Prefix Reversals on Strings.

Conference Paper · January 2010

Source: DBLP

CITATIONS
0

READS
213

2 authors:

Bhadrachalam Chitturi
University of Texas at Dallas
50 PUBLICATIONS **197** CITATIONS
SEE PROFILE

Ivan Hal Sudborough
University of Texas at Dallas
163 PUBLICATIONS **2,902** CITATIONS
SEE PROFILE

Prefix Reversals on Strings

Bhadrachalam Chitturi[1], **Hal Sudborough**[2]
[1]Department of Biochemistry, University of Texas Southwestern Medical Center, Dallas, TX 75390, USA
[2]Department of Computer Science, University of Texas at Dallas, Richardson, Texas 75080, USA

Abstract - *A reversal is an operation that reverses a substring. When the chosen substring is restricted to a pre*fix *it is called a pre*fix *reversal. Given two strings, α and β, the problem of finding the minimum number of pre*fix *reversals that transform α into β is called the pre*fix *reversal distance problem. This problem is a version of the reversal distance problem, related to genetic rearrangements. An upper bound of 18n/11 and a lower bound of 15n/14 are known for permutations. We give bounds for pre*fix *reversals on strings. For example, n*-1 *pre*fix *reversals are sufficient to transform a given n-binary string into a compatible string, i.e. a string with same frequency for each symbol as the given string. We also prove that the pre*fix *reversal distance for strings is NP-complete.*

Keywords: Complexity, genetic mutations, prefix reversals

1 Introduction

Transpositions, reversals, prefix reversals or *flips*, and prefix transpositions have been studied over permutations. Gates and Papadimitriou [10] showed that $(5n+5)/3$ flips are sufficient to sort any permutation on n symbols. For a permutation σ of the integers from 0 to n-1, let $f'(\sigma)$ be the smallest number of flips that will transform σ to the identity permutation and let $f(n)$ be the largest such $f'(\sigma)$ over all permutations σ of length n. They showed that $f(n) \leq (5n+5)/3$ for all values of n and $f(n) \geq 17n/16$ when n is divisible by 16. Heydari and Sudborough [13] improved this lower bound to $f(n) \geq 15n/14$, when n is divisible by 7. Chitturi *et al.* [3] improved the upper bound to $18n/11$. Heydari [12] showed that deciding whether one can make k adjacencies in k flips for a given permutation is NP-complete.

Researchers have been studying transforming strings with finite alphabet size since 1990s. Christie and Irving [5] showed that both $d_r(S,T)$ and $d_t(S,T)$ are bounded above by $n/2$, where $d_r(S,T)$ (respectively, $d_t(S,T)$) is the minimum number of reversals (respectively, transpositions) necessary to transform a binary string S of length n into a compatible string T. They also showed that the problem of finding $d_r(S,T)$ for binary strings is NP-hard. Radcliffe *et al.* [15] showed that (a) the reversal distance, $d_r(S,T)$, between a pair of compatible *k-ary* strings of length n is at most $n - max\{a_i\}$ where $max\{a_i\}$ is the frequency of the most frequent symbol, (b) the reversal distance between random strings, each having a positive fraction of every element, is $\theta(n/\log n)$, and (c) a ternary string can be optimally sorted by reversals. They also showed that the problem of finding the transposition distance and the signed reversal distance between two strings over a finite alphabet is NP-hard. Chitturi and Sudborough [4] studied prefix transpositions over strings; they showed that: (a) $d_{pt}(S,T) \leq n-\alpha$, where $d_{pt}(S,T)$ is the prefix transposition distance between S and T, α is the frequency of the most frequent symbol in S (or T), and n is the length of S (T), and (b) $d_{pt}(S,T)$ for strings is NP-complete.

Earlier, Hurkens *et al.* [14] also studied flips on strings. Compared to the current paper, [14] has some similar results and some additional results. Some of their distinct work is: (i) for a k-ary string $d_{pr}(S,T) \leq 2(n-\alpha)$ where d_{pr} is the *flip distance* for strings of length n and α is the frequency of the most frequent symbol, (ii) it is NP-complete to find $d_{pr}(S,T)$ for k-ary ($k > 1$) strings S and T (we give a 2 step proof), and (iii) an algorithm to sort ternary strings in the minimum number of moves.

We denote the reverse of a given string α by α^R. Given two strings π and σ, where $\pi = \alpha\,\beta$ and α, β are strings over a finite alphabet ∑, we denote a *flip* by $\pi \rightarrow \sigma$, where $\sigma = \alpha^R\,\beta$. In this paper a string drawn from ∑ has all the symbols of ∑ unless specified otherwise. For example, a binary string is not referred to as a ternary string. A flip is shown by enclosing the prefix to be reversed in square brackets. A small case letter indicates an element and a capital letter indicates an arbitrary string. We use the standard notation of regular expressions: $(a)^*$ represents zero or more repetitions of a, $(a)^+$ represents one or more repetitions of a, and '(a)?' represents zero or one occurrence of a.

Given a pair of strings S and T, we define two metrics: (1) b_{pr} denotes the number of flip breakpoints between S and T, and (2) b_{epr} denotes the end breakpoint between S and T; enhancing the reversal breakpoints defined in [5]. The function $f_{ab}(S)$ represents the number of 'ab' adjacencies in S, where $\{a, b\} \in \sum$. Function $\delta(x) = x$, if $x > 0$, otherwise $\delta(x)=0$.

$$b_{pr}(S,T) = \sum_{(a,b\in\sum)\wedge(a<b)} \delta(f_{ab}(S) + f_{ba}(S) - f_{ab}(T) - f_{ba}(T)) + \sum_{a\in\sum} \delta(f_{aa}(S) - f_{aa}(T))$$

$b_{epr}(S,T) = 0$ if S and T end in the same symbol, otherwise $b_{epr}(S,T) = 1$.

For two given strings S and T to be identical, a necessary but insufficient condition is: $b_{pr}(S,T) = b_{epr}(S,T) = 0$. For example if S = 0001001 and T = 0010001then $f_{00}(S) = f_{00}(T) = 3$, $f_{01}(S) = f_{01}(T) = 3$, $f_{11}(S) = f_{11}(T) = 0$. Since S and T end with a '1' $b_{pr}(S,T) = b_{epr}(S,T) = 0$; yet S and T are not identical.

Definition 1. We say that a string S defined over $\sum$ is *sorted* if, for any two consecutive elements of S, x and y, y does not occur before x in $\sum$.

A sorted string consists of blocks of symbols with increasing values. The string S = 11222001 after sorting yields $sort(S)$ = S_i = 00111222. We call a substring of length at least one composed entirely of symbol x as an *x block*.

