Improving the Efficiency of Sorting by Reversals

Yijie Han
School of Computing and Engineering
University of Missouri at Kansas City
5100 Rockhill Road
Kansas City, MO 64110, USA.
hanyij@umkc.edu

Abstract

Sorting signed permutations by reversals is a fundamental problem in computationial molecular biology. In this paper we present an improved algorithm for sorting by reversals. Our algorithm runs in $O(n^{3/2})$ times. This improves the best previous results which runs in $O(n^{3/2}\sqrt{\log n})$ time.

Keywords: Algorithms, computational biology, sorting by reversals, genome rearrangements.

1 Introduction

Sorting by reversal is a problem of genome rearrangement in computational biology. Genes are represented with integers in $\{1, 2, ..., n\}$ with a plus or minus sign to indicate the direction of a gene. A chromosome is a sequence of genes which is represented by a signed permutation π of $\{\pm 1, \pm 2, ..., \pm n\}$. π_i will be used to indicate the i-th element of π. We indicate the sign of an element in a permutation only when it is minus.

The reversal of of the interval $[i, j] \subseteq [1, n]$ $(i < j)$ is the signed permutation $\rho = 1, ..., i, -j, ..., -(i+1), j+1, ..., n$. $\pi\rho$ is the permutation obtained from π by reversing the order and flipping the signs of the elements in the interval. That is $\pi\rho = \pi_1, .., \pi_i, -\pi_j, ..., -\pi_{i+1}, \pi_j, ..., \pi_n$. If $\rho_1, .., \rho_k$ is a sequence of reversals, then it sorts a permutation π if $\pi\rho_1 \cdots \rho_k = Id$ where Id is the all positive identity permutation $1, ..., n$. Let $d(\pi)$ be the number of reversals in

a minimum size sequence sorting π. Sorting by reversals is to compute this minimum size sequence of reversals.

The problem of sorting by reversals has been studied extensively. The first polynomial algorithm was obtained by Hannenhalli and Pevzner[2]. Kaplan, Shamir and Tarjan gave an $O(n^2)$ time algorithm[3]. Bader, Moret and Yan [1] gave a linear time algorithm for computing $d(\pi)$ only. Kaplan and Verbin [4] gave a randomized algorithm with time complexity $O(n^{3/2}\sqrt{\log n})$. Currently the best result is due to Tannier and Sagot[6] and Tannier, Bergeron and Sagot[5]. They used data structure given by Kaplan and Verbin[4] and achieved $O(n^{3/2}\sqrt{\log n})$ time.

In this paper we show how to improve on the data structure given by Kaplan and Verbin[4] and achieve $O(n^{3/2})$ time.

2 Preliminaries

We will always augument permutations by adding $\pi_0 = 0$ and $\pi_{n+1} = n + 1$. To sort the permutation we must locate a reversal ρ such that $d(\pi\rho) = d(\pi) - 1$. Such a reversal is called a safe reversal.

Given an augmented signed permutation, $\pi = (0, \pi_1, \pi_2, ..., \pi_n, n + 1)$, a pair (π_i, π_j) with $i < j$ and π_i, π_j being consecutive integers (i.e. $|\pi_i| - |\pi_j| = \pm 1$) is called an oriented pair if its elements are of opposite signs, and unoriented pair otherwise. For each pair, the element $\pm x$ is called the local end of the pair and the element $\pm(x+1)$ is the remote end of the pair. There are exactly $n + 1$ pairs in π and there are no oriented pairs iff all elements are positive.

The reversal $\rho = \rho(i, j)$, which changes permutation $\pi = (\pi_0, \pi_1, ..., \pi_{n+1})$ to $\pi\rho = (\pi_0, ..., \pi_{i-1}, -\pi_j, -\pi j - 1, ..., -\pi_i, \pi_{j+1}, ..., \pi_{n+1})$, is oriented if either $\pi_i + \pi_{j+1} = 1$ or $\pi_{i-1} + \pi_j = -1$. Oriented reversals can be derived from oriented pairs: if (π_i, π_j) is an oriented

pair, then the reversal

$$\begin{cases} \rho(i, j-1) & if\ \pi_i + \pi_j = 1, \\ \rho(i+1, j) & if\ \pi_i + \pi_j = -1, \end{cases}$$

is an oriented reversal.

