Efficient Data Structures and a New Randomized Approach for Sorting Signed Permutations by Reversals

Haim Kaplan and Elad Verbin*

School of Computer Science, Tel Aviv University, Tel Aviv, Israel.

Abstract. The problem of sorting signed permutations by reversals (SBR) is a fundamental problem in computational molecular biology. The goal is, given two genomes (represented as permutations of the gene collection), to find the most parsimonious scenario of reversals that transforms one genome into the other.
In this paper we describe a randomized algorithm for sorting signed permutations by reversals. The algorithm tries to sort the permutation by repeatedly drawing a random oriented reversal. It relies on the observation that, typically, a very large percentage of oriented reversals are indeed part of a most parsimonious scenario.
To obtain an efficient implementation of our algorithm we describe some efficient data structures that allow us to maintain a permutation under reversals and draw random oriented reversals from it, all in sub-linear time per operation. We demonstrate empirically that our algorithm offers a substantial improvement in running time over current algorithms (namely, that it runs in $O(n^{3/2}\sqrt{\log n})$ expected running time on a random permutation). The data structures we present may also be of independent interest for developing other algorithms for SBR.

1 Introduction

A common method that biologists use when they want to estimate the evolutionary distance between two species is to compare their DNA sequences and estimate how many global mutations (i.e. mutations that span a relatively large portion of the DNA) occurred on the evolutionary path between them. In this context, a DNA sequence can be thought of as an ordered sequence of genes. Since the most common global mutation that happens in nature is a reversal of an entire segment of genes (in which the segment is plucked out of the sequence and reinserted in reversed order), a good estimate is that the evolutionary distance is proportional to the length of a shortest scenario of reversals required to transform one of the DNA sequences into the other. For more details on this, and on algorithmic issues in computational molecular biology in general, see the book by Pevzner [13].

Formally, we identify the genes by the numbers $1, \ldots, n$ and represent their order by a permutation π of $1, \ldots, n$. As genes have directionality, signs correspond to their direction. A reversal of a segment of the permutation reverses the order and flips the signs of the elements in this segment. So, given two signed permutations π_1 and π_2 the problem of *sorting signed permutations by reversals* (SBR) asks for a minimum-length sequence of reversals that transforms π_1 to π_2. It is easy to see that this is in fact equivalent to asking for a sequence of reversals of minimum length that transforms a given signed permutation π to the positive identity permutation $(+1, +2, \ldots, +n)$.

Mathematical analysis of sorting by reversals was initiated by Sankoff [14, 15]. Kececioglu and Sankoff [11] gave the first constant-factor polynomial approximation algorithm for the problem, and conjectured that the unsigned variant of sorting by reversals (in which we are dealing with an unsigned permutation) is NP-hard, a conjecture that was later verified by Caprara [5]. However, in 1995, Hannenhalli and Pevzner [9] showed that the problem of sorting a *signed* permutation by reversals is in fact polynomial. They proved a duality theorem that equates the reversal distance with the sum of three combinatorial parameters associated with the permutation. Based on this theorem, they described an algorithm that sorts signed permutations by reversals in $O(n^4)$ time. More recently, Kaplan, Shamir, and Tarjan [10] simplified the underlying combinatorial structure and described an algorithm that finds a shortest sorting sequence in $O(n^2)$ time. Last year Bergeron [3] simplified the underlying combinatorial structure even further and described a somewhat

* email: {haimk, eladv}@cs.tau.ac.il

different algorithm that runs in $O(n^3)$ time. Unfortunately Bergeron has not been able to use her simplified analysis of the underlying structure to beat the $O(n^2)$ algorithm of Kaplan et. al., which is still the asymptotically fastest algorithm to date. We further note that if one is interested only in the length of the shortest sequence, and does not require the sequence itself, then it could be calculated in linear time [1].

1.1 Our Results

In this paper we present an algorithm and empirically show that it runs in $O(n^{3/2}\sqrt{\log n})$ expected running time on a random permutation. To that end we develop new data structures for manipulating permutations, and present a new approach for sorting a permutation that uses randomization. To better motivate our randomized scheme for finding a sorting sequence we need some background about the underlying combinatorial structure of the problem.

The underlying combinatorial theory (discovered by [2] and refined in [9] and [10]) distinguishes a class of reversals called oriented reversals. An *oriented reversal* is a reversal that makes consecutive elements in the permutation adjacent with the same sign[1]. It is easy to see that a permutation has an oriented reversal iff it has a negative element . Some of the oriented reversals are further distinguished as *safe*. A *safe* oriented reversal transforms π into π' where $d(\pi') = d(\pi) - 1$. The theory guarantees that if a permutation has an oriented reversal then it also has a safe oriented reversal. All existing algorithms for solving SBR invest at least linear time in searching for a safe oriented reversal, "performing it" (i.e. updating the data structures so that they represent π' rather than π), and repeating the process. Since the distance of a permutation is at most $n + 2$ (and is almost always $\Theta(n)$) this means that in order to get a sub-quadratic algorithm we must take a different approach.