Definition 2. In the *standard notation*, we denote a x block by an x. Note that if a symbol 'z' is adjacent to an x block, then $z \neq x$ (otherwise z would be a part of the x block). The string 1000011110001 is represented as 10101. Let S be a string where S= 010101011 then S_i =$sort(S)$ = 01; where the '0' represents a block of four zeroes and the '1' represents a block of five ones.

For a given string S, o(S) and z(s) denote the number of 1 and 0 blocks respectively; let B be the number of all blocks in S. Thus, the length of S is B (= o(S)+z(s) for binary strings).

Lemma 1. If S' is obtained from S by a single flip, then
(a) $b_{pr}(S',T) \geq b_{pr}(S,T) - 1$ and $b_{epr}(S',T) = b_{epr}(S,T)$*;* if the flip is not [$S$]
(b) $b_{pr}(S',T) = b_{pr}(S,T)$ and $b_{epr}(S',T) \geq b_{epr}(S,T) - 1$; if the flip is [$S$]. ■

Theorem 1. Given a pair of strings S and T, $d_{pr}(S,T) \geq b_{pr}(S,T) + b_{epr}(S,T)$.

Proof: As stated earlier, $d_{pr}(S,T)$ is the minimum number of flips necessary to transform S into T. According to Lemma 1, a flip will change either $b_{pr}(S,T)$ or $b_{epr}(S,T)$ by at most one but it does not change both. Therefore, the sum $b_{pr}(S,T) + b_{epr}(S,T)$ is a lower bound of $d_{pr}(S,T)$. ■

Definition 3. A *single* is a move that reduces (*e.g*. o(S) or z(S)) B by 1. The *left single* is the flip of the shortest prefix that is single. Given S = 10010011, the singles are (a) [100]10011, (b) [100100]11, and (c) [1001001]1 where only (a) is the left single. If only one single is possible, then it is a left single. A move that is not a single is called *non-single*.

Lemma 2. If S' is obtained from S with one flip then o(S') ≥ o(S) –1 and $z(S)$ = z(S') or z(S') ≥ $z(S)$ – 1 and o(S) = o(S').

Proof: The flip ([x…x^c] x^+…) reduces either o(S) or z(S) by 1 but it does not reduce both. The flip ([x…x]…) does not alter o(S) or z(S). The flip ([x...xc]x^c…) increments only one of o(S) and z(S) by one. So, our claim holds. ■

Lemma 3. There is a flip that reduces B by one unless the string in standard notation is 01, or 10.

Proof: If the string in the standard notation is neither '01' nor '10' then it is of the form (a) $(x\ x^c)^+$, or (b) $(x\ x^c)\ x$. If the string is of either form, the flip ([x…x^c]x…) reduces B by one and the resultant string also is of the form (a) or form (b). Thus, the same flip can be repeated to obtain $x\ x^c$, *i.e.* , one obtains 01 or 10. ■

Theorem 2. Let S_i =sort(S). Then $d_{pr}(S,S_i) = B$–2 if S ends with 1 else $d_{pr}(S,S_i) = B$–1.

Proof: S_i contains exactly two blocks and S contains exactly B blocks. Since at most one block can be reduced by a flip, a minimum of B–2 flips are required to sort S. However, if S does not end in '1', a flip of entire string is required at some point, since S_i ends in '1' (Lemma 1). Thus, the minimum number of flips required to sort S is B–2 if S ends with 1, B–1 otherwise. 2Sort sorts S with the minimum number of flips:

2Sort(S)
1) Execute left singles on S until no more singles are possible.
2) If the resulting string is 01 then terminate the computation. Otherwise, flip the entire string.

Lemma 3 shows that Step 1) is feasible. So, unless there is only one '0' block and one '1' block there is a single and 2Sort executes the single. This process stops only when no singles exist, *i.e.* the current string is of the form $x(x)^c$. ■

Sorting ternary strings in minimum+2 flips

We give a simple algorithm to sort a ternary string that executes no more than two flips over the minimum number of flips. We enumerated all strings of length five and gave minimum moves to sort them (built from moves for strings of lengths four and three). Seventeen of them require 4 flips (the worst case); for example, [02]121→[20121]→ [12]102→ [210]2 →012. Due to lack of space the table is not shown.

3Sort:
Step (1) Execute left singles until either (a) no single exists or (b) B = 5.

Step (2):
Step (2a) If B = 5 then execute the optimal sequence from the lookup table and terminate the computation.
Step (2b) Here, no single exists, yielding **(2b)(i)** and **(2b)(ii)**.
Step (2b)(i) If the string starts with a '2' (representing 2^+), then flip the entire string, the '2' need not be moved again; otherwise execute Step (2b)((ii).
Step (2b)(ii) The string is of the form $x\ ((y\ 2\ )^+\ (y)^* + (2^{\ y}\ )^+\ (2)^*)$, where x = 0 or x = 1 and $y = x^c$. Perform the respective flip [$x\ y$] or [x 2] depending on whether the string is $x\ (y\ 2\ )^+\ (y)^*$ or $x\ (2\ y\ )^+\ (2)^*$ respectively. In case there is more than one choice, the shortest flip is executed. The flips that are executed in individual subcases are listed below:

Step (2b)(ii)(I) For $0(12)^+$ then perform $[01]2(12)^*$.
Step (2b)(ii)(II) For $0(12)^+1$ perform [012] $(12)^*1$ (also possible is $[0(12)^+1]$) .
Step 2b)(ii)(III) For $0(21)^+$ execute $[02]1(21)^*$.
Step (2b)(ii)(IV) For $0(21)^+2$ execute $[02]1(21)^*2$.
Step 2b)(ii)(V) For $1(02)^+$ execute $[10]2(02)^*$.
Step (2b)(ii)(VI) For $1(02)^+0$ execute $[10]2(02)^*0$.
Step (2b)(ii)(VII) For $1(20)^+$ execute $[12]0(20)^*$.
Step (2b)(ii)(VIII) For $1(20)^+2$ execute $[12]0(20)^*2$.

After one of the flips from {2(b)(i), 2(b)(ii)(I) - 2b)(ii)(VIII)} is executed, perform the following two steps:
Step (3) Execute left singles until B = 5.

Step (4) Execute the optimal flips given by the table.

Analysis

We compare the number of flips 3Sort performs to the direct theoretical minimum (that executes only singles), the lower bound. Even an optimum algorithm requires non-singles (2 needed for 201, [201]→[10]2→012). Thus, the number of non-singles executed by 3Sort is an upper bound on the number of moves it executes over the minimum number of flips.

