The 2-approximation algorithm for sorting by prefix transposition revisited

Md. Shafiqul Islam
Bangladesh University of Engineering and Technology
Email: s.tushar053@gmail.com

Md. Khaledur Rahman
Bangladesh University of Engineering and Technology
Email: khaled_cse_07@yahoo.com

M. Sohel Rahman
Bangladesh University of Engineering and Technology
Email: sohel.kcl@gmail.com

***Abstract*—A transposition is an operation that exchanges two adjacent blocks in a permutation. A prefix transposition always moves a prefix of the permutation to another location. In this article, we use a data structure, called the permutation tree, to improve the running time of the best known approximation algorithm (with approximation ratio 2) for sorting a permutation by prefix transpositions. By using the permutation tree, we improve the running time of the 2-approximation algorithm to $O(n \log n)$.**

I. Introduction

One of the most promising ways to trace evolutionary events is to compare the order of appearance of identical genes in two different genomes. In the 1980s, evidence was found that different species have essentially the same set of genes, but their order may differ between species [Palm er and Herbon 1986; Hoot and Palmer 1994]. This suggests that global rearrangement events such as *reversal, transposition*, and *block interchange* can be used to trace the evolutionary path between genomes. Such rare events may provide more accurate clues to evolution than local mutations (i.e., *insertions, deletions*, and *substitutions* of nucleotides).

In the last decade, a large body of work was devoted to genome rearrangement problems. The basic task here is as follows. Given two permutations, find a shortest sequence of rearrangement operations that transforms one given permutation into the other. Assuming that one of the permutations is the identity permutation, the problem is to find the shortest way of sorting a permutation using specific given rearrangement operations.

The best studied rearrangement event is the reversal. A reversal inverts a block of any size in a genome. Caprara proved that finding the minimum number of reversals needed to transform one genome into another is an NP-Hard problem **[8]**. Bafna and Pevzner presented an algorithm with approximation ratio 2 for this problem **[9]**. Later Berman, Hannenhalli and Karpinski presented the best known algorithm for the problem, with approximation ratio 1.375 **[10]**.

Another interesting variation of the problem involves applying prefix reversal as the gnome rearrangement operator. In this problem, also known as the pancake flipping problem in the literature, only reversals involving the first consecutive elements of a genome are permitted. Gates and Papadimitriou **[12]** and Heydari and Sudborough **[13]** have studied the diameter of prefix reversals.

Another interesting and much studied rearrangement event is transposition. Transposition refers to the event of exchanging two adjacent blocks of any size in a genome. The transposition distance problem, that is, the problem of finding the minimum number of transpositions necessary to transform one genome into another, has been studied by Bafna and Pevzner **[5]**. After a long-standing 15-years of being unclassified this problem has very recently been classified to be NP-Hard in **[15]**. The best known algorithm for sorting a permutation by transposition has an approximation ratio 1.375 with running time $O(n \log n)$ **[3] [14]**.

Like the prefix reversal, prefix transposition has also been given some attention in the literature. Dias, Fortuna and Meidanis presented an algorithm to sort a permutation by prefix transposition having approximation ratio 2 **[2]**. They also presented an algorithm to sort the reverse permutation, $R_n = [n, n-1, ..., 3, 2, 1]$ with $n - \lfloor \frac{n}{4} \rfloor$ prefix transpositions.

In the literature, the problems of sorting by different rearrangement operations have been tackled from at least to different directions. In one direction, researchers have been trying to improve the approximation ratio and in the other, efforts have been made to improve the running time of the algorithm keeping the approximation ratio intact. In this paper, we are concerned with the latter direction. In **[2]**, the authors did not formally present and analyze their 2-approximation algorithm for the problem of sorting by prefix transposition. If we implement this algorithm without any sophisticated data structure, the running time becomes $O(n^2)$ in the worst case. In this paper, we use an efficient data structure called permutation tree to sort a permutation by prefix transpositions. The motivation of our approach comes from a recent work of Firoz et al **[3]** where permutation tree has been used to achieve a faster running time for the best known approximation algorithm for sorting by transpositions. Using the permutation tree, in this paper, we present an $O(n \log n)$ time implementation of the 2-approximation algorithm of **[2]** for sorting by prefix transposition.

