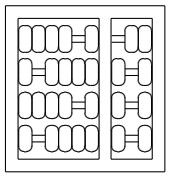


INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Partial Enumeration of Traces of Solutions for the Problem of Sorting by Signed Reversals

C. Baudet *Z. Dias*

Technical Report - IC-11-12 - Relatório Técnico

May - 2011 - Maio

The contents of this report are the sole responsibility of the authors.
O conteúdo do presente relatório é de única responsabilidade dos autores.

Partial Enumeration of Traces of Solutions for the Problem of Sorting by Signed Reversals

Christian Baudet *† Zanoni Dias *

Abstract

Traditional algorithms to solve the problem of sorting by signed reversals output just one optional solution while the space of optimal solutions can be huge. Algorithms for enumerating the complete set of traces of solutions were developed with the objective of supporting the biologists in their studies of alternative evolutionary scenarios. Due to the exponential complexity of the algorithms, their practical use is limited to small permutations. In this work, we propose and evaluate three different approaches for producing a partial enumeration of the complete set of traces of a permutation.

1 Introduction

Reversals have an important role in the evolution of many species. Polynomial algorithms to solve the problem of sorting permutations by signed reversals are already known [11]. However, these algorithms output just one solution while the space of optimal solutions can be huge.

To represent the set of optimal solutions in a more compact way, we can use the concept of traces which was introduced by Bergeron *et al.* [5]. Thanks to the algorithm of Braga *et al.* [8], we are able to enumerate the complete set of traces of solutions that sort a signed permutation by reversals.

Despite this compact representation, the set of trace solutions increases exponentially with the size of the permutations. To reduce the space of solutions, Braga *et al.* incorporated to the algorithm a series of biological constraints [9].

With the aim to improve the algorithm of enumeration of traces, we adopted a stack structure to handle the intermediary data. By using this structure, we are able to extend the processing capacity of the algorithm and reduce the execution time [3].

For big permutations ($n \geq 20$), the necessary time to produce the complete set of traces is generally enormous and makes impracticable any analysis. In this work, we propose new methods which have the objective of producing the maximum possible number of traces considering the imposition of time and memory limits. These methods are able to produce much more traces than the original algorithm when it is submitted to the same limitations.

*Institute of Computing, University of Campinas, 13081-970 Campinas, SP

†UMR CNRS 5558 LBBE, Université Claude Bernard – Lyon I, 69622 Villeurbanne CEDEX, France

2 Sorting by signed reversals

When studying genome rearrangements, we can identify homologous markers with integers $1, \ldots, n$ with a plus or minus sign to indicate the strand they lie on. By using this notation, we can represent with a *signed permutation* the order and the orientation of the genomic markers of one species in relation to another.

A subset of numbers $\rho \subseteq \{1, \ldots, n\}$ is said to be an *interval* of a permutation π if there exist $i, j \in \{1, \ldots, n\}$, $0 \leq i \leq j \leq n$, such that $\rho = \{|\pi_i|, \ldots, |\pi_j|\}$. Two intervals are said to *overlap* if they intersect but none is contained in the other. For example, if $\pi = (+1, -4, +3, +2, -5, -6)$, then $\rho_1 = \{1, 3, 4\}$ and $\rho_2 = \{2, 3, 4\}$ overlap, while $\rho_3 = \{2, 3, 4, 5\}$ and $\rho_4 = \{2, 3\}$ do not.

Given a permutation π and an interval ρ of π, we can apply a reversal on π, that is the operation which reverses the order and flips the signs of the elements of ρ. If $\rho = \{|\pi_i|, \ldots, |\pi_j|\}$, then $\pi \circ \rho = (\pi_1, \ldots, \pi_{i-1}, -\pi_j, \ldots, -\pi_i, \pi_{j+1}, \ldots, \pi_n)$. Due to this, we can use an interval ρ to denote a reversal.

A sequence of reversals $\rho_1 \ldots \rho_k$ sorts a permutation π if ρ_i is an interval of $\pi \circ \rho_1 \cdots \rho_{i-1}$ for all i, and $\pi \circ \rho_1 \cdots \rho_k$ is the identity permutation. A shortest sequence of reversals sorting a permutation is called an optimal sorting sequence. The length of such a sequence of reversals is called *reversal distance*.

The problem of finding an optimal sorting sequence is called *Sorting Permutations by Signed Reversals*. The first polynomial algorithm with complexity $O(n^4)$ was proposed in 1995 by Hannenhali and Pevzner [11]. In 2001, Bergeron presented a quadratic algorithm [4]. In 2004, Tannier, Bergeron and Sagot built the first sub-quadratic algorithm with complexity $O(n^{3/2}\sqrt{\log n})$ [14].

More recently, Swenson *et al.* proposed an $O(n \log n + kn)$ algorithm, where k is the number of successive corrections which must be applied when the algorithm chooses an unsafe reversal. Swenson *et al.* showed a permutation family where k is $\Theta(n)$ (worst-case for k) and, in these conditions, the algorithm is quadratic. However, tests performed by the authors showed that k generally is a constant smaller than 1 and independent of the permutation size. Because of this, the algorithm has, with high probability, execution time $O(n \log n)$ [13].