Step (1) does not add to the count of non-singles. In Step (2a), the maximum number of flips required for any string of length five (by exhaustive search) is four and theoretical minimum is two. Thus, here 3Sort is minimum+2.

In Step (2b)(i) 3Sort executes $[2(xy)^+ x(?)]\rightarrow(x)(?)(y\,x\,)^+2$, which is a non-single. After several singles, the resulting string will be either *y x* 2 or *x y* 2. If the second symbol is '1', the string is sorted. Otherwise, '10' is flipped which is a second non-single. However, it must be noted that any algorithm requires one flip to put 2 at the end; so, this move is necessary. Effectively, 3Sort needs at most one additional flip compared to an optimum sort. Thus, here 3Sort is minimum+2.

For Step (2b)(ii) the set of strings is $x\,((y\,2\,)^+ y(?) + (2\,y)^+ 2(?))$. These strings fall into eight categories which are analyzed below showing that 3Sort is minimum+2 where (+1) and (+2) indicate the number of non-singles executed by that category.

Step 2b)(ii)(I): For $0(12)^+$, 3Sort will perform the first non-single: $[01]2(12)^*\rightarrow102(12)^*$. After several left singles, the resultant string is 102. A second non-single on 102 sorts the string, *i.e.* [10]2→012. (+2)

Step 2b)(ii)(II): For $0(12)^+1$, 3Sort will perform the first non-single: $[01]2\ (12)^*1\rightarrow\ 102(12)^*1$. After several left singles, the resultant string will be 20121, which can be sorted with the following sequence: [20121] →[12]102→[210]2→ 012. Only the first flip is a non-single. (+2)

Step 2b)(ii)(III): For $0(21)^+$, 3Sort will perform the first non-single: $[02]1(21)^*\rightarrow201(21)^*$. After several left singles, the resultant string will be 20121, which can be sorted in three moves, *i.e.* [20121]→[12]102→ [210]2 →012. The first flip is a non-single. (+2)

Step 2b)(ii)(IV): For $0(21)^+2$, 3Sort will perform the first non-single: $[02]1(21)^*2\rightarrow201(21)^*2$. After several left singles, the resultant string will be 201212, which can be sorted in three flips (same flips as in Step (2b)(ii)(III), ignoring the last symbol 2 which is in the correct place) including a non-single. (+2)

Step 2b)(ii)(V): For the string $1(02)^+$, 3Sort will perform the first non-single: $[10]2(02)^*\rightarrow012(02)^*$. After several left singles, the resultant string will be 01202, which can be sorted in two flips [012]02→[210]2→ 012. (+1)

Step 2b)(ii)(VI): For the string $1(02)^+0$, 3Sort will perform the first non-single: $[10]2(02)^*0\rightarrow$ $012(02)^*0$. After several left singles, the resultant string will be 21020, which can be sorted in three flips [210]20→[012]0→[210]→012. Here, the last flip is a non-single. (+2)

Step 2b)(ii)(VII): For the string $1(20)^+$, 3Sort will perform the first non-single: $[12]0(20)^*\rightarrow$ $210(20)^*$. After several left singles, the resultant string will be 21020 which can be sorted in three flips [210]20→ [012]0→[210]→012 (+2); where the last flip is a non-single. (+2)

Step 2b)(ii)(VIII): If the string is $1(20)^+2$, 3Sort will perform: $[12]0(20)^*2\rightarrow$ $210(20)^*2$, the first non-single. After several left singles, the resultant string 01202 can be sorted with 2 left singles: [012]02→ [210]2→012 (+1). (+1)

In the above cases, the non-single which puts a '2' at the end is necessary for any algorithm. Hence it seems very likely that a detailed analysis would give an optimum algorithm clearly delineating the cases that need 0,1 and 2 non-singles respectively. In fact, Hurkens *et al.* [12] gave an algorithm that sorts ternary strings in the minimum number of flips.

2 Transforming strings

Let *S* and *T* be two strings of length *n* with a common suffix ξ. The suffix ξ need not be involved in the transformation of *S* into *T* since it is already a suffix of *T*. When there is a common suffix ξ, the problem of transforming *S* in to *T* reduces to the problem of transforming S' of length $n - |\xi|$ into T' of length $n - |\xi|$, where S' and T' are prefixes of *S* and *T* respectively. Hence the problem size is reduced by $|\xi|$, we call this size reduction.

BinTran(*S,T*) transforms *S* of length *n* into a compatible string *T* over the alphabet {a,b} in at most *n*–1 steps and is based on the size reduction described above. At each step, the algorithm will apply one of the six subcases, which exhaust all possibilities. The subcases are listed by priority (decreasing), and describe the flips to be applied to the string being transformed. At each step, the algorithm starts with the highest priority subcase, 1(a), and proceeds downward. A feasible subcase with highest priority will always be executed. For example, if subcases 1(a) and 2(b)(i) are both feasible then flip(s) of subcase 1(a) is (are) applied. Within a subcase, if more than one flip is feasible then the shortest prefix is reversed.

Let $b(X)$, $e(X)$, and $s(X,Y)$ be the beginning symbol of *X*, the ending symbol of *X* and the size of prefix of *X* that needs to transformed into a prefix of *Y* respectively. Without loss of generality let *T* end in 'a', (T = (…a)).

BinTran(*S,T*)

(1) If $b(S)$ =a then $s(S,T)$ is reduced by 1 in at most one flip.
   (a) If $e(S)$=a then *S* and *T* have a common suffix of length one and the problem size is reduced by 1 without making any flips.
   (b) If $b(S)$ =a, then the flip [S] will put 'a' at the end and $s(S,T)$ is reduced by 1 because of the common suffix.

(2) S = (b…b) and *T* =(…a). We begin by examining the second-to-last symbol of *T*.
   (a) Let *T* end with 'ba' although *S* = (b…b), *S* must have at least one a. Since $b(S)$=b, *S* has at least one 'ba'. So, S= (…ba…b). The flips ([…ba]…b)→([ab…b])

→(b…ba) reduce $s(S,T)$ by two in two flips, since a common suffix of size 2 is created.

(b) Let T end with 'aa', *i.e.* S = (b…b) and T = (…aa). Here we examine the first symbol of T.

(i) Let $b(T)$=b, *i.e.* S = (b…b) and T = (b…aa). The flip ([b…aa]) on T yields T' = (aa … b), where $e(T')=e(S)$. In one move T' transforms into T. That is, after S' transforms into T' one more move is required to obtain T. Thus in one move will reduce the problem size by one.

(ii) Let $b(T)$ =a, *i.e.* S = (b…b) and T = (a…aa). Examine whether aa ∈ S.