There is one extra complication, though: For many permutations the process of repeatedly picking a safe oriented reversal and applying it will indeed generate a shortest sorting sequence. However, this is not true for all permutations. There are permutations containing structures called *unoriented components* (defined in the next section) which this process will *always* fail to sort. For permutations that contain unoriented components *all* sequences of safe oriented reversals will end with a permutation that has no oriented reversals, but is not the identity permutation. The theory suggests algorithmic ways of handling these special permutations: One can detect in linear time whether a permutation contains unoriented components or not. Furthermore, in case π contains unoriented components one can find in linear time a "good" sequence of reversals that "clears these components", i.e. a sequence of t reversals that transforms π into π' that has no unoriented components and $d(\pi') = d(\pi) - t$. It is also important to note that an oriented reversal is safe iff by applying it we do not create new unoriented components. So if we start with a permutation with no unoriented components and perform an unsafe oriented reversal we end up with a permutation that contains an unoriented component. Furthermore, subsequent oriented reversals cannot clear this unoriented component, and so once we perform an unsafe oriented reversal, the process of repeatedly picking a safe oriented reversal and performing it will not succeed in fully sorting the permutation.

Our randomized algorithm described in Sec. 2 is motivated by the above observations, as well as by empirical studies on random permutations showing that, typically, a very large portion of oriented reversals are safe, and so we may be able to wave the complex task of finding a safe oriented reversal, instead just picking a random oriented reversal and hoping it is safe. This argument is supported by the fact that an unsafe oriented reversal creates an unoriented component, but easy calculations show that the fraction of the permutations of length n that contain an unoriented component is very small, that is $O(\frac{1}{n^2})$. More results that support our claims were presented in a recent paper by Bergeron et. al. [4].

Now we are ready to describe our algorithm. We first clear the unoriented components (if there are any) using the method of [10]. To sort the "cleared" permutation we iterate the following "random walk"-like process. This random walk repeatedly picks a random oriented reversal and applies it (without bothering to check if it is safe or not). It repeats this process until we get a permutation with no oriented reversals (i.e. a positive permutation). The theory above implies that this walk generates a shortest sorting sequence iff it ends with the identity permutation. If the process fails (i.e. we ended with a permutation that is not the

[1] By adjacent we mean either $i, i+1$ or $-(i+1), -i$, in an extended permutation where we add 0 at the beginning and $n+1$ at the end, both considered to be positive

identity permutation) we repeat it and when it fails about $\log n$ times we resort to running one of the known algorithms to sort the permutation. In Sec. 2 we give a more comprehensive discussion on the algorithm.

This algorithm raises a few intriguing questions. First, what is the probability that a walk in fact succeeds in sorting the permutation ? Second, how much time does it take to sort a permutation using this method ?

Regarding the success probability we obtain good empirical results. The success probability on a random permutation is about 0.63 regardless of the length of the permutation, and the expected number of times we need to run the walk until we get a sorting sequence is roughly 1.6 (The expectation is over the space of all permutations). We consider the empirical results to be quite strong, as the statistics seem to be independent of the size of the permutation, showing a trend that hints at the existence of some hidden structure.

There is one important missing link in our analysis. That is how fast can we perform one step of the random walk ? Note that an obvious implementation which enumerates the oriented reversals at each step takes linear time per step, and that keeping a list of those oriented reversals and just updating it does not improve the situation because sometimes the number of reversals that become oriented is linear in n. In Sec. 3 we show that by using a clever data structure for representing the permutation we can draw a random oriented reversal and apply it in $O(\sqrt{n \log n})$ time. Combining this with the empirical study we conjecture that the expected running time of our algorithm on a random permutation is $O(n \cdot \sqrt{n \log n})$. The data structure we present may be interesting in its own right, because if we could expand it to maintain some more data about the permutation then we would get a provably worst-case sub-quadratic algorithm for SBR.

1.2 Some Definitions

We begin with some definitions that will be used throughout the paper. Note that we will always be working with augmented permutations, that is, we define $\pi_0 = +0$, $\pi_{n+1} = n+1$. This is common practice in SBR papers, and allows us to avoid some special cases.

Given an augmented signed permutation, $\pi = (+0, \pi_1, \pi_2, \ldots, \pi_n, n+1)$, we consider *pairs* (π_i, π_j) so that $i < j$ and π_i, π_j are consecutive integers (that is, $|\pi_i| - |\pi_j| = \pm 1$). Such pair is called an *oriented pair* if its elements are of opposite signs, and *unoriented* otherwise. Conventionally, 0 is considered to be positive, and not negative. Note that there are exactly $n+1$ pairs in π, and that there are no oriented pairs iff the permutation is positive (i.e. all of its elements are positive).

