Sorting Signed Permutations by Reversals, Revisited*

Haim Kaplan Elad Verbin[†]

September 14, 2004

Abstract

The problem of sorting signed permutations by reversals (SBR) is a fundamental problem in computational molecular biology. The goal is, given a signed permutation, to find a shortest sequence of reversals that transforms it into the positive identity permutation, where a reversal is the operation of taking a segment of the permutation, reversing it, and flipping the signs of its elements.

In this paper we describe a randomized algorithm for SBR. The algorithm tries to sort the permutation by repeatedly performing a random oriented reversal. This process is in fact a random walk on the graph where permutations are the nodes and an arc from π to π' corresponds to an oriented reversal that transforms π to π'. We show that if this random walk stops at the identity permutation, then we have found a shortest sequence. We give empirical evidence that this process indeed succeeds with high probability on a random permutation.

To implement our algorithm we describe a data structure to maintain a permutation, that allows to draw an oriented reversal uniformly at random, and perform it in sub-linear time. With this data structure we can implement the random walk in $O(n^{3/2}\sqrt{\log n})$ time, thus obtaining an algorithm for SBR that almost always runs in sub-quadratic time. The data structures we present may also be of independent interest for developing other algorithms for SBR, and for other problems.

Finally, we present the first efficient parallel algorithm for SBR. We obtain this result by developing a fast implementation of the recent algorithm of Bergeron [5] for sorting signed permutations by reversals that is parallelizable. Our implementation runs in $O(n^2 \log n)$ time on a regular RAM, and in $O(n \log n)$ time on a PRAM using n processors.

1 Introduction

A permutation over $1, \ldots, n$, where each element has a sign, + or -, is called a *signed permutation*. The problem of *sorting signed permutations by reversals (SBR)* is defined as follows. Given a signed permutation, find a shortest sequence of reversals that transforms it to the positive identity permutation, $id = (+1, \ldots, +n)$, where a reversal is the operation of taking a segment of the permutation, reversing it, and flipping the signs of its elements. The reversal $\rho(i, j)$ transforms

*A preliminary version of this paper has appeared as [18].

[†]School of Computer Science, Tel Aviv University, Tel Aviv, Israel. email: {haimk, eladv}@cs.tau.ac.il. This research was partially supported by the German-Israeli Foundation for Scientific Research and Development (GIF) under grant No. 2051-1156/2002

$\pi = (\pi_1, \ldots, \pi_n)$ into $\pi' = (\pi_1, \ldots, \pi_{i-1}, -\pi_j, -\pi_{j-1}, \ldots, -\pi_i, \pi_{j+1}, \ldots, \pi_n)$. The length of such a shortest sequence of reversals is denoted $d(\pi)$, and called the *reversal distance* of π from id.

This problem of sorting signed permutations by reversals is of great interest in computational biology, because it allows one to find a possible evolutionary path that transformed one species into another. For more details on this application see the book by Pevzner [22].

The unsigned variant of the problem, in which we deal with unsigned permutations, is NP-hard [7]. Somewhat surprisingly, Hannenhalli and Pevzner [14], showed in 1995 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, Hannenhalli and Pevzner described an algorithm that sorts signed permutations by reversals in $O(n^4)$ time. Kaplan, Shamir, and Tarjan [17] simplified the underlying combinatorial structure and described an algorithm that finds a shortest sorting sequence in $O(n^2)$ time. Bergeron [5] in 2001 simplified the combinatorial structure even further and described a somewhat different algorithm that runs in $O(n^3)$ time (which can be reduced to $O(n^2)$ on a bit-vector machine). 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 was still the asymptotically fastest when a preliminary version of this paper appeared [18]. Ozery-Flato and Shamir [21] showed that both Bergeron's algorithm and the algorithm of Kaplan, Shamir, and Tarjan have permutations on which they spend $\Omega(n^2)$ time.

Recently, Tannier and Sagot [28] managed to beat the $O(n^2)$ bound and present an algorithm for SBR that runs in $O(n^{3/2}\sqrt{\log n})$-time. This new algorithm use the data structure we develop here, together with additional new ideas. We further note that one can compute $\mathrm{d}(\pi)$ without a sequence that realizes it in linear time [2].

Meidanis, Walter and Dias [20] consider the problem of sorting signed circular permutations. They show that the reversal distance of a signed circular permutation π of n elements is equal to the reversal distance of a corresponding signed linear permutation π' of $n-1$ elements. Furthermore, they show that π' can be derived from π in linear time. This implies that all algorithms thus far, including the ones we present here, can be used to sort circular permutations within the same time bounds. We have not run tests on circular permutations, but we expect that the performance would be similar.

Our first result in this paper is a randomized algorithm for sorting by reversals. This algorithm repeatedly executes a random-walk process, which we call *SBR-RandWalk*. Each run of SBR-RandWalk either finds a minimum sorting sequence or fails. Empirical tests on SBR-RandWalk indicate that the average over all permutations of the expected number of times we need to run SBR-RandWalk to get a minimum sorting sequence is a small constant (namely, 1.6). We consider these empirical results to be quite strong, in the sense that they indicate that the success probability is independent of the size of the permutation. However, we could not find any mathematical proof of this observed behavior and analyzing SBR-RandWalk theoretically is an open problem.

We also describe a data structure for representing a permutation so that we can draw a random oriented reversal and apply it in $O(\sqrt{n \log n})$ time, thus obtaining an implementation of SBR-RandWalk that runs in time $O(n^{3/2}\sqrt{\log n})$. Combining this with the empirical study we conjecture that the expected running time of our algorithm on a random permutation is $O(n^{3/2}\sqrt{\log n})$.

Since the publication of the preliminary version of this paper variants of our data structure have been proved useful to obtain other results. Hartman in the upcoming journal version of [15] used this data structure to improve the running time of a 1.5-approximation algorithm for Sorting by Transpositions. Hartman and Sharan [16] used it to efficiently implement a 1.5-approximation algorithm for Sorting by Transpositions and Transreversals. Also, the recent breakthrough result by Tannier and Sagot [28] that gives a new $O(n^{3/2}\sqrt{\log n})$-time algorithm for SBR uses a variant of our data structure.

Finally, we describe a fast serial implementation of the algorithm of Bergeron for solving SBR. We obtain this implementation by reducing the score calculations in each iteration of the algorithm to a well studied geometric problem called *2-set point dominance counting* (2SDC) in the plane. By exploiting known solutions to the 2SDC problem we obtain a simple implementation that runs in $O(n^2 \log n)$ time. Furthermore, by exploiting efficient parallel solution to the 2SDC problem we can parallelize our implementation to run in $O(n \log n)$ time in the CREW PRAM model, using $O(n)$ processors. This is the first efficient parallel solution for sorting signed permutations by reversals.

The structure of the rest of this paper is as follows. Section 2 gives some definitions. In Section 3 we describe SBR-RandWalk. Section 4 gives the results of our empirical tests on SBR-RandWalk. In Section 5 we describe our data structure. In Section 6 we describe our parallel algorithm for SBR. We give our conclusions in Section 7.

