Edit Distance with Move Operations

Dana Shapira and James A. Storer
Computer Science Department
shapird/storer@cs.brandeis.edu

Brandeis University, Waltham, MA 02254

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 strings being 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 also 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 operations imply O(1) substring exchange operations as well, a form of transformation that has been of interest in molecular biology. We show that this problem is NP-complete, show that a "recursive" sequence of moves can be simulated with at most a constant factor increase by a non-recursive sequence, and present a polynomial time greedy algorithm for non-recursive moves with a worst-case log factor approximation to optimal. The development of this greedy algorithm shows how to reduce moves of substrings to moves of characters, and how to convert moves with characters to only insert and deletes of characters.

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 and delete-character operations where both have 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 [?] 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 values of the edit distance is needed, only one row of the matrix is necessary, and the space complexity is $O(m)$.

In addition to the insert and delete operations, we allow move operations that transfer a sequence of characters 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 allow operations to apply to the pointer associated with a vertex as well as the character associated with the vertex. To define the problem properly, 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 (\$'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. Move operations 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 $m << n$ 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, and visa verse. Using the new model the edit distance is reduced to $O(1)$, by changing $O(1)$ pointers.

Kececioglu and Sankoff [?] and Bafna and Pevzner [?] consider the reversal model, 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, instead of moving a substring B to the right in S, let C be the substring of S between A and its destination position (so for some possibly empty strings A and D we wish to transform $S = ABCD$ to $ACBD$) and we can simply 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 [?] consider approximate nearest neighbors and sequence comparison with block operations, and without "recursion". With block operations, they include moves, copies, deletes, and reversals. The addition of the copy and reversal block operations changes the problem greatly. Reversals are more powerful than moves (i.e., reversals can simulate moves with at most a constant factor increase in cost but it is not necessarily true that moves can simulate reversals). 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.

Bafna and Pevzner [**?**] 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 similar in related to the problem considered here, the restriction that all characters are distinct greatly changes the problem.

Lopresti and Tompkins [**?**] 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 [**?**] 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. Hannenalli [**?**] 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.

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 $1 \leq p \leq n$, the operation $insert(\sigma, p)$ inserts the character σ to the p^{th} position of S. After performing this operation, S is of the form $s_1 \cdots s_{p-1} \sigma s_p \cdots s_n$. The operation $delete(p)$ deletes the character which occurs at the p^{th} position of S, and returns the character which was deleted, i.e., $s = s_1 \cdots s_{p-1} s_{p+1} \cdots s_n$ and it returns the character s_p. Given two distinct positions $1 \leq p_1 \neq p_2 \leq n$ and a length $1 \leq \ell \leq n - p_1 + 1$, $move(\ell, p_1, p_2)$ moves the string at position p_1 of length ℓ to position p_2. After performing the move operation, if $p_1 < p_2$ then S is in the form of $s_1 \cdots s_{p_1-1} s_{p_1+\ell} \cdots s_{p_2-1} s_{p_1} \cdots s_{p_1+\ell-1} s_{p_2} \cdots s_n$, and if $p_2 < p_1$, S is in the form of $s_1 \cdots s_{p_2-1} s_{p_1} \cdots s_{p_1+\ell-1} s_{p_2} \cdots s_{p_1-1} s_{p_1+\ell} \cdots s_n$. For simplicity, we may write $move(str, p_1, p_2)$, where str is the string which is moved.

In the following section we show that computing the edit-distance between two linked lists 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 a constant factor 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. Sections 6 and 7 then show that 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: Given two strings S and T, an integer $m \in N$, using only the three unit-cost operations *insert*, *delete*, and *move*. It is NP-complete to determine if S can be converted to T with cost $\leq m$.

Proof: Since a non-deterministic algorithm need only guess the operations and check in polynomial time that S is converted into T with cost $\leq m$, 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 $X = \{x_1, ..., x_n\}$ where $x_i \leq B$, and a positive integer k, the bin packing problem is to determine if there is a partition of X into disjoint sets $X_1, ..., X_k$, 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 [**?**]); 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, $X = \{x_1, ..., x_n\}$, and k of the bin-packing problem, let a, #, and \$ denote three distinct characters, and let:

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

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

$$m = n$$

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 $\leq m$ if and only if there is a partition of X into disjoint sets $X_1, ..., X_k$ 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 $X_1, ..., X_k$, such that the sum of the items in each X_i is B or less. Then S can be transformed to T by, for each item $x_i \in X_j$, $1 \leq j \leq k$, 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, the full draft of this paper uses a sequence of lemmas to show that if the edit distance with moves between S and T is $\leq n$, then there is a bin packing in k bins of size B of the items of X. ∎

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

The next 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 sub sequence of A defines a partition of S, and if $A_1 \subseteq A_2$ are two sub sequences 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 is *recursive* if it contains a move operation which moves a string for which its characters did not occur continuously in S.

