus $n > 2^j \cdot 1$ and $\log n > j$. This means that there are at most $\log n$ greedy phrases in the G sequence. □

Although it is not possible in general to bound the number of greedy phrases covered by a primary phrase, preliminary experiments indicate that problems for which there are only O(1) greedy phrases for each primary phrase may be a broad class in practice. For example, in [13], using different distributions of move sizes, problems were generated where S is a DNA file and T is obtained by performing random moves on S. The greedy algorithm was run (with no knowledge of how T was generated) to compute an edit distance between S and T. Although the number of moves used to generate T from S is not necessarily the same as the optimal number of moves required to convert S to T, it is very close to it with high probability for a wide class of distributions of move operations. Over all experiments presented in [13], the greedy algorithm was never more than a factor of 6 worse (in most cases the factor was much lower).

Note that by definition the primary blocks alternate, and the corresponding occurrence of the string of a primary block of S does not contain a primary block of T, and vice versa. The following lemma gives us a bound on the number of move operations when the bound on the number of phrases is known.

Lemma 11. *Every greedy parsing of a string into N blocks gives a lower bound of $\frac{N+1}{3}$ move operations, and an optimal upper bound of $N - 1$ for this edit distance with moves.*

Proof. If these blocks occur in their reversed order, then every block except one must be moved, which gives us $N - 1$ operations. Every move operation creates at most 3 new boundaries (two in its source location, and one in its destination). Thus, N blocks, which have $N + 1$ boundaries, reflect at least $\frac{N+1}{3}$ move operations. □

Corollary. *If all primary phrases contain a constant number of greedy phrases then the number of move operations in this greedy parsing is at most a* $\log n$ *factor times the number of move operations in an optimal parsing.*

Proof. Lemma 10 gives us a bound on the number of primary phrases in a single optimal phrase, and Lemma 11 gives us at most a factor of three on the number of moves. □

To illustrate the different parsing of the greedy and optimal algorithms, suppose $S = abaxxxababaxxxxab$ and $T = baxxxxababaxxxaba$. Using the greedy algorithm we find that $S' = BAxab$ and $T' = baxAB$, where $A = xxxababaxxx$ and $B = aba$. The edit distance of S' and T' is 4 (since these blocks occur in their reverse order). The optimal parsing of S and T includes only two blocks which require only one move operation, $move(baxxxab, 1)$ (Fig. 3).

8. Future research

Here we have shown the edit distance problem with substring move operations to be NP-complete and have presented a greedy algorithm that is a worst-case log factor approximation to optimal for a broad class of instances the problem. It would be useful to more tightly characterize this class, as well as to investigate modifications of the greedy

approach that might further broaden the class. We have limited our attention to when all operations have the same unit cost, and hence an obvious area of future research would be to extend these ideas to non-uniform costs.

Another area of interest is the incorporation of substring deletions, which are needed to capture the full power of the linked list model (to within a constant factor), where an edit-distance computation can modify both the data items and the links.

Acknowledgements

We thank Maxime Crochemore for suggesting the edit distance with the exchange operation, which helped to motivate this work. We also thank the anonymous referees for their careful work and useful comments.

References

[1] V. Bafna, P.A. Pevzner, Genome rearrangements and sorting by reversals, in: 34th IEEE Symposium on Foundations of Computer Science, (1993, pp. 148–157.
[2] V. Bafna, P.A. Pevzner, Sorting by transpositions, SIAM Journal on Discrete Mathematics 11 (2) (1998) 124–240.
[3] M. Chrobak, P. Kolman, J. Sgall, A greedy algorithm for the minimum common string partition problem, in: Proceedings of 7 International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX 04), 2004, pp. 84–95.
[4] G. Cormode, M. Paterson, S.C. Sahinalp, U. Vishkin, Communication complexity of document exchange, in: Proceedings of the 11th Symposium on Discrete Algorithms, 2000, pp. 197–206.
[5] G. Cormode, S. Muthukrishnan, The string edit distance problem with moves, in: Proceedings of the 13th Symposium on Discrete Algorithms, 2002, pp. 667–676.
[6] F. Ergun, S. Muthukrishnan, S.C. Sahinalp, Comparing sequences with segment rearrangements, in: Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2003), 2003, pp. 183–194.
[7] M.R. Garey, D.S. Johnson, Computers and Intractability, A Guide to the Theory of NP-Completeness, Bell Laboratories Murry Hill, NJ, 1979.
[8] D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, Cambridge, 1997.
[9] S. Hannenhalli, Polynomial-time algorithm for computing translocation distance between genomes, in: Annual Symposium of Combinatorial Pattern Matching CPM, 1996, pp. 162–176.
[10] J. Kececioglu, D. Sankoff, Exact and approximation algorithms for the inversion distance between two permutations, in: Proc. of 4th Ann. Symp. on Combinatorial Pattern Matching, in: Lecture Notes in Computer Science, vol. 684, Springer, Berlin, 1993, pp. 87–105.
[11] D. Lopresti, A. Tomkins, Block edit models for approximate string matching, Theoretical Computer Science 181 (1997) 159–179.
[12] W.J. Masek, M.S. Paterson, A faster algorithm computing string edit distances, Journal of Computer and System Sciences 20 (1980) 18–31.
[13] M. Meyerovich, Longest common substrings and edit distance, Senior Thesis, Brandeis University, 2003.
[14] S. Muthukrishnan, S.C. Sahinalp, Approximate nearest neighbors and sequence comparison with block operations, in: STOC'00, ACM Symposium on Theory of Computing, 2000, pp. 416–424.
[15] S. Muthukrishnan, S.C. Sahinalp, Simple and practical sequence nearest neighbors with block operations, in: Annual Symposium of Combinatorial Pattern Matching CPM, 2002, pp. 262–278.
[16] J.A. Storer, An Introduction to Data Structures and Algorithms, Birkhäuser–Springer, 2001.
[17] W.F. Tichy, The string to string correction problem with block moves, ACM Transactions on Computer Systems 2 (4) (1984) 309–321.