(I) 'aa'∈S (S = (b…aa…b)).Then the flips of S, ([b…aa]…b)→ ([aa…b...b])→ (b...b…aa), reduce $s(S,T)$ by two in two moves.

(II) 'aa' ∉ S, *i.e.* S = (b… …b) and T = (a…aa) and aa ∉ S. Here we examine the second-to-last symbol in S.

Let S end with 'ab'. Then S = (b…ab) and T = (a …aa). Since S has two b's and $b(T)$=a, T must have at least one ab. That is T = (…ab…aa), and in two flips ([…ab]…aa)→ ([ba…aa]) →(aa…ab), T matches the last two elements of S.

Thus, the only subcase left is: S = (bαbb) and T = (aβaa), where α and β ∈ {a,b}*, aa ∉ α and bb ∉β. If α has 'aa' then in two moves this 'aa' can be put at the end and match the last two symbols of β. Similarly if β has 'bb' in two moves the last two symbols of S can be matched. Let $\sum_u(\omega)$ denote the number of occurrences of u in ω. Since number of 'a's and the number of 'b's in S match those of T. We have the following conditions:

Condition 1: $\sum_a(\alpha) = \sum_a(\beta) + 3$.

Condition 2: $\sum_b(\beta) = \sum_b(\alpha) + 3$).

Condition 3: $aa \notin \alpha$ and

Condition 4: $bb \notin \beta$.

Conditions 3 and 4 impose restrictions on α and β. α will have one of the forms: (1) $(ab^+)^*(a)$? or (2) $\alpha = (b^+a)^*(b)^*$ and β will have one of the forms: (1) $(ba^+)^*(b)$? or (2)$\beta=(a^+b)^*(a)^*$. Therefore, we obtain the following conditions:

Condition 5: $\sum_b(\alpha) \geq \sum_a(\alpha) - 1$

Condition 6: $\sum_a(\beta) \geq \sum_b(\beta) - 1$

Conditions (1) and (6) yield $\sum_a(\alpha) \geq \sum_b(\beta) + 2$ and (2) and (5) yield $\sum_b(\beta) \geq \sum_a(\alpha) + 2$; which is a contradiction. So, the subcases 1(a)-2(b)(II)(a) cover all possibilities. Every subcase reduces $s(S,T)$ at least by the number of flips it executes. Once n–1 elements are matched the last element will be at the correct position. This yields an upper bound of n–1. Since it takes n–1 flips to transform $(01)^n$ to 1^n0^n, the bound is tight. ■

Algorithm to transform k-ary (k>2) strings

Hurkens *et al.* [14] gave the following algorithm to transform a given k-ary string into a given compatible k-ary string. The associated upper bound is $2(n-\alpha)$, where n is the length of the string and α is the frequency of the most frequent symbol 'a'. The algorithm matches the symbols from the back end, similar to our algorithm for binary strings and yields the same upper bound for binary strings.

Hurkens Algorithm: If $S_n=T_n$ then $S_1S_2...S_{n-1}$ and $T_1T_2...T_{n-1}$ are compatible strings of length n–1, the problem size reduces by 1, and the algorithm applies to the prefixes of length n–1 of S and T.

If $S_n \neq T_n$, there are two cases: (a) neither of the symbols is 'a', in this case, perform two flips on S, the first flip bringing $b=T_n$ to the front and the next flip putting 'b' at the n^{th} position of S, and (b) one of them is *'a'* and the other is not, in this case, perform two flips on the string that ends with *'a'* and match the n^{th} symbol of the other string.

Theorem 3. Let $S=S_1S_2...S_n$ and $T=T_1T_2...T_n$ be compatible k-ary strings and α be the frequency of the most frequent symbol 'a'. Then $d_{pr}(S,T) \leq 2(n-\alpha)$.

Proof: The bound holds trivially when n = 2, since at most one flip is needed. If $S_n = T_n$ = a then $S_1S_2S_3...S_{n-1}$ and $T_1T_2T_3...T_{n-1}$ are compatible strings of length n – 1 where 'a' occurs α – 1 times, the problem size reduces by 1. Thus by induction $d_{pr}(S,T) \leq 2\ (n-1-(\alpha-1)) = 2(n-\alpha)$. If $S_n = T_n \neq$ a, induction gives $d_{pr}(S,T) \leq 2(n-\alpha-1) = 2(n-\alpha)-2$. Suppose $S_n \neq T_n$, without loss of generality assume t_n = b (≠ a). Then in two flips of S, S_n can match T_n and the prefixes of length n–1 still have α 'a's. Hence by induction $d_{pr}(S,T) \leq 2 + 2((n-1)-\alpha) = 2(n-\alpha)$. ■

Chitturi *et al.* [3] gave an upper bound of 18n/11for permutations. Thus, Theorem 3 performs better than [3] if $2(n-\alpha) < 18n/11$, *i.e.* $k < 6$. The bounds given by Theorem 3 for alphabet of size two, three, four and five are n–1, 4n/3, 3n/2 and 8n/5 respectively. A string of length n over alphabet of size k can be converted into a permutation of length n by treating each occurrence of a symbol as a unique. This can be done in numerous ways. Given S = 123123 and T = 332211, one can map S to 123456 and T to 362514, where the repeated symbols are mapped to consecutively larger values. Thus, for k>5 one can use the method of [3].

Bounds for binary flip distance

Theorem 1 gives $b_{pr}(S,T) + b_{epr}(S,T) \leq d_{pr}(S,T)$ and Theorem 3 gives $d_{pr}(S,T) \leq 2(n-\alpha)$. Therefore, the bounds for binary flip distance are $b_{pr}(S,T) + b_{epr}(S,T) \leq d_{pr}(S,T) \leq 2(n-\alpha)$.

Approximation algorithm for sorting k-ary strings

In this section we describe a general approximation algorithm, KSort, to sort a string with k symbols by flips. We use the standard notation for representing strings. A *complete block* of x consists of all the 'x's in a given string. We maintain a table for all strings (with their optimum flips) whose length in standard notation is $L(\geq k)$. For small values of k this is feasible. For example, when sorting the strings involving human genes, k =4 and $L \geq 4$. If the alphabet size is large (yet less than 18) then we can restrict the size of strings in the table to k, *i.e.* we store the optimum flips on permutations of size k. The optimum flipping sequences are known for permutations of length up to 17. For permutations

of length >17, we instead use the algorithm given by Chitturi *et al*. [3] with an upper bound of 18*n*/11.

KSort(*S*)

Step1) Execute left singles until (a) *length*(*S*) in the standard notation is *L* or (b) a single does not exist.