For example, if S is the string *abcde* and the character b was 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 6 ($Move(ab, 2, 7)$, $Move(a, 3, 8)$, $Move(b, 4, 8)$, $Move(cd, 7, 2)$, $Move(c, 8, 3)$ and $Move(d, 9, 3)$). By allowing recursive moves we can reduce the cost to be 4 operations ($Move(b, 5, 4)$, $Move(d, 10, 9)$, $Move(abba, 2, 7)$ and $Move(cddc, 7, 2)$.

THEOREM: *Suppose there is a recursive sequence, A, of size n which converts S into T. Then a non recursive optimal algorithm uses no more than $3n$ moves.*

PROOF: By induction on n.

A worst case example: In the full draft of this paper we give an infinite class of strings for which non-recursive is a factor of 3 more costly than recursive.

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 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 not any 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 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 Figure 1.

```
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 $d$
```

FIGURE 1: *The Greedy algorithm*

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 $d \leq 2$. 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 reduces the strings S and T to (possibly) shorter strings by replacing the $LCS(S,T)$ by a new single character. In a first attempt it seems as if we must replace every occurrence of the $LCS(S,T)$ by a new character, without bothering about the same number of replacements in S and T. Otherwise, if the number of occurrences of the $LCS(S,T)$ in S and T is not equal, the copies which were not replaced by the same new character, are not noticed as resemble ones, which might increase the edit-distance. In the following Lemma we show that this is not the case. We denote the edit distance returned by this version of the greedy algorithm by *greedy′* (i.e., the version without the restriction of the equal number of replacements in both S and T). Let *opt* denote the edit-distance returned by an optimal algorithm which allows move operations.

LEMMA 1: *The unrestricted version of the greedy algorithm is not bounded.*

PROOF: For every n there exists an example such that $\frac{greedy'}{opt} > n$. Let $\Sigma = \{a,b\}$. Let $S = (ab)^{4n}$ and let $T = (ab)^{2n}(ba)^{2n}$.

The optimal edit distance is 2 ($insert(b, 4n+1)$, $b \leftarrow delete(8n)$). By this version of the greedy algorithm S is reduced to the string AA and T is reduced to the string $A(ba)^{2n}$, where $A = (ab)^{2n}$. The edit distance is $2n+1$ which includes the operations: $A \leftarrow delete(2)$, $insert(b, i-1)$ and $insert(a, i)$ for i, $4n+2 \leq i \leq 8n$. Now, $\frac{greedy'}{opt} = \frac{2n+1}{2} > n$. ∎

The problem is that A occurs twice in T and only once in S. Therefore the algorithm could not identify any resemblance between A and $(ba)^{2n}$. We overcome this problem by replacing the same number of occurrences of the $LCS(S,T)$ in S and in T as done by the greedy algorithm with the replace operation of the while loop.

Note that 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 substring 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 a first sight we might think that we cannot separate these two stages. It seems as if in every stage of the dynamic 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 notations: 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 where done when converting S into T using the path P. The edit-distance between the string S and T which is done 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*(σ) and *delete*(σ) 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 2: *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 $|I_\sigma^P - D_\sigma^P| = |n_\sigma^S - n_\sigma^T|$. ∎

CONCLUSION 1: *Any two paths P and P' differ only by move operations.*

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

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

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

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

LEMMA 3: *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 2 and equation (1) we find that

$$\sum_{\sigma\in\Sigma}\left(2\cdot\min(I^P_\sigma, D^P_\sigma)\right) < \sum_{\sigma\in\Sigma}\left(2\cdot\min(I^{P'}_\sigma, D^{P'}_\sigma)\right)$$

So by using Lemma 2 again and equation (3) we find that $edm^P < edm^{P'}$ ∎

Lemma 2 and Lemma 3 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

Using Lemma 2 we already know that for any two paths P and P' and any character $\sigma\in\Sigma$, $|I^P_\sigma - D^P_\sigma| = |I^{P'}_\sigma - D^{P'}_\sigma|$. Recall that n^S_σ and n^T_σ denote the number of appearances of σ in the strings S and T, respectively.