2 Preliminaries

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

Suppose we want to sort a signed permutation, π. Now, to sort the permutation we must begin by locating a reversal that transforms π into π', where $d(\pi') = d(\pi) - 1$ – such a reversal is called a *safe* reversal . The underlying combinatorial theory (discovered by [3] and refined in [14] and [17]) distinguishes a class of reversals called *oriented reversals*. An oriented reversal is a reversal that makes consecutive elements i and $i+1$ of the permutation adjacent and identically signed specifically, for some j we either get $\pi_j = i$ and $\pi_{j+1} = i+1$ or $\pi_j = -(i+1)$ and $\pi_{j+1} = -i$ as a result of performing the reversal. (We give a precise definition and an example, shortly.)

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 a pair is called an *oriented pair* if its elements are of opposite signs, and an *unoriented pair* otherwise. Conventionally, 0 is considered to be positive. 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 *oriented* if either $\pi_i + \pi_{j+1} = +1$ or $\pi_{i-1} + \pi_j = -1$. 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. For example, in the permutation $\pi = (0, -4, -1, 3, 2, 5)$ there are exactly four oriented pairs: $(0, -1), (-1, 2), (-4, 3), (-4, 5)$, and exactly three oriented reversals: $\rho(2, 3)$, which reverses the elements $4, -1$, and $\rho(3, 4)$ and $\rho(2, 5)$, which reverse $-1, 3$ and $-4, -1, 3, 2$, respectively. Note that we got $-1, 3$ both when we wanted to make 3 adjacent to -4 and when we wanted to make -1 adjacent to 2, that is – we got the same reversal from two distinct pairs. Let us present an important observation:

Proposition 1 *A permutation has an oriented reversal iff it has a negative element.*

The theory guarantees that:

Proposition 2 *If a permutation has an oriented reversal then it also has an oriented reversal which is safe.*

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. Our approach will be to choose a random oriented reversal, and "hope" it is safe – see Section 3.

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* (which we will not define here, see [17]), that this process *always* fails to sort. For permutations that contain unoriented components *all* sequences of safe oriented reversals terminate with a permutation that has no oriented reversals (i.e. a positive permutation), 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, according to the theory, an oriented reversal is safe iff by applying it we do not create new unoriented components.

The conclusion from the background given in the last paragraph is the following observation, which is central to our algorithm:

Observation 3 *If a sequence of oriented reversals sorts a permutation π, then it is a solution for SBR on π.*

Proof: If a sequence of oriented reversals sorts π, then π contains no unoriented components. Suppose, in contradiction, that one of the reversals in the sequence, was unsafe. Then it must create an unoriented component which cannot be later removed by oriented reversals, and thus the sequence cannot be a sorting sequence (i.e. one that ends with the identity permutation). Therefore all reversals in the sequence are safe, and thus this is a solution for SBR on the permutation. $\square$

3 Exploiting Randomization to Sort by Reversals

The algorithm we define in this section is motivated by the fact 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. Exact calculations done by Bergeron et al. in [6, theorem 4] show that for a random permutation over n elements, the probability that a randomly-chosen oriented reversal is unsafe is $O(1/n^2)$.

We next describe our algorithm. Given a signed permutation, π, we first clear the unoriented components, if there are any, using the method of [17]. To sort the "cleared" permutation we iterate the following "random walk"-like process, called SBR-RandWalk. This random walk repeatedly picks a random oriented reversal and applies it without bothering to check if it is safe or not. SBR-RandWalk repeats this process until it gets a permutation with no oriented reversals, i.e. a positive permutation. The theory above, and Observation 3 in particular, implies that this walk generates a shortest sorting sequence iff it ends with the identity permutation. If SBR-RandWalk fails, i.e. if we ended with a positive permutation that is not the identity permutation, we run SBR-RandWalk again on the original permutation. If SBR-RandWalk fails $f(n)$ times, where $f(n)$ is some slowly growing function of n, we resort to running one of the known algorithms (e.g. [17] or [5]) to sort the permutation. The function $f(n)$ can be any function that tends to ∞ as $n \to \infty$, and only affects the worst-case running time of the algorithm. For concreteness we use $f(n) = \log n$.

Alternatively we can run a variant of this 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 the same expected time-complexity (up to a constant, of course), 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.

It is easy to see that after at most $n+1$ reversals SBR-RandWalk reaches a positive permutation. However, a naive implementation of SBR-RandWalk would spend linear time per reversal. This implementation traverses the permutation, finds all oriented reversals, and picks one at random. This would give us a total running time of $O(n^2)$.

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 [1].

Fortunately, we found a data structure that maintains a signed permutation under reversals and allows us to draw a random oriented reversal and perform it all in sub-linear time. In section 5.2

[1] One such example is a permutation of Ozery-Flato and Shamir that will be discussed shortly.

we describe this data structure, which allows us to draw a random oriented reversal and perform it in time $O(n^{1/2}\sqrt{\log n})$. Using this representation SBR-RandWalk can be implemented to run in $O(n^{3/2}\sqrt{\log n})$ time in the worst case.

Once we see that a single run of SBR-RandWalk can be implemented to run in sub-quadratic time, the next question to ask is whether our suggested algorithm that repeats calling SBR-RandWalk up to $f(n)$ 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 SBR-RandWalk fails a super-polynomial (in n) number of times. One such example is the permutation $\pi = (2, 4, 6, \ldots, 2k, -1, -3, -5, \ldots, -(2k-1))$ of length $n = 2k$ of Ozery-Flato and Shamir [21]. The reversal distance of this permutation is n and 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 run of SBR-RandWalk are extremely low, namely $\frac{1}{2^k}$.[2] The probability that we succeed in one out of $f(n)$ trials is $O(\frac{f(n)}{2^k}) = O(\frac{\log(2k)}{2^k})$ 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, see [21]). To conclude, the expected worst-case complexity of our algorithm is the same as the complexity of the fallback algorithm we choose to run.

However, it could be that the permutations for which SBR-RandWalk fails $f(n)$ 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.

4 Empirical Results and Conjectures about SBR-RandWalk

As previously stated, we expect that iterating SBR-RandWalk would terminate fast for 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 our algorithm on random permutations. Our results are summarized in Table 1 and in Figure 1. For a random permutation without unoriented components of size ranging from 10 to 10000 we estimated the probability that a single run of SBR-RandWalk succeeds. We also estimated the average number of runs of SBR-RandWalk that our algorithm does until it gets a sorting sequence. Specifically, we drew many permutations while throwing away those that had unoriented components (such permutations are rare and have appeared only for smaller values of n). Then we ran SBR-RandWalk repeatedly until success for each of the permutations. Column 3 in Table 1 lists the percentage of permutations that were sorted in the first run of SBR-RandWalk, and column 4 lists the average over the selected permutations of the number of runs of SBR-RandWalk until finding a sorting sequence.

Figure 1 shows the distribution of the number of runs until success for 100000 permutations of size 500. Results for other values of n are similar and exhibit the same decay. Note that Fig. 1

[2] Surprisingly, this is not much better than an easily proved lower bound for the probability of success of SBR-RandWalk 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.

n	number of permutations tested	percentage of success at first try	average number of runs until success
10	300000000	66.20%	1.642
20	100000000	63.86%	1.607
50	30000000	63.01%	1.594
100	10000000	62.82%	1.594
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 SBR-RandWalk on a random permutation.

shows that the number of runs until success decays exponentially. This hints that an overwhelming majority of the permutations are successfully sorted by SBR-RandWalk.