Currently the best algorithms[4, 6] require a data structure to support the following operations:

1. Query for π_i.
2. Query for π_i^{-1}. That is given an element i to find its position in the permutation.
3. Draw an oriented pair.
4. Perform a reversal

The data structure given by Kaplan and Verbin [4] does 1 in logarithmic time, 2 in constant time, 3 in logarithmic time and 4 in $O(\sqrt{n \log n})$ time. Since $d(\pi)$ can be $O(n)$ therefore the time complexity obtained in [4, 5, 6] for sorting by reversals is $O(n^{3/2}\sqrt{\log n})$.

3 Improvements

We present modified data structure of Kaplan and Verbin[4]. We divide a permutation into $O(\sqrt{n})$ blocks and each block contains a contigious section of size $O(\sqrt{n})$ of the permutation. We use a B-tree to represent the permutation. Each block is represented by an array and stored at a leaf of the B-tree. The array contains the elements of the permutation and a doubly linked pointer for each pair of the permutation. At each node of the B-tree we use variables to record how many leaves and how many elements of the permutation are there in each branch of the B-tree. Thus for a reversal $\rho(i, j)$, by following the the array and the branches in the B-tree we can find π_i and π_j in logarithmic time.

For computing π_i^{-1} we use the same date structure as that in [4], namely we use an

array of pointers with the i-th element pointing to π_i^{-1}. By following the links in the B-tree we can find out π_i^{-1} in logarithmic time.

Now we explain the data structure for drawing an oriented pair and for performing a reversal.

At each leaf of the B-tree we use a linked list with $t = O(\sqrt{n})$ elements, where t is the number of blocks existing currently. The i-th element of the list is a structure recording how many pairs originates from the current block and has the remote end in block i, how many of these pairs are oriented and how many are unoriented.

The B-tree has $L = O(\log n)$ levels. We number levels from the root with root at level 0 and each node is at level l if it has distance l to the root. Nodes of the B-tree at level l are called blocks at level l. There are no more than $t/2^{L-l}$ blocks at level l. At each block at level l we use a linked list with $t_l = O(t/2^{L-l})$ elements, where t_l is the number of blocks at level l. The i-th element of the linked list is a structure recording how many pairs have local ends in the current block (i.e. originates from a leaf of the subtree rooted at the current block at level l) and has the remote end in block i at level l (i.e. end at a leaf of the subtree rooted at block i at level l). A doubly linked list is maintained for this pair at level l. How many of these pairs are oriented and how many are unoreinted are also recorded.

We also maintain block pointers. For each block, there is a pointer pointing to the element in the linked lists corresponding to this block.

Fig. 1. shows such a B-tree.

First we show how to split a block at a leaf and how to join two blocks at the leaves. As in [4], we maintain the invariant:

Invariant: After each operation the number of elements in each block at level L (i.e. at the leaf level of the B-tree) is between $\frac{1}{2}\sqrt{n}$ and $2\sqrt{n}$.

We do split or join when this invariant is voilated. Of course we also do split for