The reversal $\rho = \rho(i,j)$, which changes the permutation $\pi = (\pi_0, \pi_1, \ldots, \pi_{n+1})$ into $\rho \cdot \pi = (\pi_0, \ldots, \pi_{i-1}, -\pi_j, -\pi_{j-1}, \ldots, -\pi_i, \pi_{j+1}, \ldots, \pi_{n+1})$ is called *oriented* if either $\pi_i + \pi_{j+1} = +1$ or $\pi_{i-1} + \pi_j = -1$ (that is, if it makes a consecutive pair of elements adjacent and identically signed). Alternatively, oriented reversals can be derived from oriented pairs: if (π_i, π_j) is an oriented pair, then the reversal

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

is an oriented reversal; We can get all oriented reversals this way. Note that we get a specific oriented reversal from either one or two different oriented pairs.

To understand the role of oriented reversals even further and to define precisely the connected components of a permutation we also recall the following combinatorial structures associated with a permutation [10]. These definitions are for the sake of completeness and are not used in the rest of the paper.

Consider first a *positive* augmented permutation π. A consecutive pair (π_i, π_{i+1}) is a *breakpoint* of π if and only if $|\pi_i - \pi_{i+1}| > 1$; otherwise, it is an *adjacency* of π. We denote by $b(\pi)$ the number of breakpoints in π. The *breakpoint graph* $B(\pi)$ is an edge-colored graph on $n+2$ vertices $\{\pi_0, \pi_1, \ldots, \pi_{n+1}\} = \{0, 1, \ldots, n+1\}$. We join vertices π_i and π_j by a *black edge* if (π_i, π_j) is a breakpoint in π and by a *gray edge* if (i, j) is a breakpoint in π^{-1}.

Getting back to signed permutations, we define a one-to-one mapping u from the set of signed permutations of order n into the set of unsigned permutations of order $2n$ as follows. Let π be a signed permutation. To obtain $u(\pi)$, replace each positive element x in π by $2x-1, 2x$ and each negative element $-x$ by $2x, 2x-1$. For any signed permutation π, let $B(\pi) = B(u(\pi))$. Note that in $B(\pi)$ every vertex is either isolated or incident to exactly one black edge and one gray edge. Therefore, there is a unique decomposition of $B(\pi)$ into cycles. The edges of each cycle alternate between gray and black. Call a reversal $\rho(i,j)$ such that i is odd and

j even an *even reversal*. The reversal $\rho(2i+1, 2j)$ on $u(\pi)$ mimics the reversal $\rho(i+1, j)$ on π. Thus, sorting π by reversals is equivalent to sorting the unsigned permutation $u(\pi)$ by even reversals. We let $b(\pi) = b(u(\pi))$ and let $c(\pi)$ be the number of cycles in $B(\pi)$. We say that a reversal ρ *acts on* a gray edge e if it acts on the breakpoints which correspond to the black edges incident with e.

Two intervals on the real line *overlap* if their intersection is nonempty but neither properly contains the other. A graph G is an *interval overlap graph* if one can assign an interval to each vertex such that two vertices are adjacent if and only if the corresponding intervals overlap (see, e.g., [8]). For a permutation π, we associate with a gray edge (π_i, π_j) the interval $[i, j]$. The *overlap graph* of a permutation π, denoted $OV(\pi)$, is the interval overlap graph of the gray edges of $B(\pi)$. Namely, the vertex set of $OV(\pi)$ is the set of gray edges in $B(\pi)$, and two vertices are connected if the intervals associated with their gray edges overlap. A connected component of $OV(\pi)$ that contains an oriented edge is called an *oriented component*; otherwise, it is called an *unoriented component*.

An oriented reversal ρ is called *safe* if there are no new unoriented components in $OV(\pi' = \rho \cdot \pi)$. It is known [9] that for π without unoriented components any sequence of safe oriented reversals is a minimum sorting sequence.

2 Exploiting randomization to sort by reversals

Our randomized algorithm for SBR works as follows. First if π has unoriented components we clear them using e.g. the method of [10]. Then we repeat the following *RandomWalk* procedure either until it provides a sorting sequence or until it fails $\log n$ times. In the latter case we resort to the algorithm of Kaplan et. al. [10] or the algorithm of Bergeron [3] and let it generate a sorting sequence. Here is the definition of the *RandomWalk* procedure.

INPUT: A permutation π with no unoriented components.
OUTPUT: A solution for SBR on π, or a "failed" indicator.

While (π is not positive)
  Select uniformly at random an oriented reversal of π and perform it.
If ($\pi = id$) return the sequence of reversals we performed.
  else, return "failed".

Alternatively we can run a variant of this RandomWalk procedure which instead of selecting a random oriented reversal selects a random oriented pair and performs the oriented reversal which corresponds to it. Computationally the two procedures are equivalent – if we can select a random oriented pair then we can also select a random oriented reversal in roughly the same time as follows: We pick a random oriented pair p that defines an oriented reversal ρ and check whether there is another oriented pair which defines ρ. If only one pair defines ρ, we return ρ. If two pairs define ρ, we flip a coin; If the coin comes out heads, we return ρ, and if tails, we pick at random a new oriented pair and repeat this procedure. Clearly the average number of oriented pairs we need to draw to get a random oriented reversal is no more than two. Likewise, given a method for choosing a random oriented reversal we can select a random oriented pair in approximately the same time. Considering this, we will freely define only one of these or the other.