The conclusions from our experiments are as follows:

- If we select a random permutation that has no unoriented component uniformly of all such permutations of size n and run Procedure SBR-RandWalk on it once then it has about 0.63 chance of success, regardless of n. Also, for a specific permutation the chance of success of SBR-RandWalk is well-behaved, and is for almost all permutations between 0.55 and 0.75, also regardless of n (data not shown).

- If we select a permutation as above and run SBR-RandWalk repeatedly until we get a sorting sequence we will need roughly 1.6 runs on average, also regardless of n.

When running a variant of SBR-RandWalk 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 no adjacencies (which are, in some sense, the only permutations of "true" size n), both when choosing random oriented reversals and when choosing random oriented pairs.

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

Conjecture 4 *The average number of runs of SBR-RandWalk needed to find a sorting sequence of a uniformly-chosen signed permutation with no unoriented components of size n is $O(1)$.*

5 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} (that is, the inverse array, which for each element gives its location in π, disregarding its current sign). With this representation we can locate elements of the

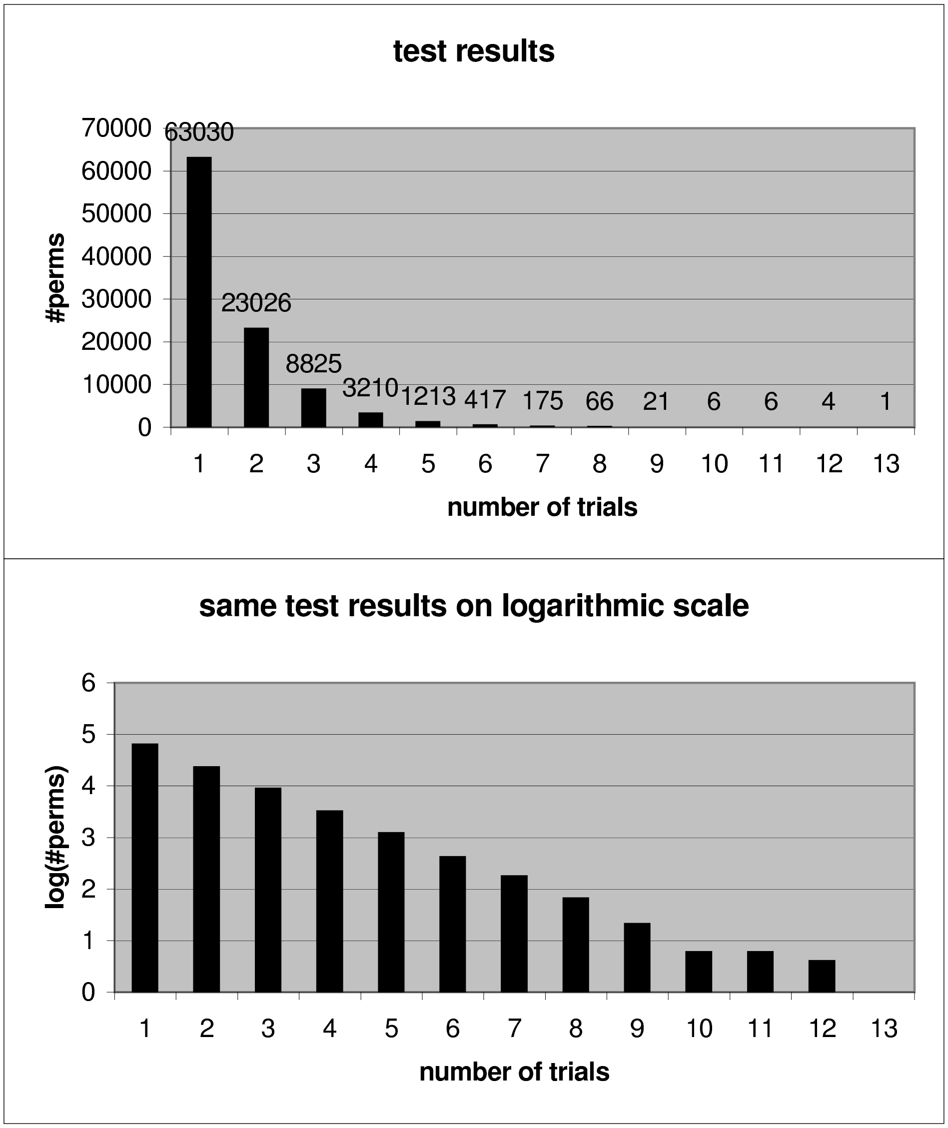

Figure 1: Distribution over 100,000 permutations of length 500 of the number of runs of SBR-RandWalk until finding a sorting sequence. The maximum number of runs was 13. The first diagram gives the number of permutations which were sorted after k runs for $k = 1, \ldots, 13$. For example, for 1213 permutations, which is about 1.2 percent of the sample, only the fifth run of SBR-RandWalk found a sorting sequence. The second diagram shows the same data in a logarithmic scale, so we could clearly see that the decay is roughly exponential. Similar distributions of other values of n look similar to these.

permutation in $O(1)$ time, but it takes time proportional to $j-i$ to perform the reversal $\rho(i,j)$, i.e. to update the arrays so that they represent the permutation after performing the reversal. So in the worst case, when $j-i$ is large, it takes $\Theta(n)$ time to perform the reversal. 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. For a discussion of the drawbacks of current algorithms that solve SBR, see [21].

Alternatively we could represent π and π^{-1} as doubly linked lists. With this representation we can perform a reversal in $O(1)$ time (given access to its endpoints), but it is harder to locate a particular element 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 SBR-RandWalk to run in sub-quadratic time with either of these two obvious representations. In this section we present alternative representations that do allow faster implementation of SBR-RandWalk. Any such representation should support two basic operations: 1) Draw a random oriented reversal (or say that none exists), 2) perform a reversal ρ on π, both in sub-linear time.

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.

5.1 Maintaining the Permutation

The two simple representations we describe here are similar to representations of permutations that were used by Chrobak et al. [8] and further investigated by Fredman et al. [13]. In both cases they were used to efficiently implement a common local improvement heuristics for the travelling salesman problem (looking at a TSP tour as a permutation over the vertex set of the graph). These data structures are quite practical.

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. The 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 structure that allows to draw a random oriented reversal borrows ideas from both of these two simple structures.

5.1.1 Tree-Based Data Structure

A data structure as described here can be based on any balanced search tree data structure that supports split and concatenate operations, such as splay trees [25], red-black trees, 2-4 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. For similar reasons our description use splay trees as well. Since splay trees have only logarithmic amortized time bounds, using them, the running time of our algorithm would be amortized as well. We can obtain worst-case bounds using a different kind of search trees such as red-black trees.

In our representation we hold a tree with n nodes containing the elements of the permutation, such that an in-order traversal of the tree gives us the permutation. We complicate matters by introducing a *reverse flag* for each node, which, if turned on, indicates that the subtree rooted at that node is read in reverse order, that is – from right to left, and the signs of its elements in it are flipped. Further reverse 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 reverse flags, always gives us the permutation.