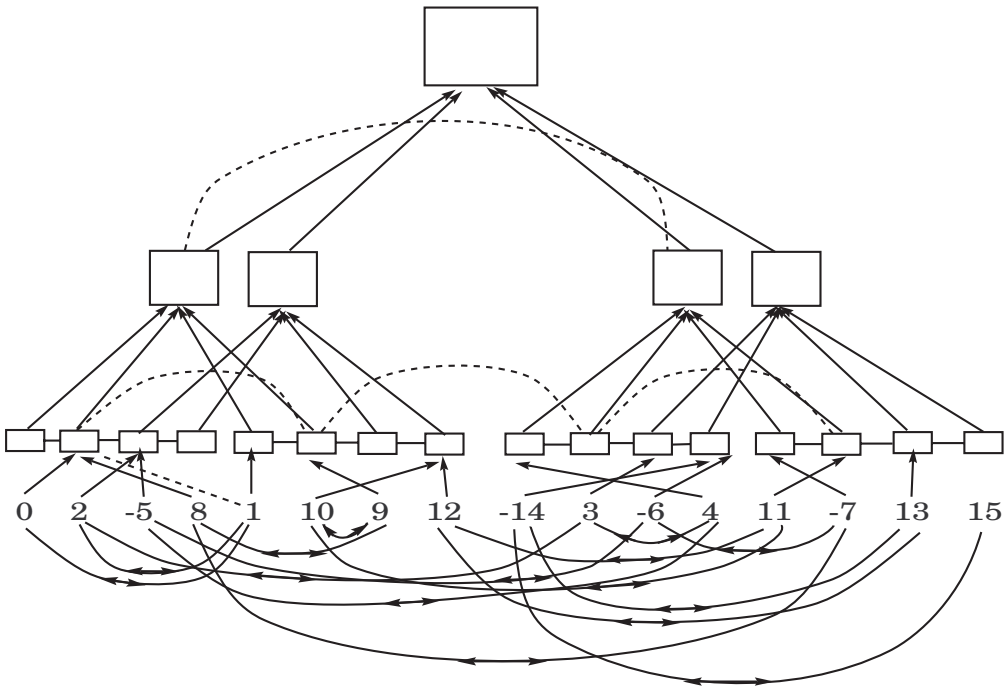

Fig. 1. At the leaf level four elements are in a block. There is a linked list for each block. Dashed lines are block pointers. For clarity only block pointers for block 2 is shown at the leaf level. Only block pointers for block 1 is shown in level 1. u: unoriented. o: oriented.

performing reversals.

To split a block at a leaf of the B-tree, for each element $\pm(x+1)$ in the block we use the doubly linked pointers to find its local end $\pm x$. If $\pm x$ is in block i we modify the linked list at block i. We use pointer from $\pm(x+1)$ to $\pm x$ and pointer from $\pm x$ to the linked list at block i to access the linked list at block i. A new node is added to block i and also pointers from block i to the split current blocks are also added to reflect that the current block is split into two. This change is carried from block i to its ancesters. We also add an element to the linked list of each block by using block pointers. Fig. 2. shows such a change. At the leaf level this involves $O(\sqrt{n})$ operations because each block is of size $\sqrt{n}$. At lower numbered levels the operation is geometrically decreasing because each block at such level has geometrically decreasing size.

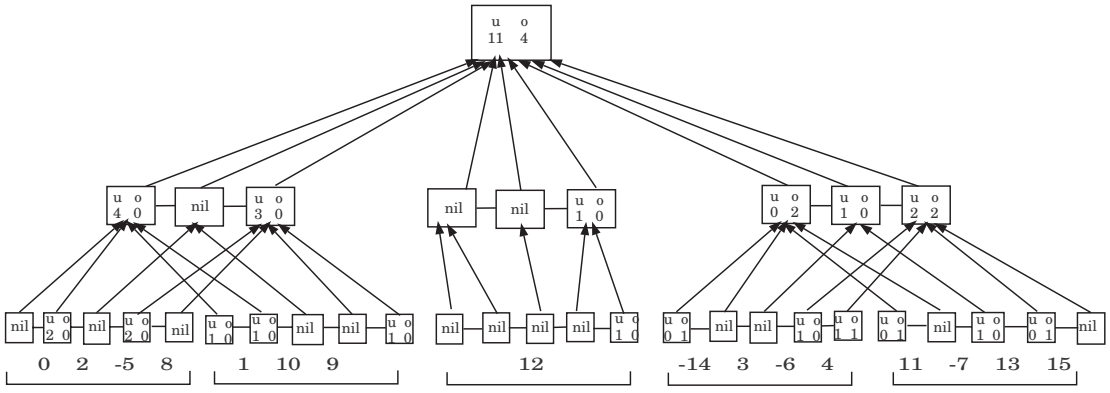

Fig. 2. 12 is split from the second block to form its own block.

To join two blocks at the leaves of the B-tree, we combined the two linked lists at the two blocks and by using doubly linked pointers for pairs we can access block i if block i has the local end of a pair whose remote end is in the current blocks(blocks to be combined). Therefore we can adjust the linked list at block i to reflect the changes. We also delete the elements corresponding to the deleted block in the linked list in each block by using block pointers. Fig. 3. shows such a change. Again we spend $O(\sqrt{n})$ time at the leaf level and

spend geometrically decreasing time at lower numbered levels. Therefore the total time will be $O(\sqrt{n})$.

