Sorting signed permutations by inversions in $O(n \log n)$ time

Krister M. Swenson, Vaibhav Rajan, Yu Lin, and Bernard M.E. Moret

Laboratory for Computational Biology and Bioinformatics
EPFL (École Polytechnique Fédérale de Lausanne), Switzerland
{krister.swenson,vaibhav.rajan,yu.lin,bernard.moret}@epfl.ch

Abstract. The study of genomic inversions (or reversals) has been a mainstay of computational genomics for nearly 20 years. After the initial breakthrough of Hannenhalli and Pevzner, who gave the first polynomial-time algorithm for sorting signed permutations by inversions, improved algorithms have been designed, culminating with an optimal linear-time algorithm for computing the inversion distance and a subquadratic algorithm for providing a shortest sequence of inversions—also known as sorting by inversions. Remaining open was the question of whether sorting by inversions could be done in $O(n \log n)$ time.

In this paper, we present a qualified answer to this question, by providing two new sorting algorithms, a simple and fast randomized algorithm and a deterministic refinement. The deterministic algorithm runs in time $O(n \log n + kn)$, where k is a data-dependent parameter. We provide the results of extensive experiments showing that both the average and the standard deviation for k are small constants, independent of the size of the permutation. We conclude (but do not prove) that almost all signed permutations can be sorted by inversions in $O(n \log n)$ time.

1 Introduction

Genomic rearrangements have been the subject of intense research over the last 10 years. Initially identified in the 1920s in the fly genome through genetic studies [17, 18], then studied in detail in chloroplast organelles in

the 1980s (for instance in a series of papers from Palmer's lab, beginning with [12, 13]), they were brought to the attention of the computational community in the early 1990s [14]. A large number of papers have since been published on the combinatorics and algorithmics of genomic rearrangements (see [11] for a survey and [7] for a thorough mathematical treatment). Starting at the beginning of this century, genomic rearrangements have assumed much more importance with the advent of whole-genome sequencing and the emergence of comparative genomics as a major discipline in biocomputing.

Of the various genomic rearrangements studied, perhaps the simplest and best documented is the *inversion* (also called reversal in much of the Computer Science literature), through which a segment of a chromosome is reversed in place. In 1987, Day and Sankoff [6] formalized a model of genomic inversions in which a chromosome is represented as a permutation of signed gene indices, the sign indicating the direction of transcription of the gene; in this framework, an inversion acts on an interval of the permutation by reversing the order in which the indices appear within the interval and by flipping the sign of each index. Sankoff later provided a probabilistic model [15] and posed two fundamental questions about inversions in this framework: given two signed permutations on the same index set, what is the smallest number of inversions required to transform one permutation into the other and what is a sequence of inversions implementing this transformation [14]. The first problem is thus to compute

an edit distance, where the edit operation is the inversion; the second is to return an edit sequence—a problem usually known as "sorting," since a simple re-indexing can turn one of the permutations into the identity. Many years of work were needed to ascertain the complexity of each of these problems. The breakthrough came in 1995, when Hannenhalli and Pevzner provided a a polynomial-time algorithm to solve both problems. (In contrast, in 1997, Caprara [5] showed that both problems were NP-hard if phrased in terms of unsigned permutations.) The running time for both problems has been steadily reduced over the years. In 2001, our group gave an optimal linear-time algorithm to compute the edit distance [1]; and in 2004, Tannier and Sagot [20], building on the work of Kaplan and Verbin [9], gave a $O(n\sqrt{n\log n})$ algorithm to produce a sorting sequence. Remaining open was the question of whether signed permutations can be sorted by inversions in $O(n\log n)$ time, just like sorting plain numbers.

In this paper, we give a qualified positive answer to this question by describing two new algorithms for sorting signed permutations by inversions. The first is a randomized algorithm that runs in guaranteed $O(n\log n)$ time, but may fail; successive restarts reduce the probability of failure, but we cannot guarantee that every permutation will be sorted with high probability with a finite number of restarts, so that it is not a true Las Vegas algorithm. (Indeed, we give a family of permutations that cannot be sorted by this algorithm regardless of the number of restarts.)

The other is a deterministic algorithm that always sorts the permutation and runs in $O(n \log n + kn)$ time, where k is the number of successive "corrections" (detailed in Section 5) that must be applied—a value, incidentally, that is not related to the edit distance d, although it is bounded by it. We give a family of permutations for which k is $\Theta(n)$ (the worst-case value for k) and thus for which our sorting algorithm will run in quadratic time. However, we present the results of very extensive experimentation showing that the expected value and the standard deviation of k are small constants (less than 1), independent of n, so that the running time of the algorithm is, with high probability, $O(n \log n)$. Thus we conclude (but do not prove) that almost all permutations can be sorted in optimal $O(n \log n)$ time.

2 Preliminaries

A permutation π is written as $(\pi_1 \pi_2 \ldots \pi_n)$, where each element π_i is a signed integer and the absolute values of these elements are all distinct and form the set $\{1, 2, \ldots, n\}$. The absolute value of π_i is denoted by $|\pi_i|$. An *inversion* $\rho(i, j)$ on a permutation $\pi = (\pi_1 \ldots \pi_i \ldots \pi_j \ldots \pi_n)$ reverses all elements between π_i and π_j while changing their signs giving $(\pi_1 \ldots \pi_{i-1} {}^{-}\pi_j \ldots {}^{-}\pi_i \pi_{j+1} \ldots \pi_n)$. We assume that every permutation of n elements is framed by elements 0 and $n+1$. In this way we consider each permutation to be linear, noting that each linear permutation corresponds to $n+1$ circular permutations (of length $n+1$), which are equivalent in

terms of the sequences of inversions used to sort them. The *span* of an inversion $\rho(i,j)$ is the closed interval on the natural numbers $[i,j]$ and two spans $[i,j]$ and $[k,l]$ *overlap* if and only if either $i<k$ and $k<j$ or $k<i$ and $j<l$.

Two adjacent elements, π_i and π_{i+1} for $0 \leq i \leq n+1$, form an *adjacency*. An adjacency is a *non-breakpoint* if and only if we have $\pi_{i+1} - \pi_i = 1$, otherwise it is a *breakpoint*. An *oriented pair*, (π_i, π_j), in a permutation is a pair of integers with opposite signs such that $\pi_i + \pi_j = \pm 1$. The inversion induced by an oriented pair (π_i, π_j), called an *oriented inversion*, is $\rho(i, j-1)$ for $\pi_i + \pi_j = +1$, and $\rho(i+1, j)$ for $\pi_i + \pi_j = -1$. An oriented inversion always creates a non-breakpoint; we say that it *heals* the breakpoint (or breakpoints—there could be two) to which the elements of the oriented pair belonged before the inversion.