Step2)

(a) If *length*(*S*)=*L*, then perform the flips given by the corresponding entry in the table and terminate.

(b) Here, the string starts with a complete block. Reverse the prefix that ends just before the symbols that are at the end and are complete blocks (default is the end of the string) and go to Step 1).

Analysis

Let the length of the string be *n*. Recall that the flip diameter for permutation of length *k* is denoted by *f*(*k*). Each single can reduces the size of the string by one. Therefore, a minimum of *n*-*k* flips are required to sort the string. We disregard the advantage obtained by maintaining a table whose member strings are longer than *k*. We assume that the table just has optimum flips for size *k* permutations.

If the input string executes 2(a) in the first iteration, then the number of flips executed is $n - k$ and the current length of string is *k*. Therefore, the maximum number of flips necessary for the string is $n–k + f(k)$.

Let α_x be a symbol of the alphabet $\sum$. Every execution of Step 2b) results in one non-single, and element say '*e'* (representing a complete block of *e*'s), is placed at the end of the string just before the other complete blocks. This element (block) will not be moved until Step 2a) is executed. The maximum number of times this can happen is *k*–2 and the string will be of the form $((\alpha_i\alpha_j)^+ (\alpha_i)^* \alpha_a\alpha_b\alpha_c\alpha_d\ldots)$. Here α_i and α_j are the only symbols that are not moved to the back of the string by move 2(b) (Note that some of the complete blocks at the end of the string might have formed independent of move 2(b).) This string after several left singles will yield $(\alpha_j\alpha_i\alpha_a\alpha_b\alpha_c\alpha_d\ldots)$ or $(\alpha_i\alpha_j\alpha_a\alpha_b\alpha_c\alpha_d\ldots)$. The size of the string thus obtained will be *k;* hence, there is no need to execute 2(b). This gives the worst case number of flips as $(n–k) + k–2+ f(k) = n–2 + f(k)$.

So, the approximation ratio is $(n–2+f(k))/(n–k) = 1+(k+f(k)–2)/(n–k)$. For large strings, $n >> k$ and $f(k)\leq 18k/11$ [3]. Therefore, the approximation ratio is close to 1.

3 Complexity

3.1 Prefix reversal distance for binary strings

Theorem 2 proved that the optimum binary sorting (a special case of transforming strings) requires either $B – 2$ flips or $B – 1$ flips depending on whether the string ends with a '1' or a '0'. In this section we study the complexity of flip distance between two arbitrary binary strings and show that it is NP-complete. Since binary strings are a subset of ternary strings, (in general a *k-ary* string is a special case of $(k+1)$-ary string) finding flip distance for a $(k+1)$-ary string is at least as hard as finding flip distance for *k-ary* string. This means that if we can show that the binary version of the problem is difficult, then any *k-ary* ($k>2$) version is also difficult.

First, we prove a weaker result. We show that flip distance between a binary string and a set of compatible strings is NP-complete. Then we show that $d_{pr}(S,T)$ for strings is NP-complete. Garey and Johnson [9] proved that the 3-Partition problem is strongly NP-complete. Thus, if 3-Partition reduces to a problem P by a pseudo polynomial reduction then P is NP-hard. The definition of 3-Partition is given below.

3-Partition

Instance: Given a set $A = \{ b_1, b_2,\ldots, b_{3m} \}$, a positive integer bound *B*, and a positive integer size $s(b_i)= a_i$, for each b_i in *A*, such that $B/4 < a_i < B/2$ and $_{(bi\in A)}\Sigma\ s(b_i) = mB$.

Question: Can A be partitioned into *m* disjoint sets $A_1,A_2,\ldots,A_m$ such that for all i $(1 \leq i \leq m)$, $_{(b\in Ai)}\Sigma\ s(b)=B$?. That is, the sum of the sizes of elements in set A_i is equal to *B*.

3.1.1 BPRS is NP-complete

Here, flip distance between a string and a set of compatible binary strings is shown to be NP-complete. We define this problem as a decision problem, the *binary pre*fix *reversal distance problem from a string to a set* (BPRS).

BPRS

Instance: A binary string *S* and *T* (a set of strings compatible with S) of a given length and a positive integer bound *d.*

Question: Is $d_{pr}(S,T) \leq d$?

Theorem 4. *BPRS is NP-complete.*

Proof: Given a source string S, a target (set of strings) *T*, and a sequence of flips of a given length ($O(|S|)$), verifying that the given sequence of flips yields a member of *T* can be done in polynomial time. Therefore, BPRS is in NP.

Let I be an instance of 3–*Partition* consisting of the integer bound *B*, $A = \{ b_1, b_2,\ldots, b_{3m} \}$, and for all the indices '*j*' the value $s(b_j) = a_j$. From I we construct an instance I* of BPRS as described hereunder. Note that $s(b_i) = a_i$ is the value of an element in 3–*Partition* problem and in I* it is converted into an input parameter whose length is a_i (pseudo polynomial reduction). Let I* be an instance of BPRS where

$S = 0^{a_1}10^{a_2}..10^{a_{3m-1}}10^{a_{3m}}$,

$T = (0^B1^+)^m$, and

$d = 4m – 1$.

Each 0 block in *S* represents an element b_i of 3-*Partition* and the number of '0's equals $s(b_i) = a_i$. We show that 4m–1 flips are necessary. First we note that $o(S) = 3m–1$ and $z(S) =3m$ also $o(T)=z(T)=m$. So, $o(S)+ z(S)–(o(T) + z(T)) = 3m – 1 + 3m – (m + m) = 6m – 1 – 2m = 4m – 1$. So, $4m–1$ flips are required to transform *S* into *T*. We complete the proof the theorem by showing that (a) if *S* can be transformed into *T* in 4m–1 flips then 3-Partition has solution, and (b) if 3-*Partition* has solution, *S* can be transformed into a member of *T* in $4m–1$ flips.

First we show that Condition (a) holds. Since *S* is transformed into a member of *T* in minimum possible moves, no 0-block can be split. Note that for all values of i, $a_i > B/4$ and a_i is an integer. Thus, any four blocks of zeros put

together will form a block which has at least B+4 zeros. Also, if two 0 blocks of S and one '10' are merged together with flips, then the total number of zeros they can produce is at most $B - 1$. So, each 0 block in T is formed from exactly three 0 blocks of S. Thus, we can list the moves made by BPRS and note which 0 blocks corresponding to which three a_i's are combined together, *i.e.* $\{a_{i1},a_{i2},a_{i3}\}$. The number m of such triples give the solution for 3-Partition.