Note that we can clear the reverse flag of an internal node by exchanging its children, flipping the reverse flag in each of these children, and flipping the sign of the element residing at the node. This procedure does not affect the permutation represented by the tree. Similarly, we can clear the reverse flag of a leaf by flipping the sign of the element contained in that leaf. One can view this procedure as "pushing down" the reverse flags. By clearing reverse flags we can implement a rotation in a splay tree so that it does not change the permutation represented by the tree. Clearly a rotation of an edge both whose endpoints have reverse flags turned off does not change the permutation. We perform a rotation of an edge with reverse flag turned on, in at least one of its endpoint, by first clearing the reverse flags from the endpoints of the edge being rotated. Once we can implement a rotation in $O(1)$ time then we can perform all standard splay tree operations including split and concatenate in $O(\log n)$ amortized time.

To implement queries on the permutation we also maintain in each node the size of its subtree, and update these subtree sizes when we do a rotation.

An example of the data structure is shown in Fig. 2.

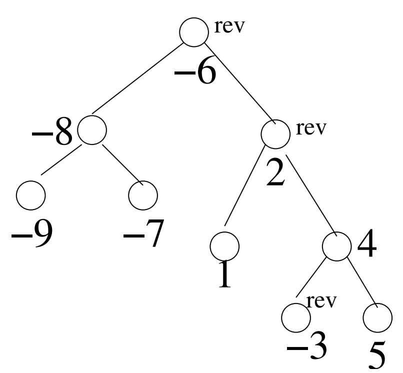

Figure 2: A representation of the permutation $\pi = (1, 2, 3, 4, 5, 6, 7, 8, 9)$. A "rev" beside a node means that the reverse flag of the node is on. Subtree sizes are not shown.

To find π_i we search for the i^{th} node in the (altered) in-order traversal using subtree sizes to direct the search. Since we use splay trees, after the search the i^{th} node is the root of the tree. The element π_i is the element stored at the root if the reverse flag of the root is off, or minus of that element, if the reverse flag of the root is on.

Consider now an inverse query. That is, given i we want to find π_i^{-1} – the index of $\pm i$ in π. Suppose first that we are given a pointer to the node x containing π_i in the splay tree representing the permutation. Then we can find π_i^{-1} by splaying the node x. After the splay x is the root. If the reverse flag of x is "off" then π_i^{-1} is one larger than the subtree size of the left child of x, and if the reverse flag of x is "on" then π_i^{-1} is one larger than the subtree size of the right child of x. We get the pointer to node x by keeping an additional array of pointers mapping an element i to the node x containing $\pm i$. This array is static, as we keep each element inside the same node at all times. Clearly the cost of the splay dominates the running time of the inverse query, and therefore inverse query takes $O(\log n)$ amortized time.

To execute a reversal $\rho(i, j)$ we split the tree at the node corresponding to π_j to a tree T_1 containing all items with indices at most j, and a tree T_2 containing all items with indices larger than j. Then we split T_1 at π_i to a tree T_3 containing all items with indices smaller than i and to a tree T_4 containing all items with indices at least i. Finally we flip the reverse flag of the root of T_4, concatenate T_3 to T_4 and concatenate the resulting tree to T_2. It is easy to see that the resulting tree represents the permutation after we performed the reversal. Since splay trees support split and concatenate in $O(\log n)$ amortized time, we perform a reversal in $O(\log n)$ amortized time. Summing up, we get the following theorem.

Theorem 5 *There exists a data structure that maintains a (signed) permutation of n elements, such that we can find π_i, π_i^{-1}, and perform a reversal $\rho(i, j)$, in logarithmic amortized time. The data structure is of size linear in n, and, given a permutation, one can initialize it in linear time.* $\square$

5.1.2 Block-Based Data Structure

The second data structure is in fact a two-level version of the previous structure. The structure is based on partitioning the permutation into blocks, each a contiguous fragment of the permutation of size $\Theta(\sqrt{n})$. We associate with each block a reverse flag, which, if *on*, indicates that the block should be read in reversed direction and the elements in it have opposite signs. This way, we can reverse a block by flipping its reverse flag. We maintain the blocks in a list. With each block we record its size, its reverse flag, and a pointer to the list of the items it contains. Each item points to the block containing it. See Figure 3. We also maintain the following invariant.

Invariant 6 *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 $\Theta(\sqrt{n})$.

5.1.3 Performing a Reversal

We perform a reversal $\rho(i, j)$ by the following three steps.

1. We locate i and j, and the blocks $b(i)$ and $b(j)$ containing i and j respectively. If $b(i)$ contains elements that do not belong to the reversal then we split $b(i)$ into two blocks: one that contains only elements that belong to the reversal, and the other contains only elements that do not belong to the reversal. Similarly we split $b(j)$ if it contains elements that do not belong to the reversal. Note that the new blocks may be small and therefore we may temporarily violate Invariant 6. We reinsert the new blocks to the block-list so that the blocks that contain the elements belonging to the reversal are consecutively placed, i.e. so that the data structure still represents the same permutation.

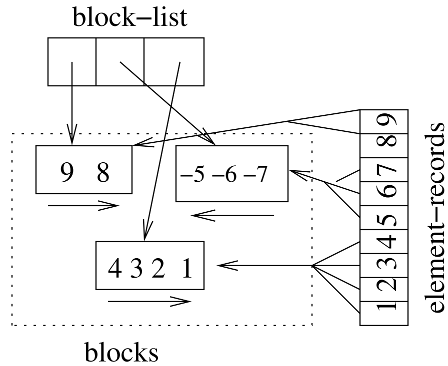

Figure 3: A representation of the permutation $\pi = (9, 8, 7, 6, 5, 4, 3, 2, 1)$. Note that we allow blocks' sizes to be between 2 and 6. The arrows represent the reverse flags.

2. Now the reversal consists of a subsequence of complete blocks. We reverse the order of these blocks in the block-list, and flip their reverse flags. It is easy to see that we now indeed represent the permutation after the reversal.

3. Finally, we reinstate Invariant 6 as follows. If one or more of the blocks that we have created in step (1) is of size smaller than $\frac{\sqrt{n}}{2}$, then we pick one such block and merge it with one of its neighbors. We repeat this step until no blocks are of size smaller than $\frac{\sqrt{n}}{2}$.

   Now some blocks may be larger than $2\sqrt{n}$ (but all are still smaller than $3\sqrt{n}$). We split each such large blocks to two small ones that satisfy the Invariant 6.

It is easy to see that using our representation we can perform these three steps in $O(\sqrt{n})$ time. Figure 4 shows an example of this process.

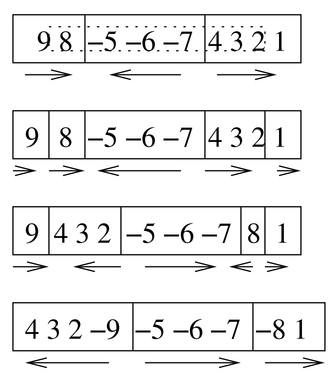

Figure 4: Performing the reversal $\rho(2, 8)$ on $\pi = (9, 8, 7, 6, 5, 4, 3, 2, 1)$ results in $\pi' = (9, -2, -3, -4, -5, -6, -7, -8, 1)$. Note that we allow blocks' sizes to be between 2 and 6. The arrows represent the reverse flags.