A *framed common interval* (FCI) [2] of a length n permutation is a substring of the permutation, $(as_1s_2\ldots s_kb)$ or $(^-bs_1s_2\ldots s_k{}^-a)$ (with $s_1s_2\ldots s_k$ possibly empty) such that

- for each i, $1 \leq i \leq k$, $|a| < |s_i| < |b|$,
- for each l, $|a| < l < |b|$, there exists a j, $1 \leq j \leq k$, with $|s_j| = l$, and
- the FCI is not a union of shorter intervals with the above properties.

The substring $s_1s_2\ldots s_k$ is thus a (possibly empty) signed permutation of the integers greater than a and less than b; elements a and b are called the *frame* elements. The framed interval is said to be common in that it also exists, in its canonical form $\left({}^+a{}^+(a+1){}^+(a+2)\ldots{}^+b\right)$, in the identity

permutation. FCI B is *nested* inside FCI A if and only if the left and right frame elements of A occur, respectively, before and after the frame elements of B.

A *component* is comprised of the frame elements from an FCI along with all elements inside the FCI that are not used for a nested subinterval. A *non-trivial component* is a component that is comprised of at least 4 elements. A *bad component* is a component where all elements have the same sign. Two components can only overlap at the frame elements [3]. An inversion is said to be *unsafe* if it creates a bad component, otherwise it is *safe*. A permutation is *positive* if it is not the identity permutation and every element is positive. A positive permutation indicates the existence of at least one bad component. Any permutation containing bad components can be transformed to another permutation that does not contain any bad component in linear time [1]. Thus, in the algorithms we describe, we assume that the input permutation does not contain any bad components.

3 Background: Data Structures for Permutations

To implement an algorithm for sorting by inversions, we need a data structure for handling permutations that supports two basic operations: (i) choose an oriented inversion, and (ii) perform an inversion.

We now describe the data structure of Kaplan and Verbin [9] that stores a permutation in linear space and allows us to perform an inversion

in logarithmic time. The structure is a splay tree, in which the nodes are ordered by the indices of the permutation, with one additional flag maintained at each node.

To perform an inversion $\rho(i, j)$ between (and including) indices i and j, index $i-1$ is splayed and the right subtree of the root is split from the root yielding subtrees $T_{<i}$ and $T_{\geq i}$ where $T_{<i}$ ($T_{\geq i}$) contains all elements with indices less than (greater than or equal to) i. Next, index j is splayed in $T_{\geq i}$ and again the right subtree is split from its root yielding subtrees T_{rev} and $T_{>j}$ where $T_{>j}$ contains all elements with indices greater than j and T_{rev} contains the elements of the permutation that have to be reversed. Finally, there are three subtrees: $T_{<i}$, T_{rev} and $T_{>j}$. Now, actually reversing the elements in T_{rev} can take $\Theta(n)$ time since $\Theta(n)$ elements could be reversed in a single inversion. To achieve logarithmic time complexity a lazy approach is taken: a *reversed* flag is maintained in each node, which if turned on indicates that the subtree rooted at the node is reversed. Now instead of immediately reversing a subtree, we just set its reversed flag. During an inversion the reversed flag of the root of T_{rev} is flipped and $T_{<i}$ is joined to T_{rev} to get $T_{\leq j}$. This is achieved by making T_{rev} the right child of the root of $T_{<i}$, which still contains the element at index $i-1$, yielding the tree $T_{\leq j}$. $T_{\leq j}$ is then joined to $T_{>j}$ by splaying j in $T_{\leq j}$, after which $T_{>j}$ is made the right child of the root of $T_{\leq j}$, yielding the final tree which represents the permutation after the inversion. Since the only operation that takes more than constant time is

the splay and since splaying takes amortized logarithmic time [16], each inversion takes amortized logarithmic time.

A tree could have several reversed flags, but the invariant maintained is that an inorder traversal modified by the reversed flags yields the permutation. So to read the permutation one would traverse a reversed subtree in reverse order while flipping signs of elements read. Nested reversed flags cancel in the sense that a reversed flag on a node within a reversed subtree, implies that the inner subtree (rooted at that node) is not reversed. Thus, a subtree rooted at a node is reversed if and only if there is an odd number of reversed flags in the path from the root to the node (including the node).

When a sequence of inversions is performed, reversed flags can get nested to arbitrarily deep levels. We can push the flag down a traversed path in the tree, by flipping the sign of the element in the node, exchanging the left and right subtrees, and flipping the reversed flags in both children. The reversed flag of a leaf is cleared by just flipping its sign. Pushing down a flag takes constant time per node so the logarithmic time complexity of splaying is maintained. By pushing down the flags in the splay path we ensure that the three subtrees created ($T_{<i}$, T_{rev} and $T_{>j}$) reflect the changes made in all the previous inversions.

This is exactly the data structure described in [9]; it can handle a sequence of d inversions in $O(d \log n)$ time. The data structure maintains only the state of the permutation at each step (in a lazy way). However it

does not maintain information about oriented pairs, nor could it do so efficiently, as a single inversion could change the orientation of $\Theta(n)$ pairs. Indeed, using this data structure to maintain the information necessary to choose an oriented inversion at each step would increase the running time by a factor of n.

To overcome this problem both Kaplan and Verbin [9] and Tannier *et al.* [19] used a two-level version of the data structure in which a permutation is stored in linear blocks of size $O(\sqrt{n \log n})$ each. Corresponding to each block is a splay tree that maintains information about all oriented pairs (π_i, π_j) such that either π_i or π_j is in the block. Performing an inversion while maintaining information about all oriented pairs takes $O(\sqrt{n \log n})$ time and choosing an inversion at each sorting step takes $O(\log n)$ time, so that the total time complexity of their algorithms is $O(n\sqrt{n \log n})$.

In order to run in $O(n \log n)$ time, these algorithms need to be able to choose an oriented inversion in logarithmic time and thus information to identify such inversions must also be maintained in logarithmic time through an inversion.

4 Our Algorithm

Instead of addressing the data structure (by designing a new data structure that can somehow process $O(n)$ new pair orientations in logarithmic time), we address the root question of identifying an oriented inversion.

Our key contribution is that we need not maintain information about *all* oriented inversions for every permutation at each sorting step—a few suffice in most cases.