A linear algorithm, proposed by Bader, Moret and, Yan, can calculate the reversal distance in linear time [1].

3 Enumeration of Traces

The traditional algorithms that solve the problem of sorting permutations by signed reversals output just one optimal solution, while the space of optimal solutions can be huge. For example, the permutation

$$(-4, -11, +6, -9, -2, +1, -8, +3, -10, +7, -5)$$

has 6345019 optimal solutions.

Bergeron *et al.* introduced for the reversal problem the concept of traces that allows the organization of a set of optimal solutions into classes [5]. If sequences of reversals are

identified as words, Bergeron *et al.* define a relation over them: if ρ and θ are reversals (intervals) and they show no overlap, then the words $\rho\theta$ and $\theta\rho$ are said to be equivalent. We say that ρ and θ commute. Based on this relation, any word that has a sub-word $\rho\theta$ is equivalent to the same word that replaces $\rho\theta$ by $\theta\rho$. For example, the sequences of reversals (words) $\{1\}\{1,2,3\}\{2,3,4\}$ and $\{1,2,3\}\{1\}\{2,3,4\}$ are equivalent because the reversals $\{1\}$ and $\{1,2,3\}$ commute. Oppositely, none of these sequences of reversals are equivalent to $\{1\}\{2,3,4\}\{1,2,3\}$ because the reversals $\{1,2,3\}$ and $\{2,3,4\}$ overlap.

A class of optimal reversal sequences over this relation is called a trace. Bergeron *et al.* proposed that for a given signed permutation π, the set of all optimal reversals is a union of traces. Thus, traces can be used to produce a more relevant result to the problem of sorting by reversals because they provide a more compact representation of an enormous set of solutions.

An element s of a trace T is in its *normal form* if it can be decomposed into sub-words $s = u_1|\ldots|u_m$ such that:

- every pair of elements of a sub-word u_i commute;
- for every element ρ of a sub-word u_i $(i > 1)$, there is at least one element θ of the sub-word u_{i-1} such that ρ and θ do not commute;
- every sub-word u_i is a nonempty increasing word under the lexicographic order.

A theorem by Cartier and Foata states that for any trace, there is a unique element that is in normal form [10]. It allows the representation of traces through their normal forms.

The number of sub-words in a trace denotes its height. The size of a trace T is the number of solutions which it represents. The trace $\{1,2,4\}\{3\}|\{1,3,4\}|\{2,3,4\}$ has height 3 and size 4 because it represents just the solutions:

```
{3}{1,2,4}{1,3,4}{2,3,4},  {1,2,4}{3}{1,3,4}{2,3,4},
{1,2,4}{1,3,4}{3}{2,3,4},  {1,2,4}{1,3,4}{2,3,4}{3}.
```

The algorithm proposed by Siepel calculates the set of all optimal 1-sequences of a permutation in time $O(n^3)$ [12]. If we get one reversal ρ of this set and apply it over the original permutation π, we obtain a new permutation π' with reversal distance $d(\pi') = d(\pi) - 1$.

If we apply Siepel's algorithm over π', we obtain a new set of optimal 1-sequences that, combined with ρ, results in a set of optimal 2-sequences. Thus, it is easy to see that, by iterating this algorithm, we can obtain the set of all optimal $d(\pi)$-sequences that sort the permutation π.

The algorithm developed by Braga *et al.* combines Siepel's algorithm with the concept of traces [9]. By using the normal form representation, it is capable of enumerating the set of all traces that represents all possible solutions. First, the algorithm calculates the set of 1-traces (that is equivalent to the set of optimal 1-sequences). Then, at each subsequent step i $(1 < i \leq d(\pi))$, every element of the set of $(i-1)$-traces is used as prefix to build an i-trace.

Additionally, Braga *et al.* adapted their algorithm for the use of biological constraints. By applying constraints, the algorithm can reduce the size of the output through the elimination of traces which do not meet some biological criteria [7–9].

The set of all solutions can be represented in a tree structure where a node A keeps the set of optimal 1-sequences of a given permutation π. Associated with each reversal ρ_i of the node A, we have a sub-tree whose root node contains the list of optimal 1-sequences of the permutation $\pi' = \pi \circ \rho_i$. A path from the root of the tree to a leaf gives an optimal solution that sorts the origin permutation into the identity permutation.

The algorithm proposed by Braga *et al.* explores this tree in breadth-first manner and adopts a complex data structure to keep the intermediary information into the main memory and into the disk [6].

In a previous work, we optimized the algorithm of enumeration of traces by adopting a stack structure to keep the intermediary data just on the main memory (Algorithm 3.1) [3]. Instead of exploring the universe of solutions in breadth-first manner, we adopted a depth-first strategy to explore the branches of the tree. With this solution, we greatly reduced the amount of data which must be kept in the main memory and we eliminated the disk accesses. However, this algorithm cannot be used with most of the biological constraints developed by Braga *et al.* and cannot compute the total number of solutions that is represented by the set of traces.

The time complexity of the algorithm proposed by Braga *et al.* is $O(Nn^{k_{max}+4})$, where k_{max} is the maximum width of a trace and N is the number of traces of solutions. The value k_{max} is, at least, equal to the maximum size of a sub-word u_i in the normal form of a trace [8]. The complexity of the algorithm that uses the stack structure is $O(Nn^4 2^n)$ [2].