5.1.4 Improving Query Times

It is easy to see how we can use this data structure to answer queries and inverse queries in $O(\sqrt{n})$ time. However we can speed up queries by maintaining the list of blocks in a balanced search tree (of any kind), and representing the items in each block in an array rather than a linked list. With each block in the search tree we maintain the total number of elements that reside in blocks in its subtree. Using this representation we can perform a query operation in logarithmic time: Given i we first locate the block containing π_i by a binary search using the tree representing the block list. Then we can directly access π_i in the array representing the block. It is also easy to see that we can update this representation in $O(\sqrt{n})$ when we perform a reversal.

We can implement an inverse query in $O(1)$ time by maintaining with each element its index in the array representing its block, and with each block the total number of items in blocks preceding it. To get the location π_i^{-1} of an element i we add the index of i within its block to the prefix count of the block if the reverse flag of the block is off. Otherwise, we first subtract the index of i from the size of the block to get the index of i as if the block has been reversed, and then add it to the prefix count of the block. We thus proved the following. As before, it is easy to check that we can update this additional structure during a reversal within the $O(\sqrt{n})$ time bound.

Theorem 7 *There exists a data structure that maintains a (signed) permutation of n elements, and allows to find π_i given i in logarithmic time, to find π_i^{-1} given i in constant time, and to perform a reversal $\rho(i,j)$ in time $O(\sqrt{n})$. The data structure is of size linear in n, and, can be initialized in linear time.* □

5.2 Adding the Ability to Draw a Random Oriented Pair

In this section we extend the data structures from Section 5.1 so that we can draw an oriented pair uniformly at random. Recall that in Section 2 we showed how one can use a procedure that draws an oriented pair uniformly at random to draw an oriented reversal uniformly at random with only an expected constant time slowdown and vice versa. We draw pairs since they are easier to work with: We can associate a single pair with each specific element of the permutation.

We present a data structure to maintain a signed permutation, that supports queries for π_i in logarithmic time, queries for π_i^{-1} in $O(1)$ time, performing a reversal in $O(\sqrt{n \log n})$ time, and drawing an oriented pair uniformly at random in logarithmic time. The data structure also allows to find the number of oriented pairs in π in constant time, and requires linear space.

Note that for the purposes of implementing SBR-RandWalk it suffices that we can draw a reversal and carry it out in $O(\sqrt{n \log n})$ time. We can achieve that without the additional structure presented in Sections 5.2.2 and 5.2.3 for speeding up queries.

5.2.1 The Structure

We build upon the block-based structure of Section 5.1.2, but add to each block a tree that stores the pairs related to that block, and their orientations. We call each such tree a *pairs tree*. See Figure 5. Each pairs tree is similar to a single instance of the data structure described in Section 5.1.1, i.e. it is a balanced tree such as a splay tree.

The pairs tree of a block contains, for each element $\pm x$ in the block, the pair whose elements are $\pm x$ and $\pm(x+1)$.[3] Since the element $\pm x$ belongs to the block, we refer to it as the *local end* of the pair, and we refer to the element $\pm(x+1)$ as the *remote end* of the pair. The order of the pairs in the tree is according to the order of their **remote ends** along the permutation. (That is, according to increasing order of $\pi^{-1}(\pm(x+1))$). Thus given an index j we can search for a pair whose remote end immediately preceding j in logarithmic time.

With each pair in a pairs tree we also store its orientation. Furthermore, each node in a pairs tree has a *reverse flag*. If the reverse flag of a node v is *on* it means that the subtree T_v rooted at v should be read in reverse order, and the orientations of the pairs in T_v is opposite to what is indicated by their orientation bits. As before, further reverse flags down the tree can again alter the orientations and the order of the pairs. Thus, more precisely, our invariant is that an in-order read that is appropriately modified by the reverse flags gives the remote ends of the pairs in increasing order along the permutation. In order to draw an oriented reversal uniformly at random we also keep with each node v two counters. The first counts the number of oriented pairs in the subtree rooted by v and the second counts the number of unoriented pairs in the subtree rooted by v. It is easy to see that the reverse flags and these counters can be maintained while doing a rotation. Therefore we can perform all standard search tree operations on a pairs tree in logarithmic time. Note that in particular we maintain the total number of oriented pairs within each block.

As in Section 5.1.2 each block has a reverse flag. This flag, if on, indicates as before, that the order of the elements in the block is reversed and their signs are flipped. Furthermore, here it also indicates that the orientations (and not the order) of the pairs associated with all of the block's elements (that is, the orientations of the pairs in the block's tree) are once again opposite to what is indicated by the tree corresponding to the block. Furthermore, in this case the counters in all the nodes of the pairs tree are in fact flipped. The counter of the oriented pairs stores the number of unoriented pairs in the corresponding subtree and vice versa.[4] For an example, see Figure 5 and Figure 6.

5.2.2 Query

Searching for π_i and π_i^{-1} in $O(\sqrt{n})$ time is straightforward. As in Section 5.1.4. using an additional search tree over the blocks, and using arrays to represent the elements in each block we can reduce the time of the queries.

5.2.3 Draw an oriented reversal uniformly at random

The structure as described so far already allows to draw a random oriented reversal in $O(\sqrt{n})$ time. We do that by summing up the number of oriented pairs in all blocks. Then we draw an index of one of the pairs uniformly at random and locate the pair, by first locating the block containing it, and then finding the pair itself by searching the pair tree of the block itself.

We can speed up this process if we maintain a search tree over the blocks as indicated in Section 5.2.2. With each block we also keep the total number of oriented pairs in the blocks of its subtree.

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

[4]The reason for this additional interpretation of the block's flag will be clear when we describe how to perform a reversal.

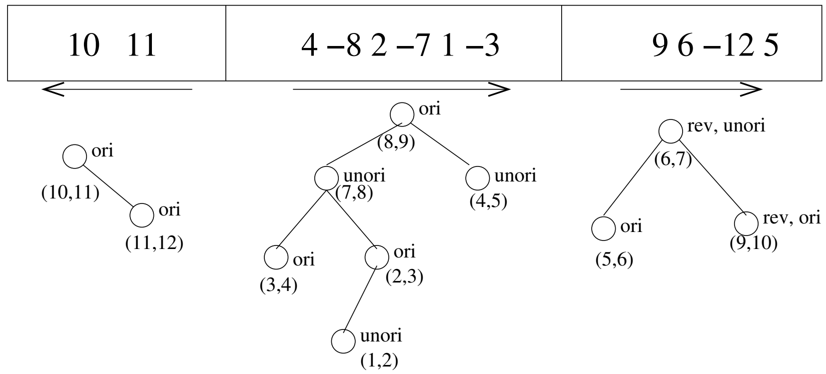

Figure 5: A representation of the permutation $\pi = (-11, -10, 4, -8, 2, -7, 1, -3, 9, 6, -12, 5)$. The arrows under the blocks represent the reverse flags of the blocks. A "rev" beside a node means that the reverse flag of the node is on. Note that both nodes in the first block are labelled oriented although the pairs are unoriented. This is because we define the reversed flag of the block to also reverse the orientations of all pairs in the block.

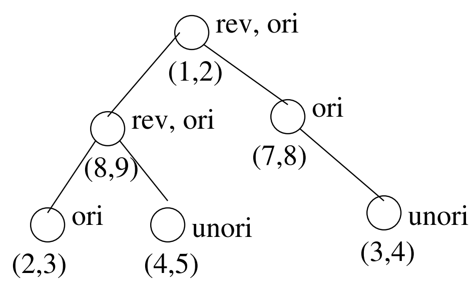

Figure 6: Another representation of the pairs tree of the middle block.