The rest of the paper is organized as follows. In Section II, we define important concepts that will be used throughout the paper. In Section III, we briefly review the permutation tree data structure. In Section IV, we give several results for

2-approximation algorithm of **[2]**. Finally, we briefly conclude in Section VI.

II. Preliminaries

In this section, we introduce and define some notions and definitions that will be used throughout the paper. An arbitrary genome formed by n genes will be represented as a permutation $\pi = [\pi(1), \pi(2), ..., \pi(n)]$ where each element of π represents a gene. The identity genome t_n is defined as $t_n = [1, 2, 3, ...n]$. A transposition $\tau(x, y, z)$, where $1 \leq x < y < z \leq n + 1$, is a rearrangement event that transforms π into the genome $\pi\tau = [\pi(1), ..., \pi(x - 1), \pi(y), ..., \pi(z - 1), \pi(x), ..., \pi(y - 1), \pi(z), ..., \pi(n)]$. In case of prefix transposition, $x = 1$ in $\tau(x, y, z)$. Given two genomes π and σ we define the prefix transposition distance $d_p(\pi, \sigma)$ between these two genomes as being the least number of prefix transpositions needed to transform π into σ, that is, the smallest r such that there are prefix transpositions $\tau_1, \tau_2, ..., \tau_r$ with $\tau_r ... \tau_2 \tau_1 \pi = \sigma$. We call sorting distance by prefix transpositions, $d_p(\pi)$, the prefix transposition distance between the genomes π and t_n, that is, $d_p(\pi) = d_p(\pi, t_n)$.

A breakpoint for the prefix transposition problem is a position i of a permutation π such that $\pi(i) - \pi(i - 1) \neq 1$, where $2 \leq i \leq n$. By definition, position 1 (beginning of the permutation) is always considered a breakpoint. Position $n+1$ (end of the permutation) is considered a breakpoint when $\pi(n) = n$. We denote by $b_p(\pi)$ the number of breakpoints of a permutation π. By definition $b_p(\pi) \geq 1$ for any permutation π and the only permutations with exactly one breakpoint are the identity permutations (t_n , for all $n \geq 1$). It is easy to see that there are no permutations with exactly two breakpoints. A strip is a subsequence $\pi(i...j)$ of π where $i \leq j$ such that i and $j + 1$ are breakpoints and there are no breakpoints between positions i and j. Given a permutation π and a prefix transposition τ, we use $\Delta b_p(\pi, \tau)$ to denote the change on the number of breakpoints due to operation τ, that is, $\Delta b_p(\pi, \tau) = b_p(\tau\pi) - b_p(\pi)$.

III. Permutation Tree

A permutation tree is firstly a balanced binary tree T with root r, where each internal node of T has two children. Let t be a node of T. The left and right children of t are denoted as $L(t)$ and $R(t)$, respectively. The height of a leaf node is defined to be zero. The height of an internal node t, denoted $H(t)$, is calculated as follows: $H(t) = \max(H(L(t)), H(R(t)) + 1$. Moreover, the tree must have the property of balance, that is, for any node t of T, $|H(L(t)) - H(R(t))| \leq 1$. The height of T is defined to be the height of the root, that is, $H(T) = H(r)$.

Secondly, a permutation tree must correspond to a permutation. For the permutation $\pi = [\pi(1), \pi(2), ..., \pi(n)]$, the permutation tree corresponding to π has n leaf nodes, which are labeled by $\pi(1), \pi(2), ..., \pi(n)$, respectively. Each node of T corresponds to an interval of π and is labeled by the maximum number in the interval. For any internal node t of T, the interval corresponding to t must be the concatenation of the two intervals corresponding to $L(t)$ and $R(t)$. The number labeled to t is called the value of t. Clearly, the value of node t must be the maximum value of $L(t)$ and $R(t)$. In the following, we may directly use the element $\pi(i)$ to represent the leaf node labeled by $\pi(i)$ and use the node t of T to represent the subtree of T rooted at t. As an example, Figure 1 is a permutation tree corresponding to the permutation $\pi = [9, 6, 1, 4, 7, 5, 2, 3, 8]$.