Recently, Badr, Swenson and Sankoff adapted the two algorithms of enumeration of traces [2]. The strategy consists in grouping i-traces according to the permutation that they produce when their sequence of reversals are applied to the original permutation. As many traces can produce the same intermediary permutation, by grouping them, the authors avoid unnecessary computations. Instead of generating the set of optimal 1-sequences for every i-trace, they compute this set just for the intermediary permutation which groups a set of i-traces. Despite the gain of 70% over the execution time of the algorithm proposed by Braga *et al.* and 50% over our algorithm which uses the stack structure, the algorithm proposed by Badr, Swenson and Sankoff uses a considerable amount of the main memory to keep the groups of i-traces and permutations.

4 Partial Enumeration of Traces

The main objective of the enumeration of traces is to offer a set of alternative evolutionary scenarios to the biologists. Due to the exponential complexity of the algorithms, their application is limited to small permutations ($n < 20$) even when constraints are applied during the computation. The time necessary to process big permutations makes impracticable their use.

In this work, we introduce the strategy of partial enumeration of traces of solutions. The idea here is to develop algorithms that can output, for big permutations, part of the

Algorithm 3.1: Algorithm which uses a stack structure for enumerating the set of all traces that sort a permutation π

```
Data: Origin permutation π
Result: Set of all traces which sort π
begin
    Create an empty stack S;
    L ← sorted set with all optimal 1-sequences of π;
    Push L onto S;
    while Height(S) > 0 do
        Build a new i-trace T with the first reversal of each level of S (from the
        bottom to the top);
        if Height(P) = d(π) then
            Print T;
            Remove the first reversal of the sorted set that is on the top of S;
        else
            π' ← π ∘ T;
            L ← sorted set with all optimal 1-sequences of π';
            L' ← ∅;
            foreach ρ ∈ L do
                T' ← T + ρ;
                if ρ is on the last position of T' then
                    L' ← L' ∪ ρ;
                end
            end
            Push L' onto S;
            while Top of S has an empty set and Height(S) > 0 do
                Pop the empty set of S;
                Remove the first reversal of the sorted set that is on the top of S;
            end
        end
    end
end
```

complete set of traces.

To perform the partial enumeration of traces, the algorithms must have specific criteria to finish their execution: execution time, number of traces, etc. In this work, we adopted an execution time limit to interrupt the enumeration of traces. By fixing the same time limit for different methods, we can evaluate their performance.

4.1 Random algorithm

A very simple solution for the partial enumeration of traces is the use of a method that constructs random traces.

Let π_0 be the original permutation and π_d be the target permutation, where d is the reversal distance between π_0 and π_d. This method consists in generating a trace through the random selection of a reversal among those in the set of optimal 1-sequences of each permutation π_i which is between π_0 and π_d ($0 \leq i < d$). Algorithm 4.1 describes the necessary steps to build a random trace.

Algorithm 4.1: Algorithm to generate a random trace

Data: π_0 and π_d ($d = d(\pi_0, \pi_d)$)
Result: Random trace which transforms π_0 into π_d
begin
  $T_0 \leftarrow$ empty trace;
  for $i \leftarrow 0$ **to** $d - 1$ **do**
    $\rho \leftarrow$ random reversal from the list of optimal 1-sequences of π_i;
    $T_{i+1} \leftarrow T_i + \rho$;
    $\pi_{i+1} \leftarrow \pi_i \circ \rho$;
  end
  Return T_d;
end

Let t_{max} be the maximum execution time and t_{acc} be the accumulated time since the beginning of the execution. Algorithm 4.2 generates a set of random traces inside of a pre-defined period of time.

4.2 Stack algorithm limited by time

Another simple alternative to produce a set of traces is to use the algorithm of enumeration of traces which uses a stack structure (Algorithm 3.1). We have only to include a verification over the elapsed time to interrupt its execution when the time limit is reached.

4.3 Sliding window

While generating random traces, every time Algorithm 4.2 calls Algorithm 4.1, it explores just one branch of the tree of solutions.

On the other hand, Algorithm 3.1 adopts a different strategy. Every times it climbs to a level $i + 1$ on the tree of traces, it is exploring the whole sub-tree of the current i-trace.

Algorithm 4.2: Algorithm to generate a set of random traces

Data: π_0, π_d and t_{max} ($d = d(\pi_0, \pi_d)$)
Result: Set of random traces which transform π_0 into π_d
begin
  $\mathcal{T} \leftarrow \emptyset$;
  $t_{acc} \leftarrow 0$;
  while $t_{acc} < t_{max}$ **do**
    $t_{begin} \leftarrow$ current time;
    $T \leftarrow$ random trace which transforms π_0 into π_d produced by the Algorithm 4.1;
    $\mathcal{T} \leftarrow \mathcal{T} \cup T$;
    $t_{end} \leftarrow$ current time;
    $t_{acc} \leftarrow t_{acc} + (t_{end} - t_{begin})$;
  end
  Return $\mathcal{T}$;
end

In a tree of traces, just a small percentage of the branches reaches the level of the leaves. This happens because of the occurrence of dead branches. A dead branch denotes a situation where a reversal ρ was added to the current i-trace and it does not appear on the last position of the resulting $(i+1)$-trace (which is represented by another branch of the tree). In fact, for big permutations, Algorithm 3.1 spends most of the time exploring dead branches. Figure 1 shows a tree of traces and its dead branches.