It is easy to see that after at most $n+1$ steps the *RandomWalk* procedure reaches a positive permutation. However, a naive implementation of *RandomWalk* would spend linear time per reversal (This implementation traverses the permutation, finds all oriented reversals, and picks one at random.), for a total running time of $\Theta(n^2)$. Another straightforward implementation picks a random oriented pair by drawing a random index $i \in [0, n]$ and checking if the pair whose elements are $\pm i$ and $\pm(i+1)$ is oriented. If it is, we select it; Otherwise, we draw another index and try again. The latter method may be more efficient for some permutations, but there are permutations for which it still takes $\Theta(n)$ expected time to draw an oriented reversal, in $\Theta(n)$ iterations, leading to $\Theta(n^2)$ expected running time on a worst case permutation [2].

Fortunately, there is a data structure that maintains a signed permutation under reversals and allows us to draw a random oriented reversal and perform it in sub-linear time. In section 3.2 we describe this data

[2] One such example is a permutation of Ozery and Shamir that will be discussed shortly [12].

structure, which allows us to draw a random oriented reversal and perform it in time $O(n^{1/2}\log n)$. Using this representation *RandomWalk* can be implemented to run in $O(n \cdot n^{1/2}\log n)$ time in the worst case.

Once we see that a single run of *RandomWalk* can be implemented in sub-quadratic time, the next question to ask is whether our suggested algorithm that repeats calling *RandomWalk* up to $\log n$ times runs in sub-quadratic expected time on the worst-case permutation. Unfortunately this is not the case. There are permutations for which with very high probability *RandomWalk* fails $\log n$ times. One such example is the permutation of size $2k-1$,
$\pi = (2, 4, 6, \ldots, 2k-2, -1, -3, -5, \ldots, -(2k-3), 2k-1)$, which was brought to our attention by Ozery and Shamir [12]. No matter which reversals we pick to sort it, throughout the sorting sequence we alternate between a permutation in which all oriented reversals are safe (and symmetric) and a permutation with exactly two oriented reversals, only one of which is safe. Therefore, the chances of succeeding in one iteration of *RandomWalk* are extremely low, namely $\frac{1}{2^{k-1}}$.[3] The probability that we succeed in one out of $\log n = \log(2k-1)$ iterations is $O(\frac{\log k}{2^{k-1}})$ which still goes rapidly to zero with k. So for this permutation our algorithm almost surely resorts to run one of the standard algorithms for SBR, both of which take $\Omega(n^2)$ on most permutations (including this one). Therefore, the worst-case complexity of Algorithm *RandomWalkIter* is the same as the complexity of the standard algorithm we choose to run.[4]

However, it still could be that the permutations for which *RandomWalk* fails $\log n$ times are rare and on a random permutation the running time is indeed $o(n^2)$. We address this question empirically in the next section.

2.1 Empirical Results and Conjectures about RandomWalk

As previously stated, we expect that iterating RandomWalk would work well on most permutations. This is because typically a large fraction of the oriented reversals are safe. To support this intuition we measured empirically the performance of *RandomWalk* on random permutations. For a random permutation without unoriented components of size ranging from 10 to 10000 we estimated the probability that a single iteration of *RandomWalk* succeeds. We also estimated the average number of iterations of *RandomWalk* that our algorithm runs until it gets a sorting sequence. Our results are summarized in Table 1 and in Figure 1.

n	number of permutations tested	percentage of success at first iteration	average number of iterations until success
10	300000000	66.20%	1.642
20	100000000	63.86%	1.607
50	30000000	63.01%	1.594
100	10000000	62.86%	1.593
200	3000000	62.76%	1.594
500	600000	62.69%	1.596
1000	200000	62.62%	1.596
2000	50000	63.35%	1.586
5000	6000	63.7%	1.58
10000	2000	63%	1.59

Table 1. The performance of *RandomWalk* and *RandomWalkIter* on a random permutation.

The conclusions from our experiments are as follows:

[3] Surprisingly, this is not much better than an easily proven lower bound for the probability of success of *RandomWalk* on any permutation with no unoriented components: That probability is guaranteed to be at least $\frac{1}{(n+1)!}$, since the number of oriented pairs starts from at most $n+1$ and decreases by at least 1 every time a safe reversal is performed, and every permutation without unoriented components has at least one safe reversal.

[4] Indeed, if we are indifferent of the worst-case behavior of our algorithm we can, simply run *RandomWalk* until it is successful instead of resorting to another algorithm when faced with difficulty. If we do that the average-case complexity stays the same and the algorithm is easier to code.