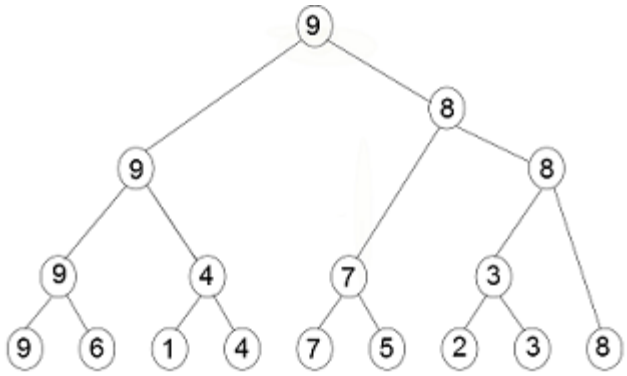

Fig. 1. A permutation tree for $\pi = [9, 6, 1, 4, 7, 5, 2, 3, 8]$

The distinctive property of the permutation tree is that one tree represents a permutation, and every node of the tree represents an interval of the permutation. In addition, we do not need to insert or delete keys in the permutation tree. Feng and Zhu introduced three operations for a permutation tree **[1]**. These are *Build*, which builds a permutation tree corresponding to a given permutation, *Join*, which joins two trees into one, and *Split*, which splits one tree into two. The following results on permutation tree will be used later.

Theorem III.1 ([1]). *The height of the permutation tree corresponding to* $\pi = [\pi(1), \pi(2), ..., \pi(n)]$ *is bounded by* $O(\log n)$.

Theorem III.2 ([1]). *Build always creates a permutation tree corresponding to a given permutation of* $[1, ..., n]$ *in* $O(n)$ *time.*

Theorem III.3 ([1]). *If* t_1 *corresponds to* $1, ..., m$, *and* t_2 *corresponds to* $m + 1, ..., n$, *then Join*(t_1, t_2) *returns a permutation tree corresponding to* $1, ..., m, m + 1, ..., n$ *in* $O(H(t_1) - H(t_2))$ *time.*

Theorem III.4 ([1]). *Suppose* T *is the permutation tree corresponding to* $\rho = [\pi(1), ..., \pi(m - 1), \pi(m), ..., \pi(n)]$. *The function Split*$(T, m)$ *always returns* T_l *corresponding to* $\rho_\ell = [\pi(1), ..., \pi(m - 1)]$, *and* T_r *corresponding to* $\rho_r = [\pi(m), ..., \pi(n)]$ *in* $O(\log n)$ *time.*

IV. 2-Approximation Algorithm of [2]

The following results are due to **[2]** and lay the foundation for the 2-approximation algorithm for sorting by prefix transposition.

Lemma IV.1 ([2]). *Given a permutation $\pi \neq t_n$, where $n = |\pi|$, it is always possible to obtain a prefix transposition τ such that $\Delta b_p(\pi, \tau) \leq -1$.*

Lemma IV.2 ([2]). *Let π be a permutation with $b_p(\pi) = 3$, then $d_p(\pi) = 1$.*

Lemma IV.3 ([2]). *For every permutation π different from the identity permutation we have $d_p(\pi) \leq b_p(\pi) - 2$.*

Lemma IV.4 ([2]). *For every permutation π different from the identity permutation we have $\lceil \frac{b_p(\pi)-1}{2} \rceil \leq d_p(\pi) \leq b_p(\pi) - 2$*

Theorem IV.5 ([2]). *Any algorithm that produces the prefix transpositions according to Lemmas IV.1 and IV.2 is an approximation algorithm with factor 2 for the prefix transposition distance problem.*

V. Implementation

In this section, we discuss possible implementations of the algorithm of **[2]**. We first discuss the implementation without any sophisticated data structure and deduce the worst case time complexity of the implementation in Section V-A. Then, in Section V-B we show how permutation tree can be employed to get a time complexity of $O(n \log n)$ for the same algorithm.