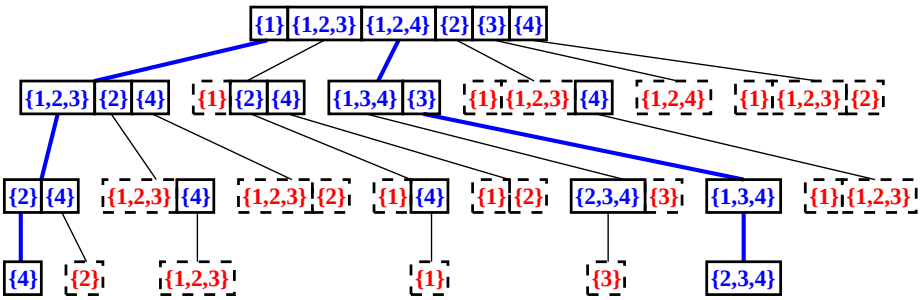


Figure 1: Tree structure that contains all traces that sort the permutation $\pi = (-3, +2, +1, -4)$. Reversals that are inside of dashed boxes indicate the occurrence of a dead branch. A dead branch occurs when a reversal is added to the current i-trace and it does not appear on the last position of the resulting $(i+1)$-traces (which is represented by another branch of the tree).

Let π_k be an intermediary permutation that is obtained after applying the first k reversals of an optimal sequence of reversals which transforms π_0 into π_d ($1 \leq k < d(\pi)$). In this context, we can define the k-trace A and the l-trace B, where $l = d(\pi) - k$. A and B are, respectively, the traces which represent all solutions that transform π_0 into π_k and π_k into

π_d.

If we get the reversals of B and add each one of them, sequentially, into A, we will produce a trace C that transforms π_0 into π_d. This new trace will represent a set of solutions which use the reversals of A and B according to the overlapping relationship that can be found among them. For example, if $A = \{2,4\}\{4\}|\{4,6\}$ and $B = \{3,4,6\}|\{5,6\}$, then $C = \{2,4\}\{4\}|\{3,4,6\}\{4,6\}|\{5,6\}$.

This strategy of combining small traces to construct a bigger one can be used in a sliding window algorithm. In this algorithm, the set of intermediary permutations which is produced by an optimal sequence of reversals is used as a "path" to guide the production of k-traces. Through the combination of these k-traces, we can produce a set of traces that transform π_0 into π_d.

The first step of this algorithm consists in generating a random path of permutations that will be explored. To do this, we can modify Algorithm 4.1 in order to produce a set of intermediary permutations produced by the random sequence of reversals. Instead of returning a random trace, the algorithm returns a list of intermediary permutations and the set of i-traces which transform π_0 into π_i. Algorithm 4.3 shows the modified version of Algorithm 4.1.

Algorithm 4.3: Algorithm that generates a set of intermediary permutations related to a random optimal sequence of reversals which transform π_0 into π_d.

Data: π_0 and π_d ($d = d(\pi_0, \pi_d)$)
Result: Two arrays of length $d+1$ that handle the list of intermediary permutations between π_0 and π_d (P) and the list of i-traces which transform π_0 into π_i (I)
begin
  $T_0 \leftarrow$ empty trace;
  $I \leftarrow$ Array of length $d+1$;
  $P \leftarrow$ Array of length $d+1$;
  $I[0] \leftarrow \emptyset$;
  for $i \leftarrow 0$ **to** $d-1$ **do**
    $P[i] \leftarrow \pi_i$;
    $\rho \leftarrow$ random reversal of the list of optimal 1-sequences of π_i;
    $T_{i+1} \leftarrow T_i + \rho$;
    $I[i+1] \leftarrow T_{i+1}$;
    $\pi_{i+1} \leftarrow \pi_i \circ \rho$;
  end
  $P[d] \leftarrow \pi_d$;
  Return (P, I);
end

With a set of intermediary permutations which transforms π_0 into π_d, we can produce sets of small traces and combine them to build bigger traces. Thus, we can define a window of size w and slide it through the sequence of permutations and produce w-traces that transform π_i into $\pi_{(i+w)}$.

Combining the i-traces that transform π_0 into π_i and the w-traces that transform π_i

into $\pi_{(i+w)}$, we obtain a set of $(i+w)$-traces that transform π_0 into $\pi_{(i+w)}$. To produce the set of all w-traces that transform π_i into $\pi_{(i+w)}$ we can use the Algorithm 3.1.

Let w be the window length. Algorithm 4.4 describes the steps to produce a set of random traces, using a sliding window, respecting a pre-determined time limit.

Algorithm 4.4: Algorithm that generates a set of random traces with a sliding window of length w.