4.1 MAX inversions

Definition 1. *Let (π_i, π_j) be an oriented pair in a permutation and let π_j be the negative element in the pair. The oriented inversion corresponding to (π_i, π_j) is a* MAX inversion *if π_j has the maximum value of all negative elements in the permutation. The pair (π_i, π_j) is called the* MAX pair *of the permutation.*

For example the MAX inversion in the permutation $(4\,5\,{-3}\,1\,{-6}\,2\,{-7})$ is $\rho(4,6)$, corresponding to the oriented pair $(2,{-3})$, and the MAX inversion in the permutation $(2\,3\,{-1}\,{-4})$ is $\rho(1,3)$, corresponding to the oriented pair $(0,{-1})$. We maintain information about only the MAX inversions in the data structure and correspondingly perform a MAX inversion in each sorting step. The result is algorithm MAX.

Algorithm 1 MAX

1: **while** there exists a negative element in the permutation **do**
2: Find index of maximum negative element π_j.
3: Find index of $\pi_i = |\pi_j| - 1$.
4: Perform inversion corresponding to oriented pair (π_i, π_j).
5: **end while**

Because any permutation that contains a negative element contains a MAX inversion and because any sequence of oriented safe inversions is optimal [8], we can conclude as follows.

Lemma 1. *In the absence of unsafe MAX inversions at any sorting step, algorithm MAX produces an optimal sorting sequence.*

Algorithm MAX fails to sort only when it is "stuck" at an all-positive permutation that is not the identity, which happens when a MAX inversion was unsafe. (We deal with unsafe inversions in the next section.) The same arguments hold *mutatis mutandis* if we choose an oriented pair with the minimum negative element, yielding another algorithm, algorithm MIN. Combining the two strategies and picking one at random at each step gives us a randomized algorithm: algorithm RAND.

Algorithm 2 RAND

while there exists a negative element in the permutation **do**
  randomly select either MAX or MIN
  if MAX **then**
    Find index of maximum negative element π_j.
    Find index of $\pi_i = |\pi_j| - 1$.
    Perform inversion corresponding to oriented pair (π_i, π_j).
  else if MIN **then**
    Find index of minimum negative element π_k.
    Find index of $\pi_l = |\pi_k| + 1$.
    Perform inversion corresponding to oriented pair (π_k, π_l).
  end if
end while

4.2 Maintaining information through an inversion

We now show how to maintain information about the maximum negative element of a permutation through an inversion using the splay tree

data structure. We describe the process for MAX, but the obvious analog works for MIN.

Let the maximum negative element of a subtree, MAX_{neg}, be the element in the subtree that has the maximum value among all negative elements in the subtree. The minimum positive element, MIN_{pos}, of a subtree is defined similarly. These values are stored in each node of the splay tree. Note that the MAX_{neg} of the root node is the maximum negative element of the permutation, that is, the negative element of the MAX pair of the permutation. The MAX_{neg} of a node is the maximum of the following three: the MAX_{neg} of the left subtree, the MAX_{neg} of the right subtree, and the element in the node if the element is negative. Also notice that whenever the reversed flag of a node is turned on, MAX_{neg} and MIN_{pos} are swapped. Therefore pushing down a reversed flag applies this swap to the children, unless there is a cancellation of flags.

A splay operation performs a series of rotations based on the structure of the tree and the index being queried. Each rotation changes at most three edges of a connected subtree while maintaining the binary search tree property. MAX_{neg} can be recalculated for only the subtree that is affected, Recall that to perform an inversion $\rho(i, j)$ the splay tree is split into three subtrees which are rejoined after the reversed flag has been set for one of the trees. The value of MAX_{neg} can be kept for each of the subtrees in the process by simply checking the children of the root after each operation.

By maintaining the MAX_{neg} values in this fashion, one can maintain the invariant that the MAX_{neg} of the root node is the maximum negative element of the permutation through any sequence of inversions. Since calculating MAX_{neg} takes $O(1)$ time per node, these modifications do not alter the time complexity of the data structure.

Lemma 2. *For any (signed) permutation of size n, there exists a data structure that handles an inversion in $O(\log n)$ time while maintaining information about the maximum negative element of the permutation.*

4.3 Finding the MAX pair

We now describe how to obtain the elements of the MAX pair in a permutation using the modified data structure described above.

First the maximum negative element of the permutation is located. If the element in a node is not equal to the MAX_{neg} of the node then MAX_{neg} of the node lies in either the left subtree or the right subtree of the node. Therefore starting at the root one can go down the tree looking for the maximum negative element. Reversed flags must be pushed down along the path to ensure that MAX_{neg} values are updated and the correct path is followed.

To find the second element of the MAX pair, a lookup vector of pointers (of n elements) maps each element to the node that contains the element. These pointers do not change throughout the computation and

enable constant-time lookup of the node containing the second element of the MAX pair.

4.4 Finding the indices of the MAX inversion

In absence of reversed flags, the indices of the MAX inversion can be obtained directly from the current location of the nodes corresponding to the MAX pair. However, the presence of a reversed flag indicates nodes that have outdated indices, forcing additional work to retrieve the correct indices.

The index of a node (with respect to the current state of the permutation) can be calculated using the index of the parent node and the sizes of the left and right subtrees. Thus the current index of a node can be calculated whenever the reversed flag is pushed down from it. The size of the subtree rooted at a node is easily maintained. If the node is a right child, then its index is one more than the sum of its parent's index and the size of the left subtree. If the node is a left child, then its index is one less than the difference of its parent's index and the size of the right subtree. The index of the root is just the size of its left subtree. Thus starting at the root, as the reversed flags are pushed down along any path in the tree, the current indices can be calculated.

As one traverses the tree from the root searching for the maximum negative element, the indices are recalculated. After the node corresponding to the second element in the MAX pair is found using the lookup

vector, its updated index can be retrieved by traversing up to the root (using parent pointers) and returning down the same path, pushing down the reversed flags and recalculating indices at each node.

4.5 Putting it all together

The previous subsections detail all the steps for performing a MAX inversion. The time complexity of each of these steps is easy to analyze. Pushing down the reversed flag takes $O(1)$ time per node. Thus, finding the maximum negative element and its updated index takes $O(\log n)$ time. Finding the other element of the MAX pair takes $O(1)$ time and obtaining its updated index takes $O(\log n)$ time. Therefore the complexity of finding the two indices (steps 2 and 3 in algorithm MAX) is $O(\log n)$. For each inversion, maintaining MAX_{neg}, MIN_{pos}, MIN_{neg}, and MAX_{pos} in the nodes takes $O(1)$ time during split and join operations, and $O(1)$ time for each rotation in the two splays. Therefore performing the inversion in step 4 of algorithm MAX takes $O(\log n)$ time. So we have proved:

Theorem 1. *For any signed permutation of size n, a data structure exists that*

- *allows checking whether there exists an oriented inversion in $O(1)$ time,*
- *allows performing a MAX (or MIN) inversion, while maintaining the permutation, in $O(\log n)$ time,*
- *and is of size $O(n)$.*

Theorem 2. *In the absence of unsafe inversions at any sorting step, algorithm MAX produces an optimal sorting sequence in $O(n \log n)$ time.*

5 Bypassing Bad Components

We saw that algorithms MAX and RAND can get stuck at a positive permutation by choosing an unsafe inversion. We offer two strategies for recovery.