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 $\Sigma^1 = \{\sigma\in\Sigma : n^S_\sigma < n^T_\sigma\}$ and $\Sigma^2 = \{\sigma\in\Sigma : n^T_\sigma < n^S_\sigma\}$.
3. $$S' \leftarrow S\cdot\prod_{\sigma\in\Sigma^1}\left((\#\sigma)^{n^T_\sigma - n^S_\sigma}\right)\$^{\sum_{\sigma\in\Sigma^2}(n^S_\sigma - n^T_\sigma)}$$
$$T' \leftarrow T\cdot\#^{\sum_{\sigma\in\Sigma^1}(n^T_\sigma - n^S_\sigma)}\prod_{\sigma\in\Sigma^2}\left((\$\sigma)^{n^S_\sigma - n^T_\sigma}\right)$$

For example: If $S = abcab$ and $T = abcdc$ then after preprocessing these strings we get that $S' = abcab\#d\#c\$\$$ and $T' = abcdc\#\#\$a\$b$. This way $\forall\sigma\in\Sigma\cup\{\#,\$\}$ $n^{S'}_\sigma = n^{T'}_\sigma$, and we can deal only with move operations.

LEMMA 4: *If the cost 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 which are needed in order to convert S into T by a minimum cost. The insert and delete operations are done only when there is a missing or an additional character, respectively. 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, $edm(S',T') \le edm(S,T)$.

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 $\sum_{\sigma\in\Sigma}|n^T_\sigma - n^S_\sigma|$ operations needed in order to locate the $\sum_{\sigma\in\Sigma}|n^T_\sigma - n^S_\sigma|$ # and \$ symbols at their final position. These operations do not influence the original characters of S and T. The number $\sum_{\sigma\in\Sigma}|n^T_\sigma - n^S_\sigma|$ is the number of insert and delete operations done when converting S into T. If $edm(S', T') < edm(S, T)$ it means that some of the move operations done to convert S into T were not done 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

Finding a bound on the number of greedy phrases, as a function of the optimal phrases, gives us a bound on the number of move operations (see Lemma 6). Obviously, we are not concerned with the greedy blocks which contain optimal ones. Therefore, let us look at an optimal block in S which contains N greedy blocks. The following Lemma gives us a bound on the number of greedy blocks of the corresponding optimal block in T, which contain "most" of these phrases.

FIGURE 2: *Schematic view of Lemma 5's proof. The curve lines represent the greedy phrases of the optimal block given by the vertical lines.*

LEMMA 5: *Let N be the number of optimal phrases, and let L be the number of characters in the longest optimal phrase. If B is an optimal block which contains m greedy blocks in S, then at least $O(m - \log L)$ of these blocks are part of at most $O(\log L)$ greedy blocks, in the correspondence occurrence of B in T.*

PROOF: Denote by B' the sub block of B containing exactly these m greedy blocks. The greedy algorithm creates a non increasing sequence of the lengths of the LCS's. As we know that B' is a string which occurs in both S and T, we examine the point in time when the greedy algorithm had chosen other phrases. Suppose C is the first greedy phrase chosen, which overlaps B'. As B' occurs in both S and T, and was not chosen by the greedy algorithm, it must be that C was chosen in T, and that $|C| \geq |B'|$. Obviously, C crosses an optimal block boundary. Without loss of generality, let us assume that it crosses the left boundary of B' in T. If C contains at least $O(m - \log L)$ greedy phases, we are done. If there exists a block in T which crosses B''s right boundary denote it by D. If C and D contain together at least $O(m - \log L)$ greedy phases, we are done. Let us define the sequence $G = \{g_1, g_2, ..., g_k\}$ to be a particular sequence of greedy blocks ordered by time, which occur in S or in T, with the following properties: The first greedy block in this sequence is C, i.e., $g_1 = C$, and for $i > 1$, g_i is a substring of B', which includes at least one more character of B' which does not already occur in the previous greedy blocks $\{g_1, ..., g_{i-1}\}$. Thus the g_i's form an increasing cover of B'. Note that the set G is finite since both S and T are finite strings, and that $|g_i| \geq |g_{i+1}|$.