Data: π_0, π_d, t_{max} and w ($d = d(\pi_0, \pi_d)$)
Result: Set of random traces which transform π_0 into π_d
begin
  $\mathcal{T} \leftarrow \emptyset$;
  $t_{acc} \leftarrow 0$;
  while $t_{acc} < t_{max}$ **do**
    $t_{begin} \leftarrow$ current time;
    $(P, I) \leftarrow$ Sets of permutations and i-traces that are output of the Algorithm 4.3 for the permutations π_0 and π_d;
    if $I[d] \notin \mathcal{T}$ **then**
      for $i \leftarrow 0$ **to** $d-1$ **do**
        $k \leftarrow i + w$;
        if $k > d$ **then** $k \leftarrow d$;
        $T \leftarrow$ set of all traces that transform $P[i]$ into $P[k]$ (output of the Algorithm 3.1);
        foreach *i-trace* $x \in I[i]$ **do**
          foreach *$(k-i)$-trace* $y \in T$ **do**
            $z \leftarrow k$-trace produced by the addition of the reversals of y into x;
            $I[k] \leftarrow I[k] \cup z$;
          end
        end
        $I[i] \leftarrow \emptyset$; `/* Free memory */`
      end
      $\mathcal{T} \leftarrow \mathcal{T} \cup I[d]$;
    end
    $t_{end} \leftarrow$ current time;
    $t_{acc} \leftarrow t_{acc} + (t_{end} - t_{begin})$;
  end
  Return $\mathcal{T}$;
end

5 Tests

Before performing comparative tests among the three proposed algorithms, we evaluated the average time which is necessary to enumerate all traces of a given permutation when we use Algorithm 3.1.

To do this, we created sets of random linear and circular permutations with 10 and 15 elements. For the permutations with 10 elements, we considered reversal distances between 4 and 10 for linear permutations and between 4 and 9 for circular permutations. In the case of the permutations with 15 elements, we considered reversal distances between 4 and 13 for linear permutations and between 4 and 12 for circular permutations. For each triplet (permutation type, number of elements, reversal distance) we created a set of 500 random permutations.

Each set was processed by Algorithm 3.1 and we calculated the average number of traces and the average elapsed time. The plots of Figure 2 show the collected values.

Figure 2(a) shows that the number of traces grows exponentially according to the increment of the number of elements and of the reversal distance. For a same reversal distance value, we can see that the number of traces grows with the ratio $d(\pi)/n$. For example, when we fix the value 9 for the reversal distance, permutations with 10 elements have on average more traces than the permutations with 15 elements. We can also see that the average amount of traces observed for circular permutations is bigger than the one observed for linear permutations.

The same observations made for Figure 2(a) can be applied to Figure 2(b) and it indicates that the time is proportional to the number of traces which must be enumerated.

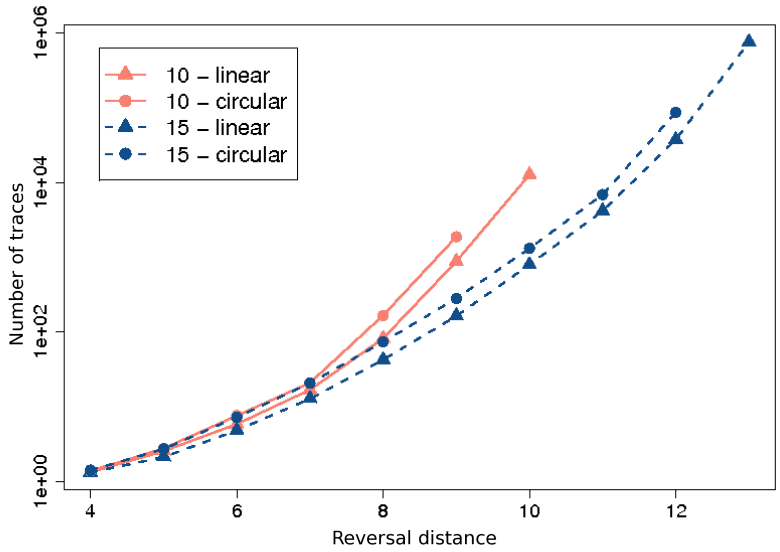
To evaluate the proposed algorithms, we decided to adopt a set of permutations which have an average execution time that is neither too short, nor too long. As the behavior of linear and circular permutations are similar, to simplify the analyses we opted for performing tests only with linear permutations.

Based on these criteria, we chose the set composed by permutations with 15 elements and reversal distance 12. This set has an average execution time of, approximately, 30 seconds. This set was processed by the following algorithms:

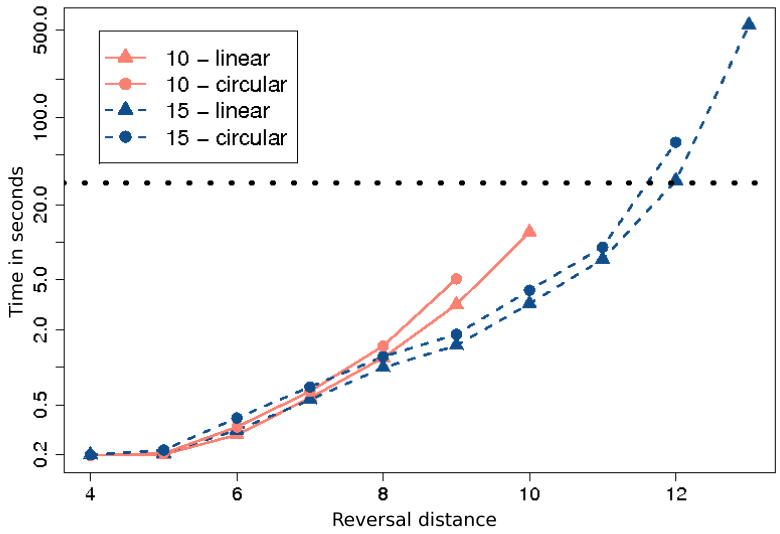
- `Stack` Algorithm 3.1 adapted to interrupt the execution after reaching the time limit;
- `Random` Algorithm 4.2;
- `Window X` Algorithm 4.4 with window of size X.

For `Window X`, we tested the values 4, 5 and 6 for the parameter window size. To facilitate the description along the text, we will refer to these algorithms as, respectively, `Window 4`, `Window 5` and, `Window 6`.

Generally, the bigger is the size w of the window, the bigger is the number of w-traces which are enumerated. On the other hand, while we increase the value of w, the bigger is the amount of dead branches. Due to this characteristic, we chose the values 4, 5 and 6 to evaluate the effect of the window size on the enumeration of traces produced by Algorithm 4.4.



(a) Average number of traces



(b) Average execution time

Figure 2: Sets of 500 random permutations (linear and circular) with 10 and 15 elements and different reversal distances were processed by Algorithm 3.1. For each set, we calculated the average number of traces and the average execution time. The horizontal dotted line in the Figure 2(b) indicates 30 seconds.

`Random`, `Stack`, `Window 4`, `Window 5` and, `Window 6` were used to process the 500 random permutations with $n = 15$ and $d(\pi) = 12$. Considering that the average time to process the traces of this set is 30 seconds, we set up the algorithms with the following time limits: 6, 12, 18, 24, 30 and, 36 seconds.

During the execution of the algorithms, for each enumerated trace, we observed its height (number of sub-words of the trace in normal form). For example, the trace $\{1, 2\}\{2\}$ has height 1 and the trace $\{1, 2\}|\{2, 3\}$ has height 2. The observed values of height for the enumerated traces are inside of the interval $[2, 12]$.

For each triplet (algorithm A, permutation P, time limit T), we counted the number of traces that have height H ($2 \leq H \leq 12$). The same procedure was performed for the set of all traces of P. After that, for each triplet (A, P, T) we calculated the percentage of the traces of P with height H that were found by A during its execution limited by T. Finally, for each triplet (A, H, T) we calculated the average percentage of traces with height H shown by the 500 permutations in the execution of algorithm A inside of the time limit T. The plots of Figure 3 show the average percentage of traces found by each algorithm according to the height of the traces and the imposed time limit.

Figure 3 shows that, among the tested time limits, the bigger is the execution time, the bigger is the number of traces which is enumerated by the algorithms.

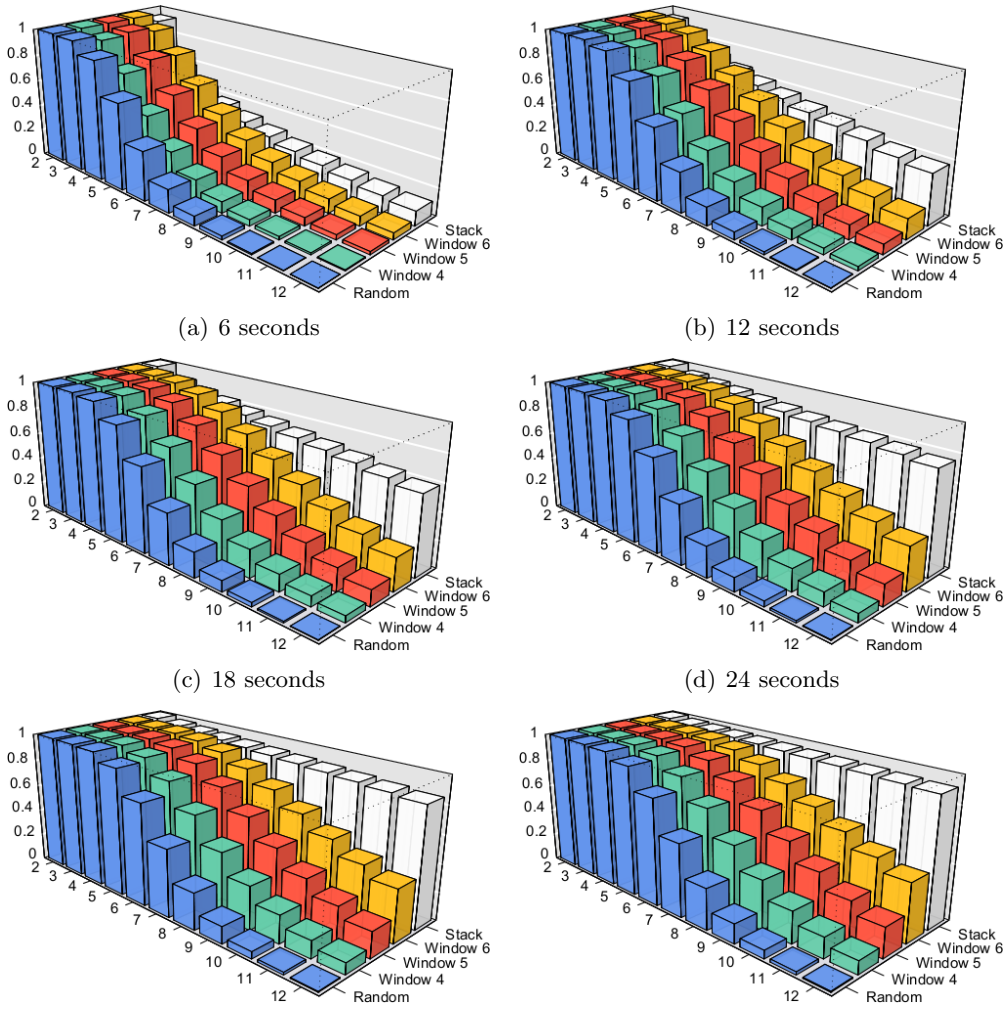
In this set of permutations, `Stack` is the one that showed better performance. As it is not a random algorithm, the increment of the time limit results in a gradual increase in the number of enumerated traces.

