

The 1.375 Approximation Algorithm for Sorting by Transpositions Can Run in $O(n \log n)$ Time

Jesun Sahariar Firoz^{1,4}, Masud Hasan^{1,3}, Ashik Zinnat Khan^{1,4}, and M. Sohel Rahman^{1,2,3}

¹ Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1000, Bangladesh

² Department of Computer Science, King's College London, UK

³ {masudhasan, msrahman}@cse.buet.ac.bd

⁴ {jesunsahariar, ashrik}@gmail.com

Abstract. Sorting a Permutation by Transpositions (SPbT) is an important problem in Bioinformatics. In this paper, we improve the running time of the best known approximation algorithm for SPbT. We use the permutation tree data structure of Feng and Zhu and improve the running time of the 1.375 Approximation Algorithm for SPbT of Elias and Hartman (EH algorithm) to $O(n \log n)$. The previous running time of EH algorithm was $O(n^2)$.

1 Introduction

Transposition is an important genome rearrangement operation and Sorting a Permutation by Transpositions (SPbT) is an important problem in Bioinformatics. In the transposition operation, a segment is cut out of the permutation and pasted in a different location. SPbT was first studied by Bafna and Pevzner [1], who discussed the first 1.5-approximation algorithm which had quadratic running time. Eriksson et al. [4] gave an algorithm that sorts any given permutation of n elements by at most $\frac{2}{3}n$ transpositions. Later, Hartman and Shamir used the concept of simplified breakpoint graph to design another 1.5-approximation algorithm with $O(n^2)$ running time [8]. They further used the splay tree to implement this simplified algorithm and thereby reducing the time complexity to $O(n^{\frac{3}{2}}\sqrt{\log n})$ [8]. Finally, Elias and Hartman presented an 1.375-approximation algorithm in [3], which is the best known approximation algorithm for SPbT in the literature so far. The running time of that algorithm [3] however is $O(n^2)$. Very recently, in [5], Feng and Zhu improved the running time of the 1.5-approximation algorithm of [8] to $O(n \log n)$ by introducing and using a new data structure named the *permutation tree*. In this paper, with the help of the permutation tree data structure we improve the running time of the 1.375 Approximation Algorithm for SPbT of [3] to $O(n \log n)$.

2 Preliminaries

A *transposition* $\tau \equiv \text{trans}(i, j, k)$ on $\pi = (\pi_0 \dots \pi_{n-1})$ is an exchange of two disjoint contiguous segments $X = \pi_i, \dots, \pi_{j-1}$ and $Y = \pi_j, \dots, \pi_{k-1}$. Given a permutation π , the SPbT asks to find a sequence of transpositions to transform π into the identity permutation such that the number of transpositions t is minimized. The *transposition distance* of a permutation π , denoted by $d(\pi)$, is the smallest possible value of t . The breakpoint graph $G(\pi)$ [1] is an edge-colored graph on

$2n$ vertices $\{l_0, r_0, l_1, r_1, \dots, l_{n-1}, r_{n-1}\}$. For every $0 \leq i \leq n-1$, r_i and l_{i+1} are connected by a grey edge, and for every π_i , l_{π_i} and r_{π_i-1} are connected by a black edge, denoted by b_i . The breakpoint graph uniquely decomposes into $c(\pi)$ cycles. A k -cycle has k black edges; if k is odd (resp. even), the cycle is *odd* (resp. *even*). Further, if $k < 3$, it is *short* and else, *long*. The number of odd cycles is denoted by $c_{odd}(\pi)$, and we define $\Delta c_{odd}(\pi, \tau) = c_{odd}(\tau.\pi) - c_{odd}(\pi)$, where $\tau.\pi$ denotes the result after τ is applied. A transposition τ is a k -move if $\Delta c_{odd}(\pi, \tau) = k$. A cycle is called *oriented* if there is a 2-move that is applied on three of its black edges; otherwise, it is *unoriented*. If $G(\pi)$ contains only short cycles, then, both π and $G(\pi)$ are called *simple*. A permutation π is *2-permutation* (resp. *3-permutation*) if $G(\pi)$ is contains only 2-cycles (resp. 3-cycles). Permutations can be made simple by inserting new elements into the permutations and thereby splitting the long cycles [6].