A. Implementation Without Permutation Tree

We use an 1-D array to hold the permutation and perform prefix transpositions on that array. For each prefix transposition, first we find the last element of the first strip as k and then, find i and j, where $i = \pi^{-1}(k) + 1$ and $j = \pi^{-1}(k + 1)$. Then, interchange block $[\pi(1), ..., \pi(i - 1)]$ and $[\pi(i), ..., \pi(j - 1)]$ by using a simple *loop*. This *loop* takes $O(n)$ time. According to *Lemma IV.1*, each such prefix transposition reduces the number of break points from the permutation by at least one. So, total number of prefix transpositions needed is at most n. As a result, time complexity of this approach is $O(n^2)$.

B. Implementation With Permutation Tree

In *Algorithm* 1 we present how to perform a prefix transposition operation using a permutation tree. In this algorithm, the procedure $Split(T, i)$ receives a permutation tree T and an integer i, and returns two permutation trees $T1$ and $T2$, that represent the permutations $[\pi(1), \pi(2), ..., \pi(i - 1)]$ and $[\pi(i), \pi(i + 1), ..., \pi(n)]$ respectively; $Join(T1, T2)$ receives two permutation trees $T1, T2$ representing permutations $[\pi(1), \pi(2), ..., \pi(i-1)]$ and $[\pi(i), \pi(i+1), ..., \pi(n)]$ respectively, and returns a permutation tree T that represents $[\pi(1), \pi(2), ..., \pi(i - 1), \pi(i), ..., \pi(n)]$. Then we present *Algorithm* 2 to perform the actual sorting of a permutation by prefix transpositions. The procedure $Build(\pi)$ used in *Algorithm* 2 receives a permutation π and returns a permutation tree. $\pi^{-1}(k)$ denotes the index position of element k in the permutation π. This algorithm runs until π becomes a permutaion of sorted list i,e identity permutaton. In the rest of this section, we state and prove a number of *Lemmas* concerning the running time of *Algorithm* 1 and *Algorithm* 2, achieving an $O(n \log n)$ running time of the overall algorithm in the sequel.

Algorithm 1 PREFIX-TRANSPOSITION(T, i, j)

1: $(T1, T2) := Split(T, i)$
2: $(T3, T4) := Split(T2, j - i + 1)$
3: $RETURN Join(Join(T3, T1), T4)$

Algorithm 2 SORTING PERMUTATION BY PREFIX TRANSPOSITION

1: Build(π)
2: $n \leftarrow$ number of elements
3: **while** π is **not** sorted
4: **do**
5: $k \leftarrow$ last element of first strip
6: $x \leftarrow \pi^{-1}(k) + 1$
7: $y \leftarrow \pi^{-1}(k + 1)$
8: **if** $k < n$ **then**
9: PREFIX-TRANSPOSITION(T, x, y)
10: **else**
11: PREFIX-TRANSPOSITION$(T, x, n + 1)$
12: **end if**
13: **end while**

Lemma V.1. *Algorithm 1 can be implemented in $O(\log n)$ time.*

Proof: In *Algorithm* 1, we see that a prefix transposition takes two $Split$ and two $Join$ operations. From *Theorem III.3*, a $Join$ operation can be implemented in $O(H(t1) - H(t2)))$ time and from *Theorem III.4*, a $Split$ operation can be implemented in $O(\log n)$ time. So time complexity is $2 \times O(H(t1) - H(t2))) + 2 \times O(\log n)$, which is $O(\log n)$. ■

Lemma V.2. *Algorithm 2 can be implemented in $O(n \log n)$ time.*

Proof: The first step of Algorithm 2 is *Build(π)* operation, which takes $O(n)$ time **[1]**. Steps $5 - 7$ needs constant time to run. According to *Lemma V.1* steps $8 - 11$ can be implemented in $O(\log n)$ time. So each iteration of the *while* loop in Line 3 takes $O(\log n)$ time. According to *Lemma IV.1*, each prefix transposition reduces the number of break points from the permutation by at least one. So the total number of prefix transpositions needed is at most n. So, the loop iterates at most n times and since each iteration takes $O(\log n)$ time, the running time of the *while* loop is $O(n \log n)$. So the time complexity of *Algorithm* 2 is $O(n) + O(n \log n)$, which is $O(n \log n)$. ■

VI. Conclusions

We introduced the time complexity of sorting a permutation by prefix transposition using permutation tree. Sorting

a permutation by prefix transpositions, implemented with permutation tree, runs in $O(n \log n)$ time.