The other algorithms show a pattern that is different from the one shown by `Stack`. Low height traces contain less overlapping reversals than high height traces. As the overlapping among reversals determines the order on which they should be applied, a small number of overlaps reflects on a bigger number of sequences of reversals which represent a valid solution. Because of this, low height traces represent set of solutions bigger than the sets of solutions which are represented by high height traces. Due to this, when we generate a trace by the random selection of reversals, the probability of obtaining a low height trace is bigger. Although `Random`, `Window 4`, `Window 5` and, `Window 6` had enumerated almost 100% of the traces with heights 2, 3 and, 4, the percentage of enumerated traces greatly decreases as we increase the height value.

The comparison among `Random`, `Window 4`, `Window 5` and, `Window 6` shows that the algorithm `Random` has the worst results. For the sliding window algorithms, the results get better as we increase the size of the window.

The average time to process this set of permutations ($n = 15$, $d(\pi) = 12$) is, approximately, just 30 seconds. It is a set of permutations whose traces can be easily enumerated. Nonetheless, these algorithms were developed with the objective of enumerating traces of big permutations which demand a huge processing time.

To verify this aspect, we created sets of 100 random permutations with a number of elements varying between 40 and 200 and a reversal distance $d(\pi) = \lceil (n + 1)/2 \rceil$. The generated sets of permutations were processed by `Stack`, `Random`, `Window 4`, `Window 5` and, `Window 6` with a time limit of 60 seconds for each permutation. For each execution, we collected the number of enumerated traces and the maximum amount of memory used by the algorithm. Figure 4 shows for each algorithm and for each value of n, the average



(a) 6 seconds (b) 12 seconds

(c) 18 seconds (d) 24 seconds

(e) 30 seconds (f) 36 seconds

Figure 3: A set of 500 random permutations with $n = 15$ and $d(\pi) = 12$ were processed by `Random`, `Window 4`, `Window 5`, `Window 6` and, `Stack`. The following time limits were imposed to the algorithms: 6, 12, 18, 24, 30 and, 36 seconds. For each triplet (A, H, T) we calculated the average percentage of traces with height H shown by the 500 permutations in the execution of algorithm A inside of the time limit T. On each plot, the axes x, y and, z represent, respectively, the heights, the algorithms and, the average percentage values.

number of enumerated traces and the average memory observed during the executions of each set of 100 permutations.

Analyzing Figure 4(a), we can see that the number of traces that are enumerated by `Stack` decreases as the size of the permutations increases. This phenomenon is associated with the time that this algorithm spends processing dead branches in the tree of traces.

`Random` has a curve very similar to the one shown by `Stack`. For each trace that is generated by `Random`, we select $d(\pi)$ random reversals. A reversal is selected from a list of optimal 1-sequences of the permutation π_i $(0 \leq i < d(\pi))$. It means that, from the first to the last reversal we are working with permutations that have reversal distance between $d(\pi)$ and 1. With the increase in the reversal distance, the time that is spent on the analyses of the breakpoint graph to find a safe reversal also grows. Because of this, the number of traces that are enumerated decreases while we increase the number of elements and the reversal distance while keeping the same time limit.

The sliding window algorithms `Window 4`, `Window 5` and, `Window 6` enumerate more traces than the algorithms `Stack` and `Random`. For permutations with $n < 55$, `Window 5` and `Window 6` outperform `Stack` and `Random`. For permutations with $n \geq 55$, all the sliding window algorithms have better performance than the ones shown by the strategies which do not use a sliding window.

While `Random` works all the time with permutations which have reversal distance values between 1 and $d(\pi)$, the sliding window algorithm adopts a different strategy. If w is the size of the window, except for the step of generation of random paths of permutations, the sliding window algorithm never works with permutations which have reversal distance greater than w. As w is, generally, much smaller than $d(\pi_0)$, we can substantially save time in the generation of the list of optimal 1-sequences.