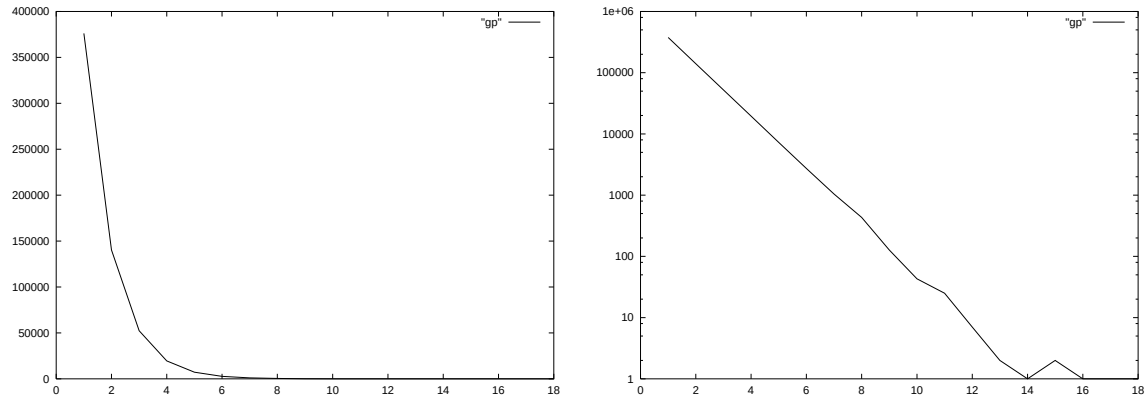

Fig. 1. Distribution of the number of *RandomWalk* iterations needed to sort a permutation of size $n = 500$. The x-axis is the number of iterations. The y-axis in the left graph is the number of permutations that are sorted in that iteration, and the y-axis in the right graph is the logarithm of the number of permutations that are sorted in that iteration.

- If we select a random permutation that has no unoriented component uniformly of all such permutations of size n and run Procedure *RandomWalk* on it once it has about 0.63 chance of success, regardless of n.
- If we select a permutation as above and run *RandomWalk* repeatedly until we get a sorting sequence we will need roughly 1.6 trials on average, also regardless of n.
- We also found that running the variant of *RandomWalk* that chooses a random oriented pair instead of a random oriented reversal we obtained roughly the same results. We also obtained similar results when we ran on a random permutation having $n + 1$ breakpoints (which are, in some sense, the only permutations of "true" size n)

We leave open the question of finding analytical support to our experimental findings. We conjecture that the following theorem holds.

Conjecture 1. The average number of runs of *RandomWalk* needed to find a sorting sequence of a uniformly-chosen signed permutation with no unoriented components of size n is $O(1)$.

It would immediately follow from this conjecture that the average running time of our algorithm on a random permutation is $O(n \cdot \sqrt{n \log n})$. Recall that at most $O(1/n^2)$ fraction of all permutations contain unoriented components. Therefore the average complexity of our algorithm on a randomly chosen permutation and on a randomly chosen permutation with no unoriented components differ by at most a constant factor if our fallback procedure when *RandomWalk* fails $\log n$ times runs in $O(n^3)$ time.

3 Data Structures for Handling Permutations

Existing algorithms for SBR maintain the current permutation via two integers arrays, one containing π and the other containing π^{-1}. With this representation we can locate elements of the permutation in $O(1)$ time. But it would take time proportional to $j - i$ to perform a reversal $\rho(i, j)$, i.e. to update the arrays so that they represent $\pi' = \rho \cdot \pi$. Thus on the worst case (i.e. for long reversals), it takes $\Theta(n)$ time to update the arrays. It follows that any algorithm for SBR that looks for one reversal at a time and runs in sub-quadratic time must manipulate the permutation via a better representation that allows to "perform" a reversal in $o(n)$ time.

Another way to hold a permutation would be to represent π and π^{-1} by threading the elements in two doubly linked lists. One representing π and the other representing π^{-1}. With this representation we can perform a reversal in $O(1)$ time. It is harder however to locate various elements of the permutation. In particular, it is not clear how using this representation one would draw a random oriented reversal in sub-linear time.

It follows that we cannot implement *RandomWalk* to run in sub-linear time with neither of these two obvious representations. In this section we present alternative representations that do allow faster implementation of *RandomWalk*. Any such representation should support two basic operations: 1) Draw a random oriented reversal, 2) perform a reversal ρ on π, and they should take sub-linear time each.

To get started we show two simple representations that allow three operations: 1) query: locating π_i given i, 2) inverse query: locating π_i^{-1} given i, 3) performing a reversal, all in sub-linear time. This allows us to demonstrate the basic techniques in a simpler setup. Later we show how to use these techniques to construct more complicated representations that also allow to draw a random oriented reversal in sub-linear time.

3.1 Maintaining the Permutation

The two simple representations we describe here were used by Chrobak et. al. [6] and further investigated by Fredman et. al. [7]. In both cases they were used to efficiently implement a common local improvement heuristics for the traveling salesman problem. These data structures are quite practical and perform well on practice.

The first representation is based on balanced binary search trees. Using this representation we can perform queries, inverse queries and reversals all in $O(\log n)$ time. Our second representation is based on a partition of the permutation into blocks of size $\Theta(\sqrt{n})$. Using this representation we can perform queries in $O(\log n)$ time, inverse queries in $O(1)$ time and reversals in $O(\sqrt{n})$ time. Our later data structures that allow to draw a random oriented reversal borrow ideas from both these two simple structures.