A simple algorithm meets Condition (b) but it is omitted due to lack of space ■

A member of T from Theorem 4 can be transformed into a string U, where $U = 1^{2m-1}(10^B)^m$ in at most $2m - 2$ flips. The following flips achieve this transformation.

$[1^+0^B..1^+0^B]1^+0^B \rightarrow 0^B1^+..0^B1^+10^B$

$[0^B1^+..0^B1^+]10^B \rightarrow 1^+0^B..1^+0^B10^B$

$[1^+0^B..0^B]1^+0^B10^B \rightarrow 0^B..0^B1^+0^B10^B$

$[0^B..0^B1^+]10^B10^B \rightarrow 1^+0^B..1^+0^B(10^B)^2$ etc.

Thus, S from the Theorem 4 can be transformed into U in $4m-1 + 2m - 2 = 6m - 3$ flips. A reduction from 3-*Partition* to the problem of transforming S to U in $6m-3$ flips, where U has $4m-1$ different adjacencies compared to S is plausible but the necessity of the additional $(6m-3)-(4m-1) = 2m-2$ flips should be justified. This is possible only if none of the singleton '1's of S can be directly used in forming the blocks of 10^B in U; *i.e.* all the original singleton '1's should become larger blocks (*i.e.* 11^+) first and then they need to be broken. This cannot be justified. To overcome this issue, in Section 3.1.2 we modify the source string such that all the singleton '1's are changed into blocks of size >1. In fact this is necessary and sufficient condition for this reduction to work. Based on the total number of 1s present in S, the length of the leading 1 block of U will change. This modification forces one to (a) form singleton '1's, and (b) reduce the '1' blocks of length greater than one. These two tasks are independent.

3.1.2 PRD is NP-complete

In this section we define the flip distance between binary strings as a decision problem, the prefix reversal distance problem, *PRD*. We reduce 3-*Partition* to PRD.

Prefix reversal distance problem (PRD)
Instance: Binary strings S and T of a given length and a positive integer bound d.
Question: Is $d_{pr}(S,T) \le d$?

Theorem 5. PRD is NP-complete.

Proof: PRD is in NP because for a solution sequence of a given length, the flips can be executed in polynomial time. Then one can verify in polynomial time whether the resultant string is the target string (T) or not.

Let I be an instance of 3-*Partition* from which we construct an instance I* of PRD as described here. The reduction is pseudo polynomial. Let I* be an instance where

$S = 0^{a_1}1^20^{a_2}..1^20^{a_{3m-1}}1^20^{a_{3m}}$,

$T = 1^{5m-2}(10^B)^m$ and

$d = 6m - 3$.

Each 0 block in S represents an element b_i of 3-*Partition* and the number of '0's equals $s(b_i) = a_i$. In S the size of 1 blocks is two, however, this reduction works for any size greater than one and correspondingly the size of the leading 1 block of T changes. We also note that all the 1 blocks need not be of the same size. We show that $6m - 3$ flips are necessary. Depending on the sizes of 1 blocks present in S, correspondingly the size of the leading 1 block of T changes.

Let the 1 blocks whose length is greater than one be *big blocks*. In transforming S into T any sequence of moves must accomplish the following: (a) reduce the number of big blocks by $3m - 2$, (b) reduce the number of 0 blocks by $2m$, and (c) create $m - 1$ singleton '1's. Let (a), (b), and (c) be *compound tasks*. A task that defines one unit of the compound task is *unit task*. Therefore, task (a) has $3m-2$ unit tasks, task (b) has $2m$ unit tasks, and task (c) has m – 1 unit tasks. These tasks combined can be accomplished in $(3m - 2) + 2m + (m - 1) = 6m - 3$ flips. If we show that these tasks are independent then indeed $6m - 3$ flips are necessary. That is, if we can show that performing one task does not contribute towards another task, $6m - 3$ flips are required.

Earlier we showed that a flip can increase or decrease the number of blocks by at most one. Consider the flips that can accomplish each of the unit tasks: Unit task (a) is accomplished by (a1) ($[1^+..0]1^+\ldots$) which reduces exactly one big block, or (a2) ($[1^+..01]1^+...$) which reduces one big block and creates a singleton 1. Unit task (b) is accomplished only by the flip ($[0...1]0^+\ldots$) and reduces exactly one 0 block. Unit task(c) is accomplished by either (c1) ($[0\ldots01]11^+0\ldots$) which creates exactly one singleton 1, or by (c2) ($[0\ldots01]10\ldots$) which creates two singleton '1's, or by (c3) ($[0\ldots11^+]10\ldots$) which creates one singleton 1. Unit task (b) decreases '0' blocks by one and cannot alter the '1' blocks. Clearly unit task (b) is independent of unit tasks (a) and (c).

If a '0' block is broken then an additional unit task (b) is required. If a '1' block is broken into two big blocks, it does not contribute to any of the tasks. Now we should establish independence of unit tasks (a) and (c). The flip (a1) does not contribute towards unit task (c). The flip (a2), *i.e.* $([1^+..01]1^+...)\rightarrow(10\ldots1^+\ldots)$, reduces one big block (unit task (a)) and also creates a singleton (unit task (c)), that comes to the front of the string. Consider any flip on the resultant string $(10\ldots1^+\ldots)$. This flip can either (i) join the leading singleton '1' to another '1' block (size> 2) and hence the singleton ceases to exist, or (ii) flip the entire string bringing the leading singleton '1' to the end of the string, or (iii) perform $([10\ldots1]0\ldots)$. Note that, if a flip joins the singleton '1' at the beginning of the string to another singleton '1' then in total two singleton 1's are lost and hence that move is not considered above. If flip (i) is executed then in 2 consecutive flips only one unit task (a) is accomplished. If flip (ii) is executed then in 2 consecutive flips only one unit task (a) is accomplished (T does not end in a '1'). If flip (iii) is executed then the first flip accomplishes unit task (a) and the second flip accomplishes unit task (c). Thus, unit task (a) does not contribute towards unit task (c).

Because of the above reasoning, in order to accomplish unit task (c), flips (c1) and (c2) cannot be used. The flip (c3) breaks a '1' block and hence cannot contribute towards unit task (a). Therefore, all the unit tasks are independent and hence $6m - 3$ flips are necessary.

We complete the proof of this theorem by showing that (a) if S can be transformed into T in $6m–3$ flips then 3-*Partition* has solution, and (b) if 3-*Partition* has solution, S can be transformed into T in $6m–3$ flips.