Two pairs of black edges (a, b) and (c, d) are said to *intersect* if their edges occur in alternated order in the breakpoint graph, i.e., in order a, c, b, d . Cycles C and D intersect if there is a pair of black edges in C that intersects with a pair of black edges in D . A *configuration* of cycles is a subgraph of the breakpoint graph that is induced by one or more cycles. Configuration A is *connected* if for any two cycles c_1 and c_k of A there are cycles $c_2, \dots, c_{k-1} \in A$ such that, for each $i \in [1, k-1]$, c_i intersects with c_{i+1} . A *component* is a maximal connected configuration in a breakpoint graph. The *size* of a configuration or a component is the number of cycles it contains. A configuration (similarly, a component) is said to be unoriented if all of its cycles are unoriented. A configuration (similarly, a component) is *small* if its size is at most 8; otherwise it is *big*. Small components that do not have an $\frac{11}{8}$ -sequence are called *bad small components* [3].

In a configuration, an *open gate* is a pair of black edges of a 2-cycle or an unoriented 3-cycle that does not intersect with another cycle of that configuration. A configuration not containing open gates is referred to as a full configuration. An (x, y) -sequence of transpositions on a simple permutation (for $x \geq y$) is a sequence of x number of transpositions, such that at least y of them are 2-moves and that leaves a simple permutation at the end.

A *permutation tree* [5] is firstly a balanced binary tree T with root r , where each internal node of T has two children. The left and right children of an internal node t are denoted by $L(t)$ and $R(t)$, respectively. The height of t is denoted by $H(t)$; a leaf node has height zero. Secondly, a permutation tree must correspond to a permutation. The permutation tree corresponding to π has n leaf nodes, labeled by $\pi_1, \pi_2, \dots, \pi_n$ respectively. Each node corresponds to an interval of π and has a *value* equal to the maximum number in the interval. The interval corresponding to an internal node t is be the concatenation of the two intervals corresponding to $L(t)$ and $R(t)$. The height of the permutation tree of π is bounded by $O(\log |\pi|)$. A permutation tree (Build operation) can be built in $O(|\pi|)$ time. Suppose, T, t_1 and t_2 correspond to $(\pi_1 \pi_2 \dots \pi_{m-1} \pi_m \pi_{m+1} \dots \pi_n)$, $(\pi_1 \pi_2 \dots \pi_m)$ and $(\pi_{m+1} \pi_{m+2} \dots \pi_n)$, respectively. Then, $Join(t_1, t_2)$ returns T in $O(H(t_1) - H(t_2))$ time and $Split(T, m)$ returns t_1 and t_2 in $O(\log n)$ time.

3 Faster Running Time for Elias and Hartman's Algorithm

The 1.375-approximation algorithm for SPbT of Elias and Hartman (referred to as the EH algorithm henceforth) is presented in Algorithm 1.

Algorithm 1 EH Algorithm

```
1: Transform permutation  $\pi$  into a simple permutation  $\hat{\pi}$  .
2: Check if there is a (2, 2)-sequence. If so, apply it.
3: While  $G(\hat{\pi})$  contains a 2-cycle, apply a 2-move.
4:  $\hat{\pi}$  is a 3-permutation. Mark all 3-cycles in  $G(\hat{\pi})$ .
5: while  $G(\hat{\pi})$  contains a marked 3-cycle C do
6:   if C is oriented then
7:     apply a 2-move on it.
8:   else
9:     Try to sufficiently extend C eight times
10:    if sufficient configuration has been achieved then
11:      apply an  $\frac{11}{8}$ -sequence.
12:    else
13:      it must be a small component. If an  $\frac{11}{8}$ -sequence is still possible apply it.
14:      if Applying a  $\frac{11}{8}$ -sequence is not possible then
15:        This must be a bad small component. Unmark all cycles of the component.
16:      end if
17:    end if
18:  end while
19: end while
20: Now,  $G(\hat{\pi})$  contains only bad small components. While  $G(\hat{\pi})$  contains
    at least 8 cycles, apply an  $\frac{11}{8}$ -sequence.
21: While  $G(\hat{\pi})$  contains a 3-cycle, apply a (3,2)-sequence.
22: Mimic the sorting of  $\pi$  using the sorting of  $\hat{\pi}$ .
```

Now, to achieve our goal we need to be able to use the permutation tree for applying (x, y) -sequence and k -move. Additionally, given a pair of black edges we can find, with the help of a permutation tree, another pair of black edges such that these two pairs intersect. Feng and Zhu [5] used the following lemma to find such a pair of black edges.