Observation 1: If g_2 was chosen in T, then $C(= g_1)$, D and g_2, cover the m greedy blocks of S, in particular they include $O(m - \log L)$ greedy phases. (Since the string between C and D occurs in both S and T and is still free to be chosen).

Observation 2: If g_{i-1} and g_i $(i > 1)$, were both chosen in S/T, then these two greedy blocks end the G sequence, i.e. $i = k$. Since the adjacent block to g_i in S/T is the longest common substring in B' of S and T, and these two occurrences are still free to be chosen. In this case, about half of these k greedy blocks of T cover the $O(N - \frac{k}{2})$ greedy blocks of S.

We still have to prove that $k = O(\log n)$. That is, that the longest sequence of alternating g_i's is of order $\log n$. For $i \geq 2$, define n_i to be the number of characters in the suffix string of B' starting at the first position of g_i. Before choosing the block g_{i+1}, S and T share a common substring in B' of length $n_i - |g_i|$. We have chosen g_{i+1} since $|g_{i+1}| \geq n_i - |g_i|$. Using the fact that $|g_i| \geq |g_{i+1}|$, we find that $|g_i| \geq n_i - |g_i|$, thus

$$|g_i| \geq \frac{n_i}{2} \tag{4}$$

Since the greedy blocks in S/T do not overlap, the difference between n_i and n_{i+2} $(i = 2, ..., k-2)$ is at least $|g_i|$, i.e., $n_i - n_{i+2} \geq |g_i|$. Therefore, by using equation (4), $2|g_i| \geq n_i \geq |g_i| + n_{i+2}$, so we find that

$$|g_i| \geq n_{i+2} \tag{5}$$

The longest sequence of $\{g_i\}$'s occurs when they are chosen alternately in S and in T and their lengths are as small as possible, i.e., when $|g_i| = \frac{n_i}{2}$. If $|g_i| = \frac{n_i}{2}$, then by using equation (5), after j stages: $n_i \geq 2n_{i+2} \geq 4n_{i+4} \geq \cdots \geq 2^j n_{i+2j}$. This alternate series terminates when the last block consists of exactly one character, i.e., when $n_{i+2j} = 1$. In particular, $L \geq |B| > n_2$, so we find that $L > 2^j \cdot 1$. Therefore, $\log L > j$. This means that there are at most $\log L$ greedy phrases in the G sequence, and that the $\{g_i\}$'s in T correspond to the m greedy blocks of B excluding those who belong to the $\{g_i\}$'s in S. ∎

A greedy block is called a *primary* one, if it is a member of the sequence G. Intuitively, a primary block is one of the "big" greedy phrases either in S or in T. We have proved that the primary blocks alternate, and that the corresponding occurrence of the string of a primary block of S does not contain a primary block of T, and vise verse.

The following theorem gives us a bound on the number of greedy phrases.
THEOREM: *The number of phrases in a greedy parsing is at most a* $\log(L)$ *times the number of phrases in an optimal parsing.*

PROOF: Given an optimal and greedy parsing of the strings S and T, recall that the number of optimal phrases is denoted by N. We show that the greedy parsing does not consist of more then $O(N \log L)$ phrases. The reduction to only move operations which was presented in the previous section ensures that each greedy or optimal phrase in S occurs also in T, and vise verse.

Let us relate to the N optimal phrases of S. There are at most $N \log(L)$ primary phrases, of which about half of them are in S. We now count the number of the remaining phrases in S, by associating each one of them to a different primary phrase of S or to a different optimal boundary of S. Since there are not more than $N \log(L)$ primary phrases and not more than $N-1$ optimal boundaries, this concludes our proof.

FIGURE 3: *Schematic view of the proof. The bold curve lines represent primary phrases of the greedy algorithm corresponding to different occurrences of the string $\mathcal{C}$.*

Consider a primary block, C, in T which covers ℓ greedy phrases of S, where $\ell >= 1$. The block C occurs in S, too. We refer to the correspondence string of the block C as $\mathcal{C}$. If C crosses an optimal boundary, this boundary can cross at most one of the ℓ original greedy phrases. We associate this particular phrase with this optimal boundary. Otherwise, suppose C is fully contained in an optimal block α in S, and therefore appears in the corresponding occurrence of α in T. If C in S is a non primary block of α, then the correspondence occurrence of the string $\mathcal{C}$ in T is contained in a primary block D of T, and we continue to the following appearance of D in S. Finally we end this process when the string $\mathcal{C}$ occurs in a primary block of S.