In particular we have the total number of oriented pairs in the permutation at the root. Using this tree once we draw a random index of an oriented pair we can locate the block containing the pair, and the pair itself in logarithmic time.

5.2.4 Performing a Reversal

We perform a reversal $\rho(i,j)$ using similar steps as in Section 5.1.3. When blocks are split and concatenated we rebuild the pairs trees associated with them. This takes time linear in the size of the trees involved, that is $\Theta(\sqrt{n})$.

In addition we have to update the pairs trees (both those associated with blocks that are in the segment defining the reversal and those associated with blocks that are out of the reversal) so that the orientations and the order of pairs inside these trees will correspond to the permutation after the reversal (We may assume now that the reversal spans complete blocks, without fractions of blocks).

To complete performing $\rho(i,j)$ we go over all blocks. For each block we split its pairs tree into at most three trees, much like we did in Theorem 5. The middle tree contains all pairs whose remote end is part of the reversal, and the other two trees will contain the pairs whose remote end is not in the reversal. We then flip the reverse flag of the root of the middle tree, switch the values of the counters of the oriented and unoriented reversals, and concatenate the trees in their original order.

It is easy to see that the order of the pairs in all trees is consistent with the new permutation. Consider the orientations of the pairs and the counters of oriented and unoriented pairs at the nodes. For blocks outside of the reversal these indeed correspond to the new permutation. However for blocks inside the reversal the orientations of the pairs are exactly opposite of their correct setting, and similarly the counters of oriented and unoriented pairs at the nodes are flipped. This is because in these blocks we flipped the orientation of those pairs with both ends in the reversal and haven't changed the orientation of the others. Recall however that we expanded the role of the blocks' reverse flags to indicate that the orientations of the block's pairs are flipped. Therefore since we flip the reversal bit of the blocks in the reversal we indeed have a correct representation of the new permutation.

In case we maintain an additional tree over the blocks to speed up queries, and the draw of a random oriented reversal, we have to update this tree when we perform a reversal. This can be done easily within the time bound as the size of this tree is $O(\sqrt{n})$

5.2.5 A small adjustment

We use block size of $\Theta(\sqrt{n})$ in the structure of Section 5.1.2 to balance the cost of splitting and concatenating the blocks containing the ends of the reversal with the cost of rearranging the blocks themselves. In the structure we present here if we use blocks of size $\Theta(\sqrt{n})$ then the cost of performing a reversal is $\Theta(\sqrt{n}\log n)$ since we have to perform a constant number of operations on each of the $\Theta(\sqrt{n})$ pairs tree. However the cost of splitting and concatenating a block is still $O(\sqrt{n})$. This indicates that a block size of $\Theta(\sqrt{n})$ is not optimal here and we would reduce the time bound if we choose a slightly larger block size so that the total number of blocks decreases.

To get the best trade-off in this data structure we maintain the sizes of the blocks between $\frac{\sqrt{n \log n}}{2}$ and $2 \cdot \sqrt{n \log n}$. This makes the entire procedure of performing an oriented reversal run in $O(\sqrt{n \log n})$ time. It takes $O(\sqrt{n \log n})$ time to perform the operations on the part of the data structure that we inherited from Theorem 5.2, and additional $O(\sqrt{n \log n})$ time to update $\Theta(\sqrt{\frac{n}{\log n}})$ trees as outlined above, in logarithmic time per tree.

Thus we have proved the following:

Theorem 8 *There exists a data structure that maintains a (signed) permutation of n elements, supporting a query for π_i in logarithmic time, a query for π_i^{-1} in constant time, and allows to perform a reversal $\rho(i,j)$ in time $O(\sqrt{n \log n})$. This data structure also allows to report the number of oriented pairs in $O(1)$ time, and to draw an oriented pair uniformly at random in $O(\log n)$ time. The data structure is of size linear in n, and, given a permutation, we can initialize it in linear time.* □

6 An Efficient Parallel Algorithm for SBR

In this section we present a new implementation of the algorithm of Bergeron [5] for the problem of sorting signed permutations by reversals. Our implementation is simple, and in contrast with previous algorithms it is parallelizable. The algorithm runs in $O(n^2 \log n)$ time, and its parallelization (on a *CREW PRAM*) runs in $O(n \log n)$ time using n processors (i.e. performs $O(n^2 \log n)$ work). We can easily adapt our algorithm to various other parallel models, such as the *reconfigurable mesh*, with similar time bounds.

A PRAM (Parallel Random Access Machine) is a theoretical model for a multi-processor computer. In this model the number of processors is a resource that may grow with the input size, and processors communicate via shared memory. A CREW (Concurrent Read, Exclusive Write) PRAM, is a special case where a memory cell can be read by several processors at the same time, but only one processor can write into a memory cell at the same time. See [12] for more information on PRAMs.

Let π be a permutation and ρ an oriented reversal of π. Bergeron defines the *score* of ρ as the number of oriented pairs in $\rho \cdot \pi$, the permutation that results from applying ρ on π. We also define the *delta-score* of ρ as the difference between the number of oriented pairs in $\rho \cdot \pi$ and the number of oriented pairs in π. A positive delta-score means that the amount of oriented pairs increases when the reversal is applied. We sometimes refer to the score of an oriented pair. By this we mean the score of the reversal that is derived from the pair.

Bergeron proved the following Theorem:

Theorem 9 ([5]) *Given a permutation π, each oriented reversal of π with maximum score is a safe reversal.*

Based on Theorem [5] Bergeron suggested the following algorithm to find a shortest sorting sequence of a signed permutation with no unoriented components. (Recall that unoriented components can be cleared from π as e.g. in [17]. We also assume, as before, that π is extended with 0 to the left and $n+1$ to the right).

While ($\pi \neq id$)

Choose an oriented pair with maximum score.

Perform the reversal that corresponds to that pair.

Bergeron gave an $O(n^2)$ time implementation of her method on a bit-vector machine[5], but on a RAM the only implementation suggested was the trivial one, which takes $O(n^3)$ time. In this straightforward implementation there are $O(n)$ steps – one for finding each safe reversal; in each step we calculate scores of $O(n)$ reversals, where each score calculation takes $O(n)$ time.

We will show how the delta-scores of all oriented reversals of a permutation can be calculated in $O(n \log n)$ time, resulting in a $O(n^2 \log n)$ serial implementation of Bergeron's Method. Then we show how to parallelize this implementation, so that calculating the delta-scores of all oriented reversals takes $O(\log n)$ time using n processors. This yields an $O(n \log n)$-time CREW-PRAM algorithm for SBR that runs on n processors.

First note that there is a natural correspondence between the pairs of a permutation and points of the grid $[1, n]^2$, whereby the pair (π_i, π_j) is identified with the point (i, j). Note that since, by our definition of a *pair*, $i < j$, then all points corresponding to the pairs lie above the line $i = j$.