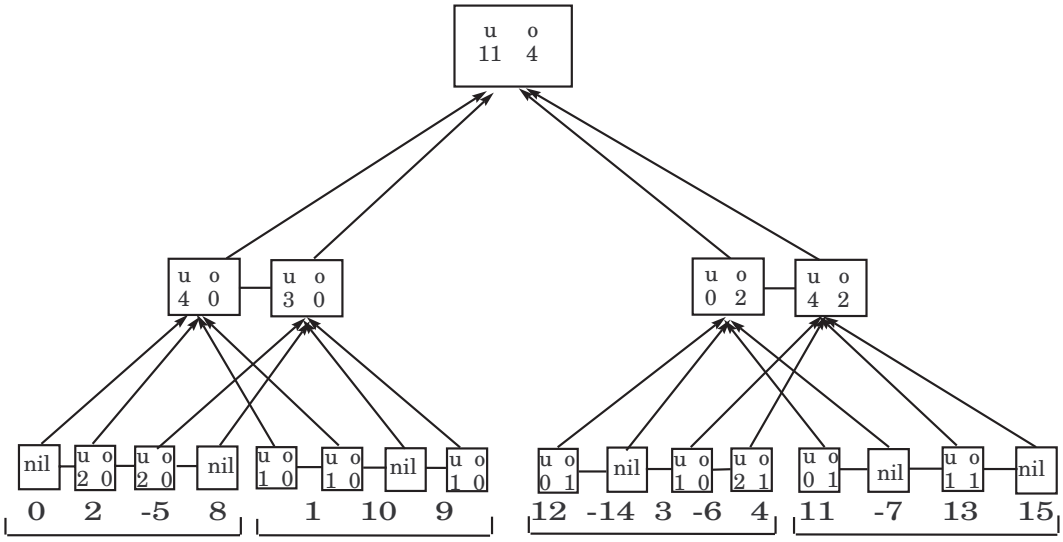

Fig. 3. 12 is joined with the third block.

Spliting and joind blocks at level lower than the leaf level can be done similarly except they do not affect higher numbered levels in the B-tree.

Now we show how to draw oriented pairs and how to do reversal. Oriented pairs can be drawn easily as at each node of the B-tree there are counters for oriented pairs and unoriented pairs acrossing blocks at the level. Thus by following the B-tree an oriented pair can be drawn in logarithmic time. To do a reversal we first split blocks and therefore reversals are done on the whole block boundary instead of within a fraction of a block. Next we split the B-tree into three trees with the middle tree represents elements in the reversal. For pairs with local end in the first or the last tree and remote end in the middle tree we swap the counter for oriented pairs and unoriented pairs at the roots of the B-trees. For pairs with local end in the middle tree and remote end in the first or last tree we also swap the counter for oriented pairs and unoritend pairs at the roots of the B-trees. We use

flags to indicate that counters for oriented pairs and unoriented pairs have been swapped. Thus when we push down the flags we can swap the counters of oriented and unoriented pairs in the subtrees. We also set the reverse flag to indicate the the middle tree should be reversed. This accomplishes a reversal. The time needed for a reversal is dominated by spliting blocks and therefore is $O(\sqrt{n})$.

Since each reversal can be done in $O(\sqrt{n})$ time and there are $O(n)$ reversals the time for sorting by reversals is $O(n^{3/2})$.

Theorem: Sorting signed permutation by reversals can be computed in $O(n^{3/2})$ time.

References

[1] D.A. Bader, B.M.E. Moret, M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Proc. of the 7th Workshop on ALgorithms and Data Structures*, 365-376(2001).

[2] S. Hannenhalli, P. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals. *Proc. of the 27th ACM Symposium on Theory of Computing*, 178-189(1995).

[3] H. Kaplan, R. Shamir, R.E. Tarjan. Faster and simpler algorithm for sorting signed permutation by reversals. *SIAM J. on Computing* 29, 880-892(1990).

[4] H. Kaplan, E. Verbin. Sorting signed permutatioins by reversals, revisited. *J. of Computer and System Sciences* 70, 321-341(2005).

[5] E. Tannier, A. Bergeron, M.-F. Sagot. Advances on sorting by reversals. To appear in *Discrete Applied Mathematices*.

[6] E. Tannier, M.-F. Sagot. Sorting by reversals in subquadratic time. *Proceedings of 2004 Combinatorial Pattern Matching*, 1-13(2004).