5.1 Randomized restarts

For algorithm RAND we can simply restart the computation hoping that a better outcome is met in the next run. Indeed, the experiments from Section 6 show that, for most permutations, this simple approach suffices. However, this approach cannot always sort a permutation as there exists a family of permutations that it cannot handle. For instance, take the permutation (3 1 4 2): both MAX and MIN inversions are unsafe because they yield the same positive permutation (3 1 2 4); this small example can be extended to any length by appending the requisite number of positive elements.

5.2 Recovering from an unsafe inversion: Tannier and Sagot's approach

Tannier and Sagot [20] introduced a powerful approach for finding unsafe inversions and augmenting the sorting sequence till it is optimal. They noticed that it is computationally difficult to detect an unsafe inver-

sion as it is applied; but it is of course trivial to find out that the process is stuck at a positive permutation. Their approach is thus *postmortem*: their algorithm traces the sorting process back to the most recent unsafe inversion and inserts two or more sorting inversions before the unsafe one without invalidating other sorting inversions. (This ensures that the sorting sequence grows in every trace-back phase.) After the trace-back, the sorting process continues from the state of the permutation just before the unsafe inversion. The new inversions that are inserted are chosen such that the bad component created by the previous unsafe inversion is no longer created and so, the (previously) unsafe inversion and all the inversions that followed it can be applied again.

They use an *overlap graph* to keep track of the remaining breakpoints (and whether or not they are oriented). Using the overlap graph they can find the most recent unsafe inversion, find and insert more inversions before the unsafe one, and continue sorting without invalidating the inversions that have been applied after the most recent unsafe inversion [20]. However, the process may have to be repeated, as, even after augmenting the sorting sequence, their algorithm may again get stuck at a positive permutation.

5.3 Recovering from an unsafe inversion: Our approach

We use the same general idea, but do not maintain the full overlap graph, as it is too expensive to maintain. Denote by p_1 the first positive permu-

tation at which the algorithm gets stuck and by p_i the i^{th} such positive permutation. Recovering from a positive permutation p_i involves three steps: finding the most recent unsafe inversion μ_i, finding and inserting two other oriented inversions with the required properties that can be applied before μ_i, and appending inversions without invalidating those sorting inversions that had been applied after (and including) μ_i. We describe each of these steps in turn.

Finding the most recent unsafe inversion: In the trace-back phase, we undo the inversions that have been done so far in order to find the most recent unsafe inversion μ_i. Note that an unsafe inversion is an inversion that, when undone, creates a good component from bad components. Denote by $\pi \cdot S$ and $\pi \cdot \rho$ the result of applying the inversions from the sequence of inversions S and the single inversion ρ to the permutation π, respectively. Let $U(\pi)$ be the set of unsafe inversions on a permutation π and let $B(\pi)$ be the set of bad components in $\pi \cdot \mu$ for $\mu \in U(\pi)$. *Undoing* the inversion ρ in $\pi \cdot \rho$ refers to performing ρ on $\pi \cdot \rho$ which yields π, and undoing the inversions $S = \rho_1, \rho_2, \ldots, \rho_n$ in $\pi \cdot S$ refers to performing the inversions of S in the reverse order which yields $\pi \cdot S \cdot \rho_n \ldots \rho_2 \cdot \rho_1 = \pi$. The sequence of inversions on input permutation π^0 that results in the positive permutation p_i is denoted by S_i, so $p_i = \pi^0 \cdot S_i$.

Remark 1. When undoing inversions from S_i, the most recent unsafe inversion μ_i is the first inversion met that turns an element in $B(p_i)$ from bad to good.

Finding μ_i is not trivial because framed intervals can be nested. For example the positive permutation (2 3 6 7 4 5 8 9 10 1) has two components: the one framed by the implicit frame elements 0 and 11, and the nested component framed by the elements 3 and 8. Undoing the inversion $\rho(2,7)$ will leave both bad components intact despite the fact that it occurs within the frame elements of the larger component. Thus, in the trace-back phase, $\rho(2,7)$ cannot be an unsafe inversion. However, undoing the inversions $\mu(5,7)$ and $\mu(4,5)$ will make the inner component good and so these two inversions, had they been performed, would have been unsafe. The following remark characterizes undoing an unsafe inversion in terms of the components in $B(p_i)$.

Remark 2. An inversion is the most recent unsafe inversion μ_i if and only if it is the most recent inversion to change the indices of a proper nonempty subset of the elements from some component in $B(p_i)$.

The trace-back algorithm is thus as follows: start undoing the inversion sequence S_i, checking after each inversion whether there exist components in $B(p_i)$ with both changed and unchanged indices and stop undoing when an unsafe inversion is found. This can be done by keeping an ancillary splay tree where nodes represent adjacencies in the permutation rather than permutation elements.

If every adjacency in p_i were a breakpoint, the most recent inversion would be unsafe; the heart of the problem, then, is with non-breakpoints and how they interact with the undoing of unsafe inversions. We present a labeling of the ancillary tree so that the safety check can be carried out by a constant-time comparison on the two adjacencies broken by an inversion. Each adjacency has a label indicating the innermost overlying component along with a label that is set only for non-breakpoints. For a given component, each group of consecutive non-breakpoints (ignoring nested components) gets a unique second label. Thus an inversion displaces only a fraction of the elements of a component if and only if both broken adjacencies are labeled as non-breakpoints with the same component and non-breakpoint labels.

In the example, the permutation (2 3 6 7 4 5 8 9 10 1) has component label X for adjacencies (0,2), (2,3), (8,9), (9,10), (10,1), and (1,11), and component label Y for the others. The non-breakpoint labels are the same for (2,3), (8,9), and (9,10), but different between (6,7) and (4,5). Inversion $\rho(2,7)$ acts upon non-breakpoints with the same pair of labels while inversion $\mu(5,7)$ acts upon non-breakpoints with different component labels and $\mu(4,5)$ acts upon non-breakpoints with different non-breakpoint labels.