A reversal and a pair are said to be *overlapping* when the reversal contains one element of the pair but not the other. It is easy to see that a reversal turns every oriented pair that overlaps with it to unoriented, and turns every unoriented pair that overlaps with it to oriented. We denote by $f_O(\rho)$ (resp. $f_U(\rho)$) the number of newly oriented (resp. unoriented) pairs created by applying ρ. The difference $f_O(\rho) - f_U(\rho)$ is the delta-score of ρ, and Theorem 9 implies that $argmax_\rho(f_O(\rho) - f_U(\rho))$ is safe. To calculate delta-scores we will find for each oriented reversal, how many oriented and unoriented pairs it overlaps with.

6.1 Reducing the score calculation to two-set dominance counting

We reduce the problem of finding, for each oriented reversal, how many oriented and unoriented pairs it overlaps to a well-known problem in computational geometry called 2-set point dominance counting in the plane (2SDC). The 2SDC problem is defined as follows. Let $\prec$ be the relation on $\mathbf{R}^2$ where $(x_1, y_1) \prec (x_2, y_2)$ iff $(x_2 < x_1) \wedge (y_1 < y_2)$[6]. Given two sets, $A, B \subset R^2$, of total size n, the 2SDC problem is to compute for each point $(x_1, y_1) \in A$, the number $Dom((x_1, y_1), B) \equiv |\{(x_2, y_2) \in B \text{ s.t. } (x_1, y_1) \prec (x_2, y_2)\}|$.

Let B_O be the set of points corresponding to oriented pairs, and let B_U be the set of points corresponding to unoriented pairs. It is easy to verify that for a reversal $\rho = \rho(i, j)$

$$f_U(\rho) = Dom((i, i), B_O) + Dom((j, j), B_O) - 2 \cdot Dom((i, j), B_O) \tag{1}$$

and

$$f_O(\rho) = Dom((i, i), B_U) + Dom((j, j), B_U) - 2 \cdot Dom((i, j), B_U) \ . \tag{2}$$

This is depicted in Fig. 7.

[5]A bit-vector machine is a RAM that can operate on bit-vectors of size $O(n)$ in unit time. In a sense, it is a parallel model too because a bit-vector machine actually models a computer with one processor and one memory-pool but many ALUs (an ALU is the processor component that performs the actual arithmetic operations).

[6]Note the non-standard, asymmetric, definition of $\prec$. This is done to ease the presentation later on.

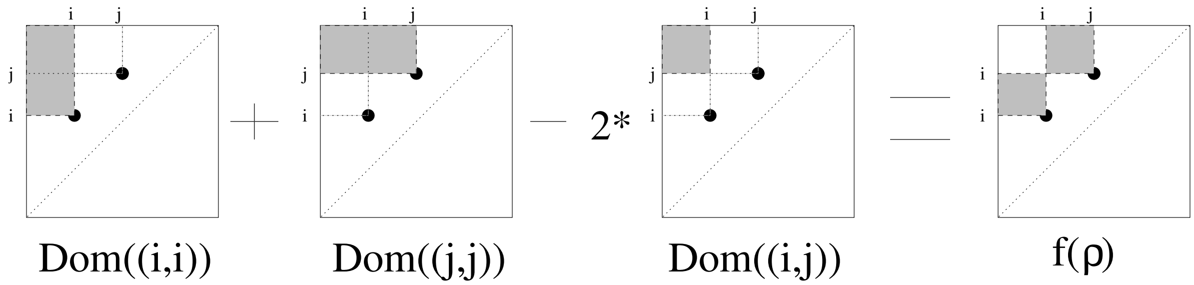

Figure 7: Graphical Representation of Equations 1,2

Let A be the set that contains, for each oriented reversal $\rho = \rho(i, j)$ of π the points (i, i), (j, j) and (i, j). It follows from Equations (1) and (2) that by solving the 2SDC problem once for A and $B = B_U$ and also for A and $B = B_O$, we can calculate $f_U(\rho)$ and $f_O(\rho)$, and delta-score$(\rho) = f_O(\rho) - f_U(\rho)$ for every $\rho \in A$ from the results of the 2SDC computation in linear time.

Putting it all together we suggest to implement the algorithm of Bergeron described above so that in each iteration we calculate scores for all oriented reversals by solving two instances of the 2SDC problem. Once having these scores we pick a reversal with maximum score as the next one in the sequence.

6.2 Solving Two-Set Dominance Counting

It is easy to solve 2SDC serially in $O(n \log n)$ time as follows. We sort $A \cup B$ by x-coordinate, resolving ties by putting points from A before those that belong to B. Then we traverse the points in increasing x order. During the traversal we maintain the points from B already traversed in a balanced search tree T ordered by their y coordinate. When we traverse a point from B we insert it into T, and when we traverse a point $p \in A$ we search T using the y coordinate of p. The answer to the query $Dom(p, B)$ is the sum of the sizes of the subtrees hanging to the right of the search path (holding points with larger y's).[7] Note that one can also solve this problem using standard geometric data structure such as priority search trees [19] or range trees [9]. Since the coordinates of the points in A and B are small integers we can use a data structure of Dietz [11] instead of the search tree T to improve the running time of the algorithm above to $O(n \cdot \frac{\log n}{\log \log n})$.

In the PRAM model the problem of two-set dominance counting is closely related to the problem of General Prefix Computation, which is a staple problem in the theory of parallel algorithms. Both of these problems were shown to take $O(\log n)$ time on a n-processor CREW PRAM in [1] via a general method for parallel divide-and-conquer. An alternative, randomized, solution appears in [24].

Going back to the algorithm of Bergeron for SBR, we observe that aside from score calculations, all other operations that are performed at each iteration can be implemented in linear time, or in $O(\log n)$ time on a CREW-PRAM. So we get:

[7]One can maintain subtree counter in a balanced search tree as in Section 5.1.

Theorem 10 *SBR can be solved in* $O(n^2 \cdot \frac{\log n}{\log \log n})$ *time on a RAM, and in* $O(n^2 \log n)$ *time on a pointer machine [29] (that is without using indirect access).*

Theorem 11 *SBR can be solved in* $O(n \log n)$ *time on a CREW-PRAM using* n *processors.*

Efficient running times can also be achieved in other parallel models for which there exist good algorithms for 2SDC. Springsteel and Stojmenović [26] show how to solve 2SDC (and also General Prefix Computation) optimally on several other parallel models, including the BSR (broadcasting with selective reduction) and the mesh-connected computer, and show an $O(\log^2 n)$ time algorithm for the hypercube. Jang [30] shows how to solve these problems in $O(n)$ time on the $n \times n$ *reconfigurable mesh*. For more results on various parallel models see [31, 27]. When running on a coarse-grained parallel computer (e.g. an Alpha machine with 4 processors, a small-scale Network of Workstations, etc.), there are special algorithms that give good running times. See, for example [10].

7 Conclusion

In Sections 2 through 5 we described a randomized algorithm for SBR that gives good results on random permutations. To implement this algorithm efficiently we described a data structure to maintain a permutation under reversals such that it is possible to draw a random oriented reversal, and to perform other queries in sub-linear time. This Data Structure has proved to be of independent interest.

In section 6 we described an efficient implementation of Bergeron's algorithm, and a method for parallelizing it, getting an efficient parallel algorithm for solving SBR.

One problem that we left open is proving Conjecture 4. This would: 1) support theoretically the good empirical performance of our algorithm, and 2) indicate that there are many minimum sorting sequences of reversals, so the biologist may want to add constraints in order to get a meaningful sequence. Research in this direction was initiated by Pinter et al. [4] and by Bergeron et al. [6].