Lemma 1. ([1]) *Let b_i and b_j are two black edges in an unoriented cycle C such that $i < j$. Let $\pi_k = \max_{i < m \leq j} \pi_m$ and $\pi_l = \pi_k + 1$. Then the black edges b_k and b_{l-1} belong to the same cycle and the pair $\langle b_k, b_{l-1} \rangle$ intersects the pair $\langle b_i, b_j \rangle$. \square*

Feng and Zhu suggested that a permutation tree can be used for query and transposition as follows. Assume that the permutation tree T corresponding to a simple permutation $\pi = (\pi_1 \dots \pi_n)$ has been constructed by procedure *Build*. Now, Procedure *Query* (π, i, j) finds a pair of black edges intersecting the pair $\langle b_i, b_j \rangle$ and Procedure *Transposition* (π, i, j, k) , applies a transposition *trans* (i, j, k) on π . These two procedures can be implemented as follows.

1. *Query* (π, i, j) : Split T into three permutation trees, t_1, t_2 and t_3 , corresponding to, respectively, $(\pi_1, \dots, \pi_i), (\pi_i + 1, \dots, \pi_j)$ and $(\pi_j + 1, \dots, \pi_n)$. Clearly this can be done in $O(\log n)$ time by two splitting operations of T . The value of the root of t_2 is the largest element (say, π_k) in the interval $[\pi_i + 1 \dots \pi_j]$. Assume that $\pi_l = \pi_k + 1$. By Lemma 1, pair $\langle b_k, b_{l-1} \rangle$ intersects pair $\langle b_i, b_j \rangle$.
2. *Transposition* (π, i, j, k) : Split T into four permutation trees t_1, t_2, t_3 and t_4 , corresponding to, respectively $(\pi_1, \dots, \pi_{i-1}), (\pi_i, \dots, \pi_{j-1}), (\pi_j, \dots, \pi_k - 1)$ and (π_k, \dots, π_n) . Then, join the four trees by executing *Join* $(Join(Join(t_1, t_3),$

$t_2), t_4)$. Clearly, adjusting the permutation tree T can be done by three splitting and three joining operations spending $O(\log n)$ time.

In the rest of this section we state and prove a number of lemmas concerning the running time of different steps of the the EH algorithm, achieving an $O(n \log n)$ running time for the algorithm in the sequel.

Lemma 2. *Step 1 of the EH algorithm can be implemented in $O(n)$ time.*

Proof. A permutation π is made simple by (g, b) -splits acting on the breakpoint graph $G(\pi)$. A (g, b) -split for $G(\pi)$ splits one cycle into two shorter ones. Equivalently, this operation inserts a new element into π [7]. A breakpoint graph $G(\pi)$ can be transformed into $G(\hat{\pi})$ containing only 1-cycles, 2-cycles, and 3-cycles by a series of (g, b) -splits [8], that is, the permutation corresponding to $G(\hat{\pi})$ becomes simple. This can be done by scanning the permutation linearly and inserting a new element when necessary. Thus Step 1 can be implemented in $O(n)$ time. \square

Lemma 3. *Step 2 of the EH algorithm can be implemented in $O(n \log n)$ time.*

Proof. To check whether a $(2, 2)$ -sequence exists, the following sub-steps are executed:

- (a) We check whether there are (at least) four 2-cycles. If yes, then we are done; otherwise we go to the next step.
- (b) If there are two intersecting 2-cycles then a $(2, 2)$ -sequence exists and we are done [3]. Otherwise we go to the following step.
- (c) If there are two nonintersecting 2-cycles, we apply a transposition on three of the four black edges of the two 2-cycles (check all four possibilities). Clearly, this is a 2-move [2]. Now, there is a $(2, 2)$ -sequence iff in the resulting graph there is an oriented cycle. Otherwise we go to the following step.
- (d) In this case the permutation is a 3-permutation. Here, if all cycles are un-oriented, there is no $(2, 2)$ -sequence. Otherwise, for each oriented 3-cycle, we need to check if, after applying a 2-move on it, there is an oriented cycle in the resulting graph. There is a $(2, 2)$ -sequence iff the answer is yes for some cycle.