First we show that condition (a) is true. Since S is transformed into T in the minimum possible moves, no '0' block can be split. Notice that for all values of i, $a_i > B/4$ and a_i is an integer. Therefore, any four blocks of zeros put together will form a block which has at least $B+4$ zeros. Also, if two '0' blocks of S and one '10' are merged together with prefix transpositions, then the total number of zeros they can produce is at most $B - 1$. Therefore, each 0 block in T is formed from exactly three 0 blocks of S. So, we can list the moves made by PRD and note which 0 blocks corresponding to which three a_i's are combined together, *i.e.* $\{a_{i1}, a_{i2}, a_{i3}\}$. The number m of such triples give the solution for 3–*Partition.*

The condition (b) specifies that if 3-*Partition* has solution then S can be transformed into T in $6m–3$ flips. Let the sum $a_{j1}+a_{j2}+a_{j3}$ be B. We define the blocks $0^{a_{j1}}$, $0^{a_{j2}}$, and $0^{a_{j3}}$ as *components* and they are *partners* of each other. In the following pairs of blocks the first block and the second block are also partners of each other: $\{0^{a_{j1}}, 0^{a_{j2}+a_{j3}}\}$, $\{0^{a_{j2}}, 0^{a_{j1}+a_{j3}}\}$ and $\{0^{a_{j3}}, 0^{a_{j1}+a_{j2}}\}$. Let a block of B '0's be a *complete component* and a block with two components joined be a *semi-complete component*. Let a 0 block with less than B '0's be an *incomplete component*. Let 10^B be a *gadget*. *PRD*(S,T) forms the complete components (and gadgets) from the rightmost end and transforms S into T in $6m - 3$ flips.

PRD(S,T)

1. Let a_{11} and a_{12} be partners. Execute the flip:
   $[0^{a_{11}}1^2..1^2]0^{a_{12}}..1^20^{a_{3m-1}}1^20^{a_{3m}} \rightarrow 1^2..1^20^{a_{11}+a_{12}}..1^20^{a_{3m-1}}1^20^{a_{3m}}$
   (This flip joins two components).
2. (a) In the next two flips bring the leftmost partner of $0^{a_{3m}}$ (a component or the semi-complete component ($0^{a_{11}+a_{12}}$)) to the front and then join it with its leftmost partner; this takes two flips. If a_{11}, a_{12} and a_{3m} are partners (*i.e.* $a_{13}= a_{3m}$) then: (I) three flips are consumed, reducing 0 blocks by two and 1 blocks by one, (II) one complete component is formed, and (III) the sub step 2(b) is not necessary, proceed to step 3.
   (b) In the next two flips bring the semi-complete component just formed in 2)(a) to the front and then join it with $0^{a_{3m}}$ forming a complete block preceded by 1 block. The resulting string is of the form $1^2..0^{a_{11}+a_{12}}..1^+10^B$.
   After executing steps 1 and 2, there are two situations, either (S1) the total number of blocks reduced is five, *i.e.* three 0 blocks and two 1 blocks and five flips are executed (steps 1, sub steps 2(a) and 2(b)), or (S2) the total number of blocks reduced is three, *i.e.* two 0 blocks and one 1 block and three flips executed (step 1 and step 2(a)).
3. Repeat the following for m–1 iterations.
   (a) In two flips bring the leftmost incomplete component to the front and join it to its partner. Then execute (b1) or (b2) based on whether the 0 block just formed is complete or incomplete. When (b1) is executed, step 3 consumes four flips per iteration; otherwise step 3 will consume six flips per iteration.
   (b1) If the 0 block formed in (a) is a complete component then perform the following two flips. In the first flip bring the complete component just formed to the front, and then execute the flip $[0^B11^+..1^+](10^B)^k \rightarrow 1^+..11^+(10^B)^{k+1}$. If $a_{13} \neq a_{3m}$ (see step 2) and $0^{a_{11}+a_{12}}$ is the leftmost incomplete component of the above step only then this step is executed.
   (b2) If the block thus formed in (a) is incomplete, then in two flips join it with its final partner. The two subsequent flips accomplish the following tasks. The first flip brings the complete component just formed to the front and the second one is $[0^B11^+..1^+](10^B)^k \rightarrow 1^+..11^+(10^B)^{k+1}$ which puts the currently formed gadget just in front of the remaining gadgets.

The final flips of steps 3(b1) and 3(b2) are the only flips that create a singleton 1 and these flips together are performed exactly m–1 times, corresponding to the last m–1 gadgets. Step 3 involves (3(a), 3(b1)) which executes four flips, or (3(a), 3(b2)) which executes six flips. The sequence (3(a), 3(b1)) is executed iff (a_{11}, a_{12}, a_{3m}) are not partners of each other. Thus the total number of flips performed by the algorithm are $3+(m–1)6 = 6m–3$ or $5 +(m–2)6 + 4 = 6m–3$. So, in m–1 iterations m–1 gadgets are formed in Step 3) and all the remaining '1's are grouped at the front of the string. Steps 1) and 2) form the first gadget.

Proof of correctness of the algorithm

The string will be of the form $1^2..0^{a_{11}+a_{12}}..1^+10^B$ after steps 1) and 2). By the design of the algorithm, after every iteration of the step 3) the string will be of the form $1^+..11^+(10^B)^{k+1}$. After $3m$–2 iterations of step 3) the string will be of the form $1^w0^p1^x0^q1^y0^r1^z1(10^B)^{m-1}$, where $p+q+r = B$ and $w+x+y+z+m = 6m–2$.

The following sequence constitutes the final flips:

$[1^w0^p]1^x0^q1^y0^r1^z1(10^B)^{m-1} \rightarrow 0^p1^{x+w}0^q1^y0^r1^z1(10^B)^{m-1}$

$[0^p1^{x+w}]0^q1^y0^r1^z1(10^B)^{m-1} \rightarrow 1^{x+w}0^{q+p}1^y0^r1^z1(10^B)^{m-1}$

$[1^{x+w}0^{q+p}]1^y0^r1^z1(10^B)^{m-1} \rightarrow 0^{q+p}1^{w+x+y}0^r1^z1(10^B)^{m-1}$

$[0^{q+p}1^{w+x+y}]0^r1^z1(10^B)^{m-1} \rightarrow 1^{w+x+y}0^{p+q+r}1^z1(10^B)^{m-1}$

$[1^{w+x+y}0^B]1^z1(10^B)^{m-1} \rightarrow 0^B1^{x+y+w+z}1(10^B)^{m-1}$

$[0^B1^{x+y+w+z+1}](10^B)^{m-1} \rightarrow 1^{x+y+w+z}(10^B)^m$.