The first data structure is built upon any balanced binary search tree data structure that is re-balanced via rotations and supports split and concatenate operations, such as splay trees [16], red-black trees, 2-3 trees and AVL trees. Fredman et. al. used splay trees which offer good constants and for which the implementations of split and concatenate are particularly elegant, and so our description, too, will use splay trees (and our running times will therefore be amortized).

In our representation we hold a tree with n nodes containing the elements of the permutation, such that an in-order read of the tree gives us the permutation. We complicate matters by introducing a *"reversed" flag* for each node, which, if turned on, indicates that the subtree rooted at that node should be read in reversed, that is – from right to left, and the signs of its elements should be flipped. Further "reversed" flags down the tree can once again alter the order of the implied permutation. The invariant we keep is that an in-order traversal of the tree, modified by the "reversed" flags, always gives us the permutation.

We note that we can clear the "reversed" flag in an internal node by exchanging its children, flipping the state of the "reversed" flag in each of them, and flipping the sign of the element at the root. The "reversed" flag of a leaf can be cleared if we flip the sign of the element contained in that leaf. One can view this process as pushing down the "reversed" flags. Using this ability to clear a "reversed" flag we can modify the rotation operation in a splay tree to correctly maintain the permutation when performing a rotation. This is done by pushing down the "reversed" flag of the nodes involved in the rotation before we actually carry it out. Clearly a rotation on nodes with "reversed" flags turned off keeps the order of the permutation.

We implement queries by maintaining subtree counts in the nodes. If we want the value of π_i we just look for the i^{th} node in the (altered) in-order traversal and return the element in it, with the correct sign according to the reversed flags of the nodes of the path that led us to it. Given a pointer to the node x containing i we can find π_i^{-1} by counting how many nodes are to the left of the path from x to the root, again taking into account the reversed flags. The value of π_i^{-1} is one more than that number. We get that pointer to node x by keeping, on the side, an array of pointers such that given an element i we can get the node x that contains it. This array does not need maintenance since an element stay with the same node at all times. All of these operations are done while splaying for the corresponding node, and since their cost is proportional to the length of the path from the root to that node then that cost can be assigned to the splay. This proves the logarithmic running time.

To execute a reversal $\rho(i, j)$ we split the tree at the j^{th} element to a tree T_1 containing all items with indexes smaller than j including j, and to a tree T_2 containing all items indexed more than j. Then we split T_1 at the i^{th} element to a tree T_3 containing all items indexed less than i and to a tree T_4 containing all items indexed at least i. Finally we flip the "reversed" flag of the root of T_4, concatenate T_3 to T_4 and the resulting

tree to T_2. It is easy to see that the resulting tree represents the new permutation after we performed the reversal. The $O(\log n)$ time bound follows from the logarithmic time bounds on split and concatenate.

The second data structure is in many ways a "Two-Layered" version of the previous structure. The structure is based on partitioning the permutation into blocks, each of size $\Theta(\sqrt{n})$. Each block is a contiguous fragment of the permutation, and is accompanied by a *'reversed'* flag, which, if *on*, indicates that the block should be read in reversed direction and the elements have opposite signs. That way, we can reverse a block simply by flipping its *reversed* flag. We maintain the blocks in a list, where each block points to a list of the items it contains (and also stores its size) and each item points to the block containing it. See Figure 2. We also maintain the following invariant.

Invariant 1 *At the end of each operation the number of elements in each block is between* $\frac{1}{2}\sqrt{n}$ *and* $2\sqrt{n}$.

Note that this implies that there are always $\Theta(\sqrt{n})$ blocks, each of size $\Theta(\sqrt{n})$. It is easy to see that one can initialize the data structure in linear time.

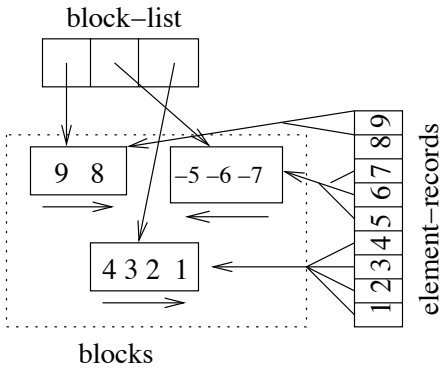

Fig. 2. A representation of the permutation $\pi = (9, 8, 7, 6, 5, 4, 3, 2, 1)$

We perform a reversal by the following three steps.

1. The extreme elements of the reversal are found, as well as the blocks containing them. We split each of these two blocks that also contains elements that do not belong to the reversal into two blocks. One of the resulting blocks contains only elements that belong to the reversal, and the other contains only elements that do not belong to the reversal. Note that this may violate Invariant 1. We reinsert the blocks to the block-list so that the blocks that contain the elements belonging to the reversal are consecutively placed.
2. Now the reversal consists of a subsequence of complete blocks. We reverse the order of those blocks in the block-list, and flip their reversed flags.
3. Finally, we reinstate the Invariant – we look at the four blocks that may now be of size smaller than $\frac{\sqrt{n}}{2}$, and we merge each of them to their neighbors until they are all above $\frac{\sqrt{n}}{2}$. Now some blocks may be larger than $2\sqrt{n}$ (but all are still smaller than $3\sqrt{n}$). We split the large blocks to smaller ones that satisfy the invariant.