We can list the endpoints of the components of a permutation in linear time [1, 2]. A simple traversal of the permutation, keeping one stack for each label, can perform the node labeling described above. Thus the setup

of the ancillary tree can be done in $O(n)$ time. Let S_i^1 be the sequence of inversions applied before μ_i in S_i and S_i^2 be the sequence of inversions applied after S_i^1 (including μ_i) in S_i. Each safe inversion in S_i^2 that is undone will cost $O(\log n)$ time so the total cost for finding a most recent unsafe inversion is $O(n + |S_i^2| \log n)$.

Inserting oriented inversions before the unsafe inversion: Theorem 3 in [19] shows that there always exists two oriented inversions $\nu 1_i$ and $\nu 2_i$ that are valid on the permutation after the unsafe inversion μ_i is undone in the trace-back phase. According to [19], if inversions $\nu 1_i$ and $\nu 2_i$ have the following properties then all the inversions after and including μ_i are safe, valid, and can be applied at the end of the sorting sequence on the i^{th} iteration:

- the span of $\nu 1_i$ overlaps the span of μ_i, and
- either the span of $\nu 2_i$ overlaps the span of $\nu 1_i$ and does not overlap the span of μ_i, or the span of $\nu 2_i$ overlaps the span of μ_i and does not overlap the span of $\nu 1_i$.

In the following we show how to find $\nu 1_i$ and $\nu 2_i$ in time proportional to the size of the bad component that we created.

Lemma 3. *Given an unsafe oriented inversion μ_i and the bad component b of size m created by μ_i, one can always find two inversions $\nu 1_i$ and $\nu 2_i$ in $O(m)$ time such that*

1. (existence) $\nu 1_i$ *and* $\nu 2_i$ *are valid sorting inversions when applied after* S_i^1.
2. (safety) *after applying* $\nu 1_i$ *and* $\nu 2_i$*, inversion* μ_i *does not create b.*
3. (validity) *after applying* $\nu 1_i$ *and* $\nu 2_i$*, inversion* μ_i *and all the inversions in* S_i^2 *remain valid sorting inversions and can be applied at the end of the sorting sequence of the* i^{th} *run.*

Proof. A bad component could have been created in one of three ways when μ_i was applied. Without loss of generality we ignore the symmetric counterpart to the first case below (both cannot happen at once). We also ignore the inverted versions of each case where the hurdle created has only negative elements. This leaves us with three cases to consider.

- $(^{\pm}\pi_0 \ldots {}^{+}\mathbf{l}{}^{+}\mathbf{x_1} \ldots {}^{+}\mathbf{x_s}\ \underbrace{{}^{\pm}\pi_x \ldots {}^{-}\mathbf{r}{}^{-}\mathbf{x_{k-1}} \ldots {}^{-}\mathbf{x_{s+1}}}\ {}^{\pm}\pi_{x+1} \ldots {}^{\pm}\pi_n)$
  where the braced inversion creates the bad component
  $b = {}^{+}l{}^{+}x_1 \ldots {}^{+}x_s{}^{+}x_{s+1} \ldots {}^{+}x_{k-1}{}^{+}r.$
- $(^{\pm}\pi_0 \ldots {}^{+}\mathbf{l}{}^{+}\mathbf{x_1} \ldots {}^{+}\mathbf{x_l}\ \underbrace{{}^{-}\mathbf{x_{r-1}} \ldots {}^{-}\mathbf{x_{l+1}}}\ {}^{+}\mathbf{x_r} \ldots \mathbf{x_{k-1}}{}^{+}\mathbf{r} \ldots {}^{\pm}\pi_n)$
  where the braced inversion creates the bad component
  $b = {}^{+}l{}^{+}x_1 \ldots {}^{+}x_l{}^{+}x_{l+1} \ldots {}^{+}x_{r-1}{}^{+}x_r \ \ldots {}^{+}x_{k-1}{}^{+}r.$
- The third case is the same as the first, except that one or more bad components are created which span the component ${}^{+}l{}^{+}x_1 \ldots {}^{+}x_s{}^{+}x_{s+1} \ldots {}^{+}x_{k-1}{}^{+}r.$

For the first case, write $L = {}^{+}l{}^{+}x_1 \ldots {}^{+}x_s$ and $R = {}^{-}r{}^{-}x_{k-1} \ldots {}^{-}x_{s+1}$ and examine the substrings L and R. Since the component $(l, \ldots, r)$ is a bad component, there must exist an element t in L such that either $t+1$ or $t-1$

is negative and not in L. Assume w is the first such element we encounter by scanning from ^{+}l to $^{+}x_s$. We locate the rightmost $^{-}(w-1)$ or $^{-}(w+1)$ in R by scanning from $^{-}x_{s+1}$ to ^{-}r. Now, there are two possibilities.

1. The rightmost element is $^{-}(w-1)$. We have $w > l+1$ and thus $(w, ^{-}(w-1))$ is an oriented pair; consequently, there exists an oriented inversion, $\nu 1_i$, which is different from μ_i. Now consider the position of those elements with absolute values between (and including) l and $w-1$. Let y be the element with the smallest value that does not appear to the left of w in L (such an element must exist because l is to the left of w but $w-1$ is in R). Thus $y-1$ must appear to the left of w in L. Not that y cannot be in R, as this would contradict the fact that w is the leftmost element in L with $^{-}(w+1)$ or $^{-}(w-1)$ in R. Thus y must be in L and to the right of w. After applying $\nu 1_i$, we will have the oriented pair $(y-1, ^{-}y)$, and consequently, another oriented inversion $\nu 2_i$. Notice that the span of $\nu 1_i$ overlaps the span of μ_i and the span of $\nu 2_i$ overlaps the span of $\nu 1_i$ but not that of μ_i. So, the required properties of safety and validity follow from Theorem 3 in [19].
2. The rightmost element is $^{-}(w+1)$. Note that $(w, ^{-}(w+1))$ is an oriented pair, so that there exists an oriented inversion $\nu 1_1$. This inversion must be different from μ_i as otherwise L would a bad component in itself. Now we examine the substring to the right of w in L. Let z be the element with the largest absolute value in that substring. Consider the following two cases:

(a) The absolute value of z is less than w: we consider the elements with absolute values in the interval $[l,z]$. Let y be the element with the largest absolute value in $[l,z]$ that appears to the left of w (such an element must exist because l is to the left of w but z is to the right of w in L). $y+1$ cannot be in R, as this would contradict the fact that w is the leftmost element in L with $\bar{}(w+1)$ or $\bar{}(w-1)$ in R. Thus $y+1$ must be in L and to the right of w. After applying $\nu 1_i$, we will have the oriented pair $(y,\bar{}(y+1))$, and consequently, another oriented inversion $\nu 2_i$. Notice that the span of $\nu 1_i$ overlaps the span of μ_i and the span of $\nu 2_i$ overlaps the span of $\nu 1_i$ but not that of μ_i. So, the required properties of safety and validity follow from Theorem 3 in [19].