ACKNOWLEDGMENT

We express our gratefulness and honor to Dr. M. Sohel Rahman, Associate Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, for his support, advice and care. His patience, scholarly guidance, encouragement, constructive criticism, valuable advice, reading inferior drafts and correcting them at all stages help us properly finish this work.

The Department of Computer Science and Engineering of Bangladesh University of Engineering and Technology provided an ideal environment for conducting our research. Faculty, students and staff alike have created an incredibly friendly and collaborative work environment.

REFERENCES

[1] J. Feng, D. Zhu, *Faster Algorithms for Sorting by Transpositions and Sorting by Block Interchanges*, ACM Transactions on Algorithms, Vol. 3, No. 3, Article 25, Publication date: August 2007.

[2] Zanoni Dias, João Meidanis, *Sorting by Prefix Transpositions*, A. H. F. Laender and A. L. Oliveira (Eds.): SPIRE 2002, LNCS 2476, pp. 65 -76, 2002.

[3] J. S. Firoz, M. Hasan, A. Z. Khan, M. Sohel Rahman, *The 1.375 Approximation Algorithm for Sorting by Transpositions Can Run in $O(n \log n)$ Time*, Journal Of Computational Biology, Volume 18, pp. 1007-1011, Number 8,2011.

[4] Marcelo P. Lopes, Marilia D. V. Braga, Celina M. H. de Figueiredo, Rodrigo de A. Hausen, Luis Antonio B. Kowada, *An analysis and Implementation of Sorting by Transpositions Using Permutation Trees*, O. Norberto de Souza, G.P. Telles, and M.J. Pal akal (Eds.): BSB 2011, LNBI 6832, pp. 42-49, 2011.

[5] V. Bafna, P. A. Pevzner, *Sorting by Transpositions*, SIAM J. Discrete Math.Vol. 11, No. 2, pp. 224-240, May 1998.

[6] D. A. Christie 1999. *Genome rearrangement problem* [Ph.D. dissertation]. University of Glasgow, Glasgow, UK.

[7] Masud Hasan, Mohammed Sohel Rahman, *Advances in Genome Rearrangement Algorithms*, Algorithms in Computational Molecular Biology, Edited by Mourad Elloumi and Albert Y. Zomaya Copyright©2010 John Wiley Sons, Inc.

[8] A. Caprara, *Sorting by reversals is difficult. In Proceedings of the First International Conference on Computational Molecular Biology*, RECOMB'97, pages 75-83, New York, USA, January 1997. ACM Press.

[9] V. Bafna, P. A. Pevzner, *Genome rearrangements and sorting by reversals*, SIAM Journal on Computing, 25(2):272-289, 1996.

[10] P. Berman, S. Hannenhalli, and M. Karpinski. *1.375-approximation algorithm for sorting by reversals*, In Proceedings of the 10th European Symposium on Algorithms (ESA'2002), Lecture Notes in Computer Science, Rome, Italy, September 2002. Springer.

[11] J. Meidanis, M. E. M. T. Walter, and Z. Dias, *Reversal distance of signed circular chromosomes*, Technical Report IC-00-23, Institute of Computing University of Campinas, December 2000.

[12] W. H. Gates and C. H. Papadimitriou. *Bounds for sorting by prefix reversals*. Discrete Mathematics, 27:47-57, 1979.

[13] M. H. Heydari and I. H. Sudborough. *On the diameter of the pancake network*. Journal of Algorithms, 25:67-94, 1997.

[14] Elias, I. and Hartman, T. 2006. *A 1.375-approximation algorithm for sorting by transpositions*. IEEE/ACM Trans. Comput. Biol. Bioinform. 3, 369379.

[15] L. Bulteau, G. Fertin & I. Rusu. *Sorting by transposition is difficult*. SIAM J. on Discrete Math. 26(3), 1148-1180.