It is easy to see that using our representation we can perform these three steps in $O(\sqrt{n})$ time. Figure 3 shows an example of this process.

It is easy to see how we can use this data structure to answer queries and inverse queries in time $O(\sqrt{n})$. However by maintaining some additional information we can answer queries even faster. To speed up queries we maintain the list of blocks in a search tree where the key of each block is the number of elements in blocks to its left (these are the "running totals" of the block sizes). Within each block we maintain the items in

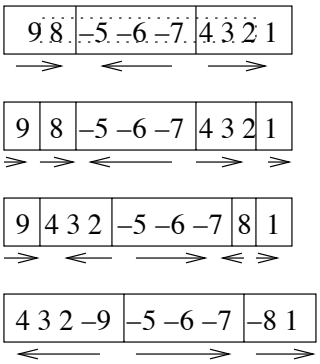

Fig. 3. Performing the reversal $\rho(2,8)$ on $\pi = (9,8,7,6,5,4,3,2,1)$ results in $\pi' = (9,-2,-3,-4,-5,-6,-7,-8,1)$. Note that I allow blocks' sizes to be between 2 and 6.

an array. With this representation we can locate the block we are looking for by performing a search on the tree, and then go directly to the required element within the block.

In order to implement an inverse query in time $O(1)$ we maintain with each element its index within its block and maintain with each block the number of elements in blocks preceding it (this is the key of the block in the search tree mentioned above). With this representation we can get the location of an element by adding up its index to the key of its block.

3.2 Adding the Ability to Draw a Random Oriented Pair

We will now give a data structure based on the second result of the previous section. Like it, the new data structure divides the permutation into blocks as well. We maintain the old structure but add to each block a tree whose purpose is to store the pairs related to that block, and maintain their orientations. Each of these trees will be similar to a single instance of the first data structure of the previous section — it will be a balanced binary search tree based on rotations, such as a Splay Tree. Each node in the tree will hold a pair and a boolean that indicates whether the pair is oriented, as well as a "reversed" flag whose function will be described shortly.

Our data structure will take linear time to initialize, $O(\sqrt{n \log n})$ time to execute a reversal, constant time to find the number of oriented pairs in π, and logarithmic time for selecting an oriented pair uniformly at random. Queries on π retain their logarithmic running time.

As we described, each block now also holds a tree. The tree contains, for each element x in the block, the pair whose elements are $\pm x$ and $\pm(x+1)$.[5] An in-order read of the tree will order the pairs according to the location of their other end in the permutation (that is, according to increasing order of $\pi^{-1}(\pm(x+1))$). With each pair we also store its orientation. As was already stated, the tree will be of similar structure and function as the one that was used in the first data structure of the previous section, i.e. a balanced search tree that supports split and concatenate operations, and relies on rotations. Each node will have a "reversed" flag. If the "reversed" flag of a node v is *on*, it means, like last section, that the subtree T_v rooted at v should be read in reversed order, and it also means that the orientations of the pairs in T_v is opposite to what is listed. As before, further "reversed" flags down the tree can once again alter the orientations and the order. We also associate an additional meaning to "reversed" flag of the *block* itself – now if a block's "reversed" flag is *on* it also means that the orientations of the pairs associated with the block's elements (that is, the orientations of the pairs in the block's tree) are once again opposite to what is listed.

As in last section, we facilitate fast query times by maintaining subtree counts of the oriented pairs in the nodes (these can be updated while doing the rotations, as before) and adding "running totals" of oriented pairs to the search tree of the block-list, where we used to keep the running totals of the block sizes. Using these it is easy to return the number of oriented reversals in constant time (it is listed in the "grand total" which is the last of the running totals), and to uniformly draw a random oriented pair in time $O(\log n)$ (by

[5] Except, of course, for the element $x = n+1$ which does not have a pair associated with it.

randomly selecting the index of the oriented pair we are looking for, searching on the running totals to find the tree it is in, and then looking for the right index in the tree).

To execute a reversal $\rho(i,j)$ we operate on the blocks like we did in the base data structure. When blocks are split and united we rebuild the trees associated with them "from scratch". This is not too costly because there is only a small number of pairs in those trees. This leaves us with the problem of updating the trees, (both those associated with blocks that are in the reversal and those associated with blocks that are out of the reversal) so that the orientations and the order of pairs inside the trees will be correct, in the post-split setting in which each block is either entirely in the reversal or entirely out of it.