(b) The absolute value of z is larger than $w+1$: We consider the elements with absolute values in $[z,r]$. Let y be the element with the largest absolute value in $[z,r]$ that appears to the left of $\bar{}(w+1)$ in R (such an element must exist because r is to the left of $\bar{}(w+1)$ in R but z is in L). $y-1$ cannot be to the left of w in L, as this would contradict the fact that w is the leftmost element in L with $\bar{}(w+1)$ or $\bar{}(w-1)$ in R. Thus $y-1$ must be either to the right of w in L or to the right of $\bar{}(w+1)$ in R. If $y-1$ is to the right of w in L, the oriented pair $(\bar{}(y-1),y)$ defines the oriented inversion $\nu 2_i$. Notice that the spans of $\nu 1_i$ and $\nu 2_i$ overlap the span of μ_i but $\nu 1_i$ and $\nu 2_i$ do not overlap. If $y-1$ is to the right of $\bar{}(w+1)$ in

R, after applying $\nu 1_i$, we will have the oriented pair $(y, {}^{-}(y-1))$, and consequently, another oriented inversion $\nu 2_i$. In this case the span of $\nu 1_i$ overlaps the span of μ_i and the span of $\nu 2_i$ overlaps the span of $\nu 1_i$ but not that of μ_i. So, the required properties of safety and validity follow from Theorem 3 in [19].

For the second case (where the span of the unsafe inversion is a proper subset of the span of the bad component), write $L = {}^{+}l\,{}^{+}x_1 \ldots {}^{+}x_l$, $M = {}^{-}x_{r-1} \ldots {}^{-}x_{l+1}$ and $R = {}^{-}r\,{}^{-}x_{k-1} \ldots {}^{-}x_{s+1}$. In substrings L and R, there must exist one element t such that ${}^{-}(t+1)$ or ${}^{-}(t-1)$ is in M and the inversion induced by this pair is not μ_i. Thus, the oriented pair $(t, {}^{-}(t-1))$ or $(t, {}^{-}(t+1))$ defines the oriented inversion $\nu 1_i$. Since $\nu 1_i$ is different from μ_i, there will be some negative elements after applying $\nu 1_i$; assume that the maximum negative element among them is ${}^{-}y$. Thus, $y-1$ must be positive and the oriented pair $({}^{-}y, {}^{+}(y-1))$ defines the other oriented inversion $\nu 2_i$. It is easy to verify that these inversions have the required properties.

The linear-time complexity can be achieved by using a lookup vector that maps each element to its index in the permutation. (This is created in the beginning and maintained throughout the sorting process.) Thus, for the first case, with a single scan of L, we can find w and ${}^{-}(w-1)$ and with another scan of elements between l and $w-1$ in the lookup vector, the pair $((y-1), {}^{-}y)$. The other cases can be analysed similarly. Note that

in no case do we need to scan any element that is not a part of b. Thus the inversions $\nu 1_i$ and $\nu 2_i$ can be found in $O(m)$ time.

Appending inversions to the sorting sequence: After we get the permutation $q_i = \pi \cdot S_i^1$, we apply the two inversions $\nu 1_i$ and $\nu 2_i$ on q_i. Now we would like to ensure that the sequence of inversions S_i' we append after $\nu 2_i$ does not invalidate the sequence $\mu_i \cdot S_i^2$. We achieve this by renaming the permutation q_i in the following way.

By definition, $q_i \cdot \mu_i$ has at least one bad component created by μ_i along with a possibly nonempty set $G(q_i \cdot \mu_i)$ of good components. The inversions that sort the components of $G(q_i \cdot \mu_i)$ correspond exactly to the sequence S_i^2. Thus, our desired sequence S_i' of inversions should only displace (if at all) such components without affecting their structure.

Say there is a component c of length m with left frame element l. The *canonical form* $\hat{c}$ of c is a permutation of length m with $\hat{c}[i] = c[i] - l + 1$, $1 \leq i \leq m$, where $p[i]$ denotes the ith element of a permutation p. Components c and d are said to be *structurally equivalent* if and only if we have $\hat{c} = \hat{d}$.

Lemma 4. *Let q_i be a permutation without a bad component and μ_i be an inversion such that $q_i \cdot \mu_i$ has at least one bad component and a set of good components $G(q_i \cdot \mu_i)$. There exists a q_i' where any sequence S_i' that sorts q_i' to the identity, when applied to q_i, will result in a permutation whose only components are those in $G(q_i \cdot \mu_i)$.*

Proof. Rename the permutation $q_i \cdot \mu_i$ such that all breakpoints from components in $G(q_i \cdot \mu_i)$ become non-breakpoints and then undo μ_i to get q_i'. Note that this renaming leaves one structurally equivalent bad component in place of each bad component, so that the renaming is unique. A inversion sequence that sorts q_i' to the identity heals all breakpoints from the bad components in $q_i \cdot \mu_i$; moreover, it does not heal any breakpoint from components of q_i that are in $G(q_i \cdot \mu_i)$ due to the nesting property of FCIs.

For example, take q_i = (2 3 6 7 4 -8 -5 -9 10 -1) and $\mu_i = \mu(6,7)$. Now $q_i \cdot \mu_i$ is (2 3 6 7 4 5 8 -9 10 -1), so that $G(q_i \cdot \mu_i)$ is comprised of the components framed by the pair (of frame elements) (0,11) and the pair (8,10). $q_i \cdot \mu_i$ is renamed to $q_i' \cdot \mu_i$ = (1 2 5 6 3 4 7 8 9 10), yielding q_i' = (1 2 5 6 3 -7 -4 8 9 10). The sorting sequence $S_i' = (\rho(3,6), \rho(3,4), \rho(4,7))$ for q_i' can be applied to q_i to get (2 3 4 5 6 7 8 -9 10 -1).

Lemma 5. *Given a permutation p with a set of bad components $B(p)$, permutation p', that has one structurally equivalent bad component in place of each $b \in B(p)$ and only non-breakpoints everywhere else, can be constructed in linear time.*

Proof. If an adjacency is not part of a bad component then label it with a null value; otherwise label it by the bad component of which it is part of. Also label adjacencies with the left and right endpoints of each component, which can be done in linear time [1, 2]. We use a stack R, the top

of which we denote by $top(R)$. Perform the following steps until the end of the permutation is reached, i.e., until we have $i = n$.