Clearly, the complexity depends on Sub-steps c and d as these two cases involve applying the 2-move and the transpositions. Hence the result follows. \square

Lemma 4. *Steps 3 and 4 of the EH algorithm can be implemented in $O(n \log n)$ time.*

Proof. We have even number of 2-cycles in the breakpoint graph for a simple permutation [5]. A 2-move in Step 3 transforms two 2-cycles into a 1-cycle and a 3-cycle. All the 2-cycles of $G(\pi)$ can be found in linear time and be eliminated by at most $\frac{n}{2}$ 2-moves. Since one transposition takes $O(\log n)$ time, Step 3 can be done in $O(n \log n)$ time. Finally, for Step 4, all the 3-cycles can be marked by a linear scan of the breakpoint graph which takes at most $O(n)$ time. Hence the result follows. \square

Lemma 5. *The while loop at Step 5 of the EH algorithm can be implemented in $O(n \log n)$ time.*

Proof. The loop iterates at most n times and each iteration takes $O(\log n)$ time as follows. Step 7 runs in $O(\log n)$ time, because, to apply a 2-move, we use the transposition operation on the permutation tree. Now, consider Steps 9 to 15. There are two types of extensions that are sufficient for extending any cycle C :

1. *Type 1*: Extensions closing open gates, and
2. *Type 2*: Extensions of full configurations such that the extended configuration has at most one open gate.

To do a sufficient extension of Type 1 (add a cycle that closes an open gate), we need to pick an arbitrary open gate and find another cycle that intersects with the open gate. For this, we query the permutation tree with the black edge $\langle b_i, b_j \rangle$ of the open gate under consideration. The query procedure in turn returns the intersecting pair $\langle b_k, b_{l-1} \rangle$ as stated above. This step takes $O(\log n)$ time.

If the configuration is full, i.e., there are no open gates, we do sufficient extension of Type 2. To do this, we query the permutation tree with each pair of black edges of each cycle in the configuration, until we find a cycle that intersects with a pair. If such a cycle is found, we extend the configuration by this cycle to find a component of size greater than or equal to 9. As there can be at most 24 such pairs of black edges, this step takes $O(\log n)$ time as well. Finally, we apply an $\frac{11}{8}$ -sequence by using the transposition procedure of permutation tree which takes $O(\log n)$ time. Hence the result follows. \square

Lemma 6. *Steps 20 to 22 of the EH algorithm can be implemented in $O(n \log n)$ time.*

Proof. In Step 20, the application of an $\frac{11}{8}$ -sequence takes $O(\log n)$ time and this step iterates at most $O(n)$ times. Now, applying a (3,2)-sequence is essentially equivalent to applying 3 transpositions such that at least 2 of them are 2-moves. Since, there can be no more than n 3-cycles, Step 21 also runs in $O(n \log n)$ time. Hence, the result follows, since Step 22 of the EH algorithm can also be implemented in $O(n \log n)$ time [5]. \square

From the above lemmas, it is easy to see that the EH algorithm, implemented with permutation tree, runs in $O(n \log n)$ time.

References

1. V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM J. Discrete Math.*, 11(2):224–240, 1998.
2. D. Christie. Genome rearrangement problem. Ph.D. Thesis, University of Glasgow, 1999.
3. I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):369–379, 2006.
4. H. Eriksson, K. Eriksson, J. Karlander, L. J. Svensson, and J. Wästlund. Sorting a bridge hand. *Discrete Mathematics*, 241(1-3):289–300, 2001.
5. J. Feng and D. Zhu. Faster algorithms for sorting by transpositions and sorting by block interchanges. *ACM Transactions on Algorithms*, 3(3), 2007.
6. Q.-P. Gu, S. Peng, and I. H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theor. Comput. Sci.*, 210(2):327–339, 1999.
7. S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46(1):1–27, 1999.
8. T. Hartman and R. Shamir. A simpler and faster 1.5-approximation algorithm for sorting by transpositions. *Inf. Comput.*, 204(2):275–290, 2006.