To complete the execution of $\rho(i,j)$ we go over all blocks. In each block we look at the tree: We split this tree to three trees, much like we did in the first data structure of the previous section. The middle tree will contain all pairs whose "remote" element is part of the reversal (i.e. all pairs for which $\pi^{-1}(\pm(x+1)) \in [i,j]$), and the other two trees will contain the pairs whose "remote" element is out of the reversal. We then flip the "reversed" flag of the root of the middle tree, and concatenate the trees in their original order. After this operation, the order of the pairs is correct for all trees. Regarding the orientations, they are correct for trees in blocks that do not belong in the reversal, while in the blocks that do belong in the reversal, the orientations are exactly opposite of their correct setting. This happens because the orientation is flipped exactly for those pairs for which one of the elements can be found inside the reversal and the other is outside of it. However, note that we expanded the role of the blocks' "reversed" flags to indicate that the orientations of the block's pairs are flipped once again, and this can be seen to fix the orientations of pairs in those blocks that *are* in the reversal so that they are now at the correct state as well. Thus, once we have done this splitting, flipping, and concatenating, all invariants are satisfied.

In order to fine-tune the asymptotic cost of performing a reversal in this structure we keep the blocks' sizes between $\frac{\sqrt{n \log n}}{2}$ and $2 \cdot \sqrt{n \log n}$, and this makes the entire procedure of performing a reversal take $O(\sqrt{n \log n})$ time for the operations on the original data structure (including the costs for rebuilding trees that belong to blocks that were split and united), and additional $O(\sqrt{n \log n})$ time for the procedure detailed above, which takes logarithmic time for each of $\Theta(\sqrt{\frac{n}{\log n}})$ trees. Therefore, $O(\sqrt{n \log n})$ is the total running time of executing an oriented reversal, as promised.

4 Conclusion and Further Questions

We described a randomized algorithm for SBR that gives very good results on random permutations. To implement this algorithm efficiently we give various data structures to maintain a permutation under reversals such that it is possible to draw a random oriented reversal, and to perform other queries. Our data structures support all operations in sub-linear time and may be useful for other algorithms trying to solve SBR in sub-quadratic time.

In addition to proving Conj. 1, another problem that is left open is expanding the data structures of Sec. 3.2 to maintain some more data about the permutation in order to get a worst-case (or almost worst-case) sub-quadratic algorithm. This could be done, for example, by adding to the data structure the ability to check whether the permutation it represents contains an unoriented component in sub-linear time. That would mean that we can perform a walk as before, but checking at each step retroactively whether the oriented reversal we performed was safe or not, and if it was not safe we can backtrack and try to find another one. This random walk would give an algorithm that takes $o(n^2)$ expected time on *all* permutations, provided we can prove that at least a constant fraction of oriented reversals is always safe (a fact that we have witnessed empirically).

Acknowledgments

We thanks Michal Ozery and Ron Shamir for showing us the example in Section 2. We also thank Ron Shamir for many helpful discussions.

References

1. David A. Bader, Bernard M. E. Moret, and Mi Yan, *A linear-time algorithm for computing inversion distance between signed permutations with an experimental study*, Workshop on Algorithms and Data Structures, 2001, pp. 365–376.
2. V. Bafna and P. A. Pevzner, *Genome rearragements and sorting by reversals*, SIAM Journal on Computing **25** (1996), no. 2, 272–289.
3. Anne Bergeron, *A very elementary presentation of the hannenhalli-pevzner theory*, CPM (2001), 106–117.
4. Anne Bergeron, Cedric Chauve, Tzvika Hartman, and Karine Saint-Onge, *On the properties of sequences of reversals that sort a signed permutation*, JOBIM, June 2002, pp. 99–108.
5. A. Caprara, *Sorting by reversals is difficult*, Proceedings of the First International Conference on Computational Molecular Biology (RECOMB), ACM Press, 1997, pp. 75–83.
6. M. Chrobak, T. Szymacha, and A. Krawczyk, *A data structure useful for finding hamiltonian cycles*, Theoretical Computer Science **71** (1990), no. 3, 419–424.
7. Fredman, Johnson, McGeoch, and Ostheimer, *Data structures for traveling salesmen*, SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1993.
8. M. C. Golumbic, *Algorithmic graph theory and perfect graphs*, Academic Press, 1980.
9. S. Hannenhalli and P. A. Pevzner, *Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals)*, Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing (Las Vegas, Nevada), 29 May–1 June 1995, pp. 178–189.
10. Haim Kaplan, Ron Shamir, and Robert Endre Tarjan, *A faster and simpler algorithm for sorting signed permutations by reversals*, SIAM J. Comput. **29** (1999), no. 3, 880–892.
11. J. Kececioglu and D. Sankoff, *Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement*, Algorithmica **13** (1995), no. 1/2, 180–210, A preliminary version appeared in *Proc. CPM93*, Springer, Berlin, 1993, pages 87–105.
12. Michal Ozery and Ron Shamir, private communication.
13. Pavel A. Pevzner, *Computational molecular biology: An algorithmic approach*, The MIT Press, Cambridge, MA, 2000.
14. D. Sankoff, R. Cedergren, and Y. Abel, *Genomic divergence through gene rearrangement.*, Methods in Enzymology **183** (1990), 428–438.
15. David Sankoff, *Edit distance for genome comparison based on non-local operations*, Lecture Notes in Computer Science **644** (1992), 121–135.
16. D.D. Sleator and R.E. Tarjan, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach. **32** (1985), 652–686.