1. Label each element $p'[i]$ with the value $p'[i-1]+1$ until an adjacency corresponding to a bad component is encountered.
2. If the adjacency is a left endpoint, then push onto R the value $p[i-1]-p'[i-1]$ and go to step 3. If it is a right endpoint, then pop the top element. If the next breakpoint is labeled with a bad component, then go to step 3 otherwise go to step 1.
3. Label each element $p'[i] = p[i] - top(R)$ until an adjacency with a different component label is reached, then go to step 2.

The renaming procedure takes linear time and works correctly because every bad component is renamed to a structurally equivalent component in step 2.

Overall running time analysis: We call this algorithm, with the recovery phase included, MAX-RECOVER or RAND-RECOVER, depending on whether algorithm MAX or algorithm RAND is used in the forward-sorting phase. If algorithm MAX or RAND gets stuck at a positive permutation p_i, we proceed by undoing inversions until a permutation q_i is found such that $q_i \cdot \mu_i$ has fewer bad components than q_i. Finding such a q_i and μ_i alone takes $O(n+|S_i^2|\log n)$ time. The inversions undone in this step are not discarded as they can be applied after inserting at least two more inversions. Notice that each inversion undone in the trace-back

must be done or undone on a splay tree at most three times and that S_i^2 and S_j^2 for any two p_i and p_j, $i \neq j$, must be disjoint. Thus the $O(n \log n)$ term describes the amount of time spent for undoing inversions over the entire course of the algorithm and just a linear amount of work beyond that must be done in each recovery phase.

Theorem 3. *The running time of MAX-RECOVER or RAND-RECOVER is $O(n \log n + kn)$ where k is the total number of unsafe inversions performed in the algorithm.*

In Section 6 we show strong empirical evidence that, on random permutations of length n, the average value and standard deviation of k remain constant (about $\frac{1}{2}$) even as n grows very large, leading us to conjecture that these algorithms sort almost all permutations in $O(n \log n)$ time. In the worst case, however, RAND-RECOVER and MAX-RECOVER can use $\Theta(n^2)$ time, as in the following family of permutations: build a permutation of length n by starting with the identity permutation of length n mod 5 as the first block, followed by $n/5$ copies of the block $i(i+3)(i+1)(i+4)(i+2)(i+5)$, each of which shares its first element with the last element of the preceding block.

6 Experimental Results

We present experimental results for algorithms MAX, RAND, MAX-RECOVER and RAND-RECOVER. All of the experiments are on random permutations of length 100, 200, 500, 1000, 2000, 5000, 10,000 and

$20,000$. For each length, we tested our algorithms on $1,000,000$ permutations.

Table 1 lists the failure rates for algorithm MAX and algorithm RAND. Algorithm MAX and algorithm RAND produce an optimal sorting sequence with frequency 61%. We also include the failure rates for RAND-RESTART: the simple heuristic that runs RAND on the input permutation a second time if it fails to sort at the first attempt. The failure rate for RAND-RESTART reduces to 16% ($\approx 0.39 \times 0.39$), which suggests that the two runs are independent with respect to the failure rate.

Tables 2 and 3 summarize the details of the number of recovery steps, k, that we observe in algorithms MAX-RECOVER and RAND-RECOVER. The average value and the standard deviation of k remain constant as n grows. Figure 1 shows the distribution of k for MAX-RECOVER on random permutations of length $10,000$. This figure is representative of the observed distribution for the other lengths as well. The similarity to the inverse exponential function suggests that the upper bound for the average value of k is a constant. Finally, Figure 2 shows the running time of MAX-RECOVER run on randomly generated signed permutations of sizes 100 through $50,000$, normalized by the running time of mergesort run on an array of integers of matched size. The normalization makes it much easier to discern the asymptotic behavior—the ratio displayed should be $\Theta(1)$ and, in particular, it should not show any tendency to rise as n increases. Moreover, normalizing by the running

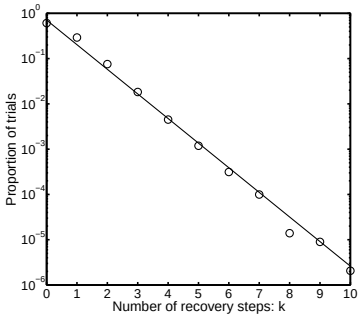

Fig. 1. The distribution of k for MAX-RECOVER on random permutations of length $10,000$.

time of another, well studied procedure that runs in the same time regardless of the input data helps in smoothing out small variations due to the memory hierarchy [10]. Figure 2 supports our conjecture, as running time ratios are tightly grouped and remain within the same range for all values of n tested. We also note that MAX-RECOVER runs fast: our implementation is not fine-tuned in any way and yet sorts permutations of size $50,000$ in 2 seconds on one core of a 4-processor, 16-core Dell PowerEdge R905 with 128GB of memory and 2.2GHz AMD 8354 processors running Linux.

7 Conclusions

We have given two new algorithms for sorting signed permutations by inversions, one a fast heuristic that works on most permutations, the other a deterministic algorithm that sorts all permutations and takes $O(n \log n)$ time on almost all of them. We have given the results of very extensive

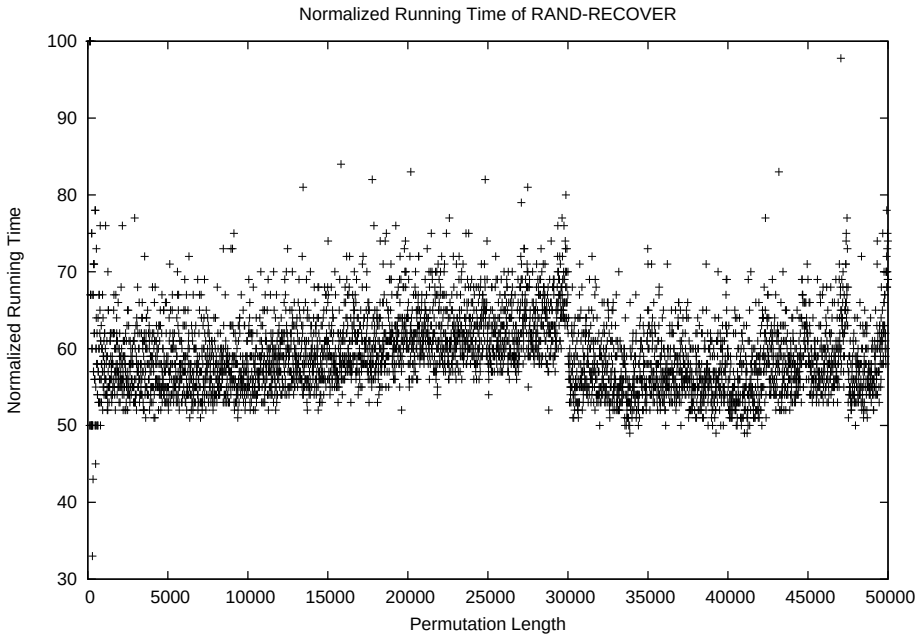

Fig. 2. Normalized running time of RAND-RECOVER on random permutations of sizes from 100 to $50,000$—shown is the ratio to the running time of mergesort on arrays of matching size.