Another such phenomenon that we have observed empirically is that there is a global parameter $0 < c < 1$, such that for every permutation, at least a c-fraction of oriented reversals is safe. From our experience, $c = 0.25$ seems to work. This kind of knowledge about the statistical behavior of SBR may prove useful to directions such as the ones pursued in [4, 6], or to new studies such as those that try to get a more realistic theoretical model to understand the way real-world genomes have mutated throughout history (e.g. Pevzner and Tesler [23]).

The new sub-quadratic algorithm of Tannier and Sagot uses a variant of our data structure that maintains some more information about the permutation, but it also uses a new method of generating the sorting sequence. This method builds the sorting sequence by considering an initial sequence of oriented reversals that doesn't necessarily sort the permutation, and repeatedly adding to the sequence (not necessarily at the end), until, finally, they get a minimum sorting sequence. However, it seems intriguing whether one can use the old methodology to solve SBR in sub-quadratic time, that is whether one can find a data structure where a safe oriented reversal can be found and executed in sub-linear time. A possible approach is to try to add to the data structures of Section 5.2 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 with backtracking would give an algorithm that takes $o(n^2)$ expected time for every permutation, provided we can prove that at least a constant fraction of the oriented reversals is always safe (see the discussion in the previous paragraph). Another (maybe harder) way to augment the data structure could be to find a way to maintain with the pairs whether the reversals they define are safe or not, such that reversals on this data structure can still be done in sub-linear time. Then you could solve SBR by performing only safe reversals until the permutation is sorted.

Variants on our data structure have already been used to speed up algorithms for SBR and similar problems, and we believe that there may be even more applications in the area of sequence analysis, and in other areas. It also seems plausible that our data structure can be improved, maybe even so that both performing a reversal and getting a random oriented reversal could be done in logarithmic time. This would immediately give us an improvement in the running time of SBR-RandomWalk. This is, in some sense, a geometric data structure (if you think of the pairs as points in N^2), so techniques from computational geometry (such as those for orthogonal range searching) may be useful.

Acknowledgments

We thank Michal Ozery-Flato, Tzvika Hartman, and Ron Shamir for many helpful discussions. We also thank both of the anonymous referees for their thoughtful comments.

References

[1] M. J. Atallah, R. Cole, and M. T. Goodrich, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, SIAM Journal on Computing **18** (1989), no. 3, 499–532.

[2] D.A. Bader, B. M.E. Moret, and M. Yan, *A linear-time algorithm for computing inversion distance between two signed permutations with an experimental study*, Journal of Computational Biology **8** (2001), no. 5, 483–491.

[3] V. Bafna and P. A. Pevzner, *Genome rearragements and sorting by reversals*, SIAM Journal on Computing **25** (1996), no. 2, 272–289.

[4] M. A. Bender, D. Ge, S. He, H. Hu, R. Y. Pinter, S. Skiena, and F. Swidan, *Improved bounds on sorting with length-weighted reversals*, Proceedings of SODA, 2004, pp. 912–921.

[5] A. Bergeron, *A very elementary presentation of the Hannenhalli-Pevzner theory*, proceedings of CPM, 2001, pp. 106–117.

[6] A. Bergeron, C. Chauve, T. Hartman, and K. Saint-Onge, *On the properties of sequences of reversals that sort a signed permutation*, proceedings of JOBIM, 2002, pp. 99–108.

[7] A. Caprara, *Sorting by reversals is difficult*, proceedings of RECOMB, 1997, pp. 75–83.

[8] M. Chrobak, T. Szymacha, and A. Krawczyk, *A data structure useful for finding hamiltonian cycles*, Theoretical Computer Science **71** (1990), no. 3, 419–424.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, MIT Press, Cambridge, MA, 1990.

[10] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin, *Scalable parallel geometric algorithms for coarse grained multicomputers*, Proceedings of the ninth annual symposium on Computational geometry, 1993, pp. 298–307.

[11] P.F. Dietz, *Optimal algorithms for list indexing and subset rank*, Proceedings of WADS, 1989, pp. 39–46.

[12] S. Fortune and J. Wyllie, *Parallelism in random acess machines*, Proceedings of STOC, 1978, pp. 114–118.

[13] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer, *Data structures for traveling salesmen*, Journal of Algorithms **18** (1995), no. 3, 432–479.

[14] S. Hannenhalli and P. A. Pevzner, *Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals*, Journal of the ACM **46** (1999), no. 1, 1–27.

[15] T. Hartman, *A simpler 1.5-approximation algorithm for sorting by transpositions*, proceedings of CPM, 2003, pp. 156–169.

[16] T. Hartman and R. Sharan, *A 1.5-approximation algorithm for sorting by transpositions and transreversals*, To appear in the Proceedings of WABI, 2004.

[17] H. Kaplan, R. Shamir, and R. E. Tarjan, *A faster and simpler algorithm for sorting signed permutations by reversals*, SIAM Journal on Computing **29** (1999), no. 3, 880–892.

[18] H. Kaplan and E. Verbin, *Efficient data structures and a new randomized approach for sorting signed permutations by reversals*, proceedings of CPM, 2003, pp. 170–185.

[19] E. M. McCreight, *Priority search trees*, SIAM Journal on Computing **14** (1985), 257–276.

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

[21] M. Ozery-Flato and R. Shamir, *Two notes on genome rearragnements*, Journal of Bioinformatics and Computational Biology **1** (2003), no. 1, 71–94.

[22] P. A. Pevzner, *Computational molecular biology: An algorithmic approach*, MIT Press, Cambridge, MA, 2000.

[23] Pavel Pevzner and Glenn Tesler, *Human and mouse genomic sequences reveal extensive breakpoint reuse in mammalian evolution*, Proceedings of the National Academy of Sciences U.S.A. **100** (2003), no. 13, 7672–7677.

[24] J. H. Reif and S. Sen, *Optimal randomized parallel algorithms for computational geometry*, Proceedings of the International Conference on Parallel Processing (1987), 270–277.

[25] D.D. Sleator and R.E. Tarjan, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach. **32** (1985), 652–686.

[26] Frederick Springsteel and Ivan Stojmenović, *Parallel general prefix computations with geometric, algebraic, and other applications*, International Journal of Parallel Programming **18** (1989), no. 6, 485–503.

[27] I. Stoica, *Time-optimal algorithms for generalized dominance computation and related problems on mesh connected computers and meshes with multiple broadcasting*, Parallel Algorithms and Applications **8** (1996), 271–283.

[28] E. Tannier and M.-F. Sagot, *Sorting by reversals in subquadratic time*, proceedings of CPM, 2004, pp. 1–13.

[29] R. E. Tarjan, *A class of algorithms that require nonlinear time to maintain disjoint sets*, JCSS **18** (1979), no. 2, 110–127.

[30] Ju wook Jang, Madhusudan Nigam, Viktor K. Prasanna, and Sartaj Sahni, *Constant time algorithms for computational geometry on the reconfigurable mesh*, IEEE Transactions on Parallel and Distributed Systems **8** (1997), no. 1, 1–12.

[31] Jingyuan Zhang, *General prefix computations on enhanced meshes, with applications*, Proceedings of the International Conf. on Computing and Information, 1994, pp. 194–213.