Because w+x+y+z+m = 6*m*–2, we have w+x+y+z = 5*m*–2. Thus, we obtain the target string *T* in 6*m* – 3 flips. Parts (a) and (b) together prove that PRD is NP-complete. ■

4 On approximating flip distance

An approximation algorithm for $d_{pr}(S,T)$ is sought because $d_{pr}(S,T)$ is NP-complete (Theorem 5). First, we survey the relevant work. Goldstein *et al.* [11] showed that Minimum Common String Partition Problem (MCSP) and signed MCSP (SMCSP) are NP-hard if the symbols repeat two or more times. Chen *et al.* [1] defined a related problem, minimum common partition problem (MCP). For a given pair of signed strings *S* and *T* and their respective partitions $S^{\,p}$ and $T^{\,p}$, the pair ($S^{\,p}$, $T^{\,p}$) constitutes a MCP for *S* and *T* if there is a bijection from $S^{\,p}$ to $T^{\,p}$ such that if $S_i \in S^{\,p}$ is mapped to $T_j \in T^{\,p}$ then $S_i = T_j$ or $S_i = T_j^R$, (thus, applicable for flips) and $|S^{\,p}|$ (or $|T^{\,p}|$) is minimum (In MCSP, $S_i = T_j^R$, is not allowed). For two strings *S* = 123123 and *T* = 321321 MCSP yields a set of six singletons, however, the size of MCP is one, *i.e.* S_1 = 123123, and T_1 = 321321 (=$(S_1)^R$). One flip [123123]→ 321321 transforms *S* into *T*. The maximum independent set problem $\leq_p$ *MCP* [1]. Thus, MCP is NP-hard. Chitturi independently proved that deciding whether a given pair of strings has MCP of size *k* is NP-complete [2].

Given a MCP ($S^{\,p}$,$T^{\,p}$) of size *k* of (*S*,*T*) any *optimum* algorithm must make *k*–1 new adjacencies to transform *S* into *T*. Each $S_i \in S^{\,p}$ ($T_i \in T^{\,p}$) of MCP is an object. A flip can make at most one new adjacency; so ≥ *k*–1 flips are needed. Because each object S_i (T_i) is a string one needs to employ $d_{spr}(S^{\,p},T^{\,p})$, where d_{spr} is the signed flip distance. A PTAS with a ratio of 2 by Cohen and Blum [6] for $d_{spr}(S,T)$ for permutations of length *n* has an upper bound of 2*n*–2. Thus, $d_{spr}(S^{\,p},T^{\,p}) \leq 2k–2$. So, MCP of *S*, *T* yields a PTAS for flips over strings with a ratio of (2*k*–2)/(*k*–1) =2. A PTAS for MCP with a ratio of *r* yields the same with a ratio of 2*r* for $d_{pr}(S,T)$. Cormode and Muthukrishnan [7] gave an $O(log\ n\ log^*\ n)$ approximation algorithm for String Edit Distance Matching Problem with Moves consisting of: (insert a character, delete a character, move a substring). Shapira and Storer showed that eliminating the first two operations changes the edit distance by at most a constant multiplicative factor [16]. Thus, [7] gives O(log *n* $\log^*$ *n*) approximation algorithm for MCSP. Similar results for MCP will yield an approximation algorithm for flip distance over strings. For permutations, a PTAS with a ratio of two exists [8].

5 Conclusions

We present upper and lower bounds for transforming strings with flips. In sorting, 2Sort is optimum for binary strings, 3Sort needs minimum+2 flips for ternary strings, and KSort needs minimum+$f(k)$ flips for *k*-ary (*k*>3) strings where $f(k) \leq k–2+18k/11$. This provides strong evidence that sorting is a special (easier) case of transforming strings. An algorithm to optimally sort any *k*-ary strings, a PTAS for $d_{pr}(S,T)$, and a PTAS for MCP remain open problems. Another open problem is the signed flips on strings.

Acknowledgements

I thank my colleague Walter E. Voit for his input on sorting ternary strings with flips.

6 References

[1] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi and T. Jiang. Assignment of orthologous genes via genome rearrangement. IEEE/ACM Transactions on Computational Biology and Bioinformatics (2005) 2(4):302-315.

[2] B. Chitturi. On transforming sequences. Ph.D. thesis, Department of Computer Science, University of Texas at Dallas, 2007.

[3] B. Chitturi, W. Fahle, Z. Meng, L. Morales, C. Shields, I. H. Sudborough and W. Voit. An (18n/11) upper bound for sorting by prefix reversals. Theoretical Computer Science (2009) 410(36) :3372-3390.

[4] B. Chitturi and H. Sudborough. Bounding Prefix Transposition Distance for Strings and Permutations. HICSS 2008, pp. 469-478.

[5] D. A. Christie and R. W. Irving. Sorting Strings by Reversals and Transpositions. SIAM Journal on Discrete Mathematics (2001) 14(2):193-206.

[6] David S. Cohen and Manuel Blum. On the Problem of Sorting Burnt Pancakes. Discrete Applied Mathematics (1995) 16(2):105-120.

[7] G. Cormode and S. Muthukrishnan. The String Edit Distance Matching Problem with Moves. SODA 2002, pp. 667-676.

[8] J. Fischer and S. Ginzinger. A 2-Approximation Algorithm for Sorting by Prefix Reversals. LNCS 3669, pp. 415-425, 2005.

[9] M. R. Garey and D. S. Johnson. "Strong" NP-Completeness Results: Motivation, Examples, and Implications. Journal of ACM, (1978) 25(3): 499-508.

[10] W. H. Gates and C. H. Papadimitriou. Bounds for Sorting by Prefix Reversal. Discrete Mathematics, v. 27, pp. 47-57, 1979.

[11] A. Goldstein, P. Kolman and J. Zheng. Minimum Common String Partition Problem: Hardness and Approximations. LNCS 3341, pp. 484-495, 2004.

[12] M. H. Heydari. The Pancake Problem. Ph.D. thesis, Department of Computer Science, University of Texas at Dallas, 1993.

[13] M. H. Heydari and I. H. Sudborough. On the Diameter of the Pancake Network. Journal of Algorithms, v. 25, pp. 67-94, 1997.

[14] C. Hurkens, L. Iersel, J. Keijsper, *S.* Kelk, L. Stougie and J. Tromp. Prefix Reversals on Binary and Ternary Strings. SIAM Journal on Discrete Mathematics (2007) 21(3):592-611.

[15] A. J. Radcliffe, A. D. Scott and E .L. Wilmer. Reversals and Transpositions over Finite Alphabets. SIAM Journal on Discrete Mathematics (2005) 19(1):224-244.

[16] D. Shapira and J. A. Storer. Edit Distance with Move operations. LNCS 2373, pp. 85-98, 2002.