experimentation to confirm these claims. We have thus taken a major step towards a final resolution of the sorting problem. Future work includes a formal proof that our deterministic algorithm sorts almost all permutations in $O(n \log n)$ time and designing an algorithm to deal with the few remaining permutations where our algorithm takes more time.

References

1. D.A. Bader, B.M.E. Moret, and M. Yan. A fast linear-time algorithm for inversion distance with an experimental comparison. *J. Comput. Biol.*, 8(5):483–491, 2001.
2. A. Bergeron, S. Heber, and J. Stoye. Common intervals and sorting by reversals: a marriage of necessity. In *Proc. 2nd European Conf. Comput. Biol. ECCB'02*, 54–63, 2002.
3. A. Bergeron and J. Stoye. On the similarity of sets of permutations and its applications to genome comparison. In *Proc. 9th Int'l Conf. Computing and Combinatorics (COCOON'03), Lecture Notes in Comp. Sci.* 2697, 68–79. Springer Verlag, Berlin, 2003.
4. A. Bergeron, J. Mixtacki, and J. Stoye. Reversal Distance without Hurdles and Fortresses. In *Proc. 15th Ann. Symp. Combin. Pattern Matching (CPM'04), Lecture Notes in Comp. Sci.* 3109, 338–399. Springer Verlag, Berlin, 2004.
5. A. Caprara. Sorting by reversals is difficult. In *Proc. 1st Int'l Conf. Comput. Mol. Biol. (RECOMB'97)*, 75–83, 1997.
6. W.H.E. Day and D. Sankoff. The computational complexity of inferring phylogenies from chromosome inversion data. *J. Theor. Biol.*, 127:213–218, 1987.
7. G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette. *Combinatorics of Genome Rearrangements.* MIT Press, 2009.
8. S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proc. 27th Ann. ACM Symp. Theory of Comput. (STOC'95)*, 178–189. ACM Press, New York, 1995.
9. H. Kaplan and E. Verbin. Efficient data structures and a new randomized approach for sorting signed permutations by reversals. In *Proc. 14th Ann. Symp. Combin. Pattern Matching (CPM'03), Lecture Notes in Comp. Sci.* 2676, 170–185. Springer Verlag, Berlin, 2003.

10. B.M.E. Moret and H.D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Monographs* **15**, 99–117, 1994.
11. B.M.E. Moret and T. Warnow. Advances in phylogeny reconstruction from gene order and content data. In E.A. Zimmer and E.H. Roalson, eds., *Molecular Evolution: Producing the Biochemical Data, Part B, Methods in Enzymology* 395, 673–700. Elsevier, 2005.
12. J.D. Palmer. Chloroplast and mitochondrial genome evolution in land plants. In R. Herrmann, ed., *Cell Organelles*, pp. 99–133. Springer Verlag, 1992.
13. J.D. Palmer and W.F. Thompson. Rearrangements in the chloroplast genomes of mung bean and pea. *Proc. Nat'l Acad. Sci., USA*, 78:5533–5537, 1981.
14. D. Sankoff. Edit distance for genome comparison based on non-local operations. In *Proc. 3rd Ann. Symp. Combin. Pattern Matching (CPM'92), Lecture Notes in Comp. Sci.* 644, 121–135. Springer Verlag, Berlin, 1992.
15. D. Sankoff and M. Goldstein. Probabilistic models for genome shuffling. *Bull. Math. Biol.*, 51:117–124, 1989.
16. D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
17. A.H. Sturtevant. A crossover reducer in Drosophila melanogaster due to inversion of a section of the third chromosome. *Biol. Zent. Bl.*, 46:697–702, 1926.
18. A.H. Sturtevant and Th. Dobzhansky. Inversions in the third chromosome of wild races of drosophila pseudoobscura and their use in the study of the history of the species. *Proc. Nat'l Acad. Sci., USA*, 22:448–450, 1936.
19. E. Tannier, A. Bergeron, and M.-F. Sagot. Advances on sorting by reversals. *Disc. Appl. Math.*, 155(6–7):881–888, 2007.
20. E. Tannier and M.-F. Sagot. Sorting by reversals in subquadratic time. In *Proc. 15th Ann. Symp. Combin. Pattern Matching (CPM'04), Lecture Notes in Comp. Sci.* 3109, 1–13. Springer Verlag, Berlin, 2004.

Figure legends

7.1 Figure 1 - The distribution of k for MAX-RECOVER on random permutations of length $10,000$.

7.2 Figure 2 - Normalized running time of RAND-RECOVER on random permutations of sizes from 100 to $50,000$—shown is the ratio to the running time of mergesort on arrays of matching size.

Tables

Table 1 - The failure rates for MAX, RAND and RAND+RESTART

Length	100	200	500	1,000	2,000	5,000	10,000	20,000
MAX	39.5%	38.9%	39.0%	39.1%	39.3%	39.3%	39.3%	39.2%
RAND	39.0%	39.2%	39.5%	39.5%	39.6%	39.5%	39.6%	39.5%
RAND-RESTART	17.2 %	17.1 %	16.8 %	16.4 %	16.3 %	16.2 %	16.0 %	16.0 %

Table 2 - Number of recovery steps (k) for MAX-RECOVER: Average and Standard Deviation

Length	100	200	500	1,000	2,000	5,000	10,000	20,000
AVE(k)	0.513	0.518	0.522	0.524	0.524	0.525	0.524	0.525
SD(k)	0.765	0.770	0.772	0.774	0.773	0.775	0.774	0.777

Table 3 - Number of recovery steps (k) for RAND-RECOVER: Average and Standard Deviation

Length	100	200	500	1,000	2,000	5,000	10,000	20,000
AVE(k)	0.485	0.489	0.492	0.493	0.495	0.495	0.495	0.499
SD(k)	0.690	0.694	0.697	0.697	0.698	0.698	0.698	0.699