Let us refer to the correspondent occurrence of this primary block in T. The greedy phrases of C were not merged by the greedy algorithm, since at least $\ell - 2$ of these original greedy phrases are covered by at most 2 greedy phrases. We associate these two uncovered phrases to this primary block of S. This process ends when there are not two adjacent original greedy phrases which remain attached in T. Otherwise these adjacent blocks can be merged into one larger block. Since every primary block is charged at most once, and each greedy block in T was separated (and therefore was counted), we have included all ℓ phrases in this count. ■

The following Lemma gives us a bound on the number of move operations when the bound on the number of phrases is known.

LEMMA 6: *Every parsing of N blocks, gives a lower bound of $\frac{N+1}{3}$ move operations, and an optimal upper bound of $N-1$.*

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: *The number of move operations in a greedy parsing is at most a $\log L$ factor times the number of move operations in an optimal parsing.*

PROOF: Lemma 6 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 $S = abaxxxabababaxxxxab$ and $T = baxxxxabababaxxxaba$. Using the greedy algorithm we get that $S' = BAxab$ and $T' = baxAB$, where $A = xxxabababaxxx$ and $B = aba$. The edit distance of S' and T' is 4 (since these blocks occur at their reverse order). The optimal parsing of S and T includes only two blocks in which require only one move operation,$move(baxxxab, 1)$.

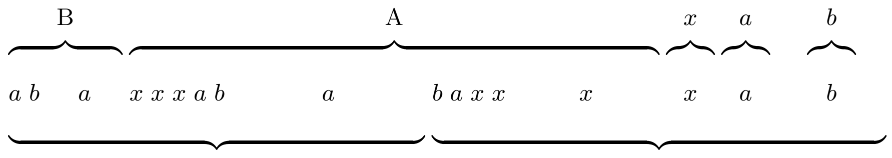

FIGURE 4: *The Different parsing of the Greedy and Optimal algorithms*

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. We have limited out attention to when all operations have 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). Experiments with the greedy algorithm on real-life data (e.g., from molecular biology) are also of interest, although given the NP-completeness of an optimal computation, a framework for measuring performance of greedy for a particular application needs to be addressed.

Acknowledgments We thank Maxime Crochemore for suggesting the edit distance with the exchange operation, which helped to motivate this work.

References

[1] Bafna V. and Pevzner P.A., Genome rearrangements and sorting by reversals, *34th IEEE Symposium on Foundations of Computer Science*, (1993) 148–157

[2] Bafna V. and Pevzner P.A., Sorting by transpositions, *34th SIAM J. Discrete Math., 11(2)*, (1998) 124–240

[3] Garey M.R. and Johnson D.S., Computers and Intractability, A guide to the Theory of NP-Completeness, *Bell Laboratories Murry Hill, NJ*, (1979)

[4] Hamming R.W., Coding and information Theory, *Englewood Cliffs, NJ, Prentice Hall*, (1980)

[5] Hannenhalli S., Polynomial-time Algorithm for Computing Translocation Distance between Genomes *CPM*, (1996) 162–176

[6] Kececioglu J. and Sankoff D., Exact and approximation algorithms for the inversion distance between two permutations. *Pro. of 4th Ann. Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science 684*, (1993) 87–105

[7] Liben-Nowell D., On the Structure of Syntenic Distance, *CPM*, (1999) 50–65

[8] Lopresti D. and Tomkins A., Block Edit Models for Approximate String Matching, *Theoretical Computer Science, 181*, (1997) 159–179

[9] Muthukrishnan S. and Sahinalp S.C., Approximate nearest neighbors and sequence comparison with block operations, *STOC'00, ACM Symposium on Theory of Computing*, (2000) 416–424

[10] Smith T.F. and Waterman M.S., Identification of common molecular sequences, *Journal of Molecular Biology, 147*, (1981) 195–197

[11] Storer J. A., *An Introduction to Data Structures and Algorithms*, Birkhauser - Springer, (2001)

[12] Tichy W.F., The string to string correction problem with block moves, *ACM Transactions on Computer Systems, 2(4)*, (1984) 309–321