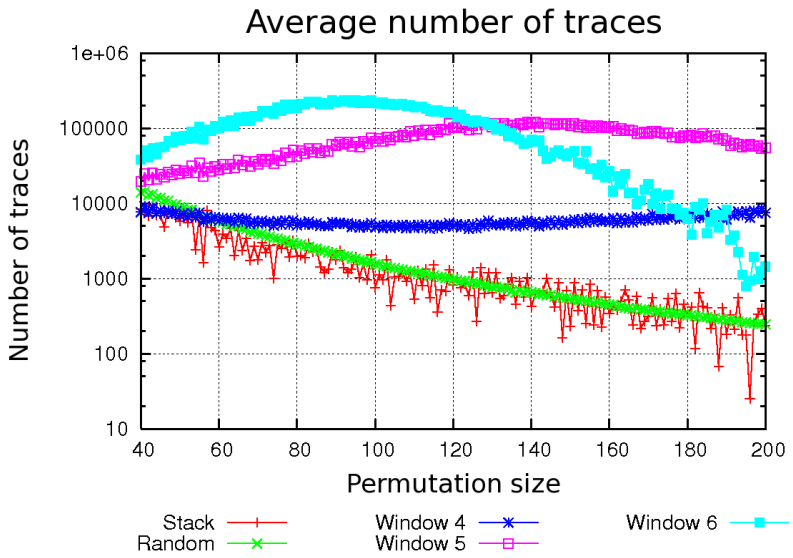
Another advantage of the sliding window strategy is that it produces all w-traces that transform π_i into $\pi_{(i+w)}$. Because of this, we profit of all the structures which are created for the generation of the optimal 1-sequences. In the case of `Random`, even if we avoid to generate all structures, the created ones are partially explored because just one reversal is considered for each intermediary permutation.

If we increase the value of w, we are augmenting the enumeration of w-traces. Thus, it means that we are also raising the number of combinations that we have to perform (i-traces with w-traces for generating $(i+w)$-traces). For $n < 80$, we can see that the curve of `Window 6` grows much faster than the ones of `Window 4` and `Window 5`.

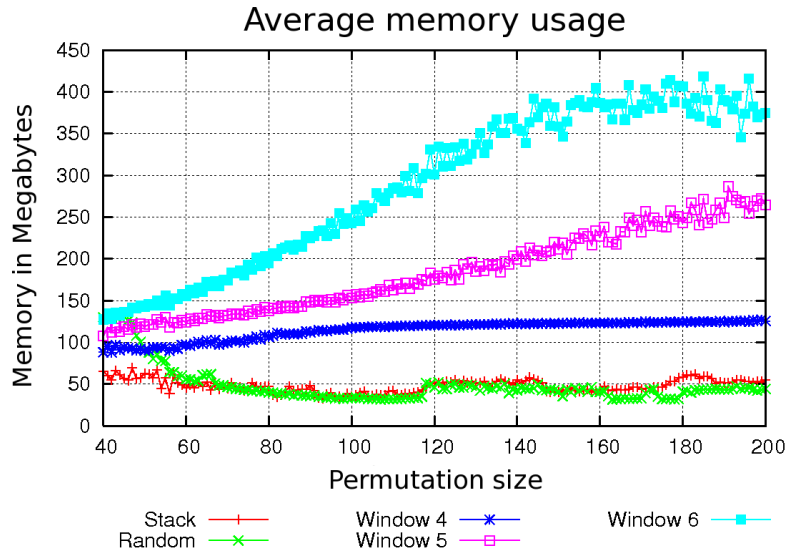
However, the curve of `Window 6` has a parabola shape. The number of enumerated traces grows for $40 \leq n \leq 80$, it reaches the maximum between $80 < n < 110$ and it decreases for $n \geq 110$. For $n \geq 130$, `Window 6` is outperformed by `Window 5` and for $n \geq 180$, it is also outperformed by `Window 4`.

`Window 5` also has a parabolic curve but, as it grows more slowly than the curve of `Window 6`, the interval where we have an ascendant curve goes up to $n = 120$. The maximum is observed in the region $120 < n < 160$ and for $n \geq 160$, the curve is descendant.

This parabolic shape is associated with the process of combination of i-traces with w-traces which were enumerated for the permutations π_i and $\pi_{(i+w)}$. When we combine x i-traces with y w-traces, we can create up to $x \times y$ $(i + w)$-traces (some of the generated traces can appear more than once). Thus, if the reversal distance of the original permutation



(a) Number of traces



(b) Maximum memory

Figure 4: Sets of 100 random permutations with $40 \leq n \leq 200$ and $d(\pi) = \lceil (n+1)/2 \rceil$ were generated and processed with the algorithms `Stack`, `Random`, `Window 4`, `Window 5` and, `Window 6` during a time limit of 60 seconds.

increases, the number of combinations (i-traces + w-traces) and the time that is spent on them also increases.

Generally, a set of 4-traces is smaller than a set of 5-traces and much smaller than a set of 6-traces. Consequently, the number of 4-traces which must be combined during the enumeration will be much smaller than the values observed for windows bigger than 4. `Window 4` shows an average number of traces slightly lower than 10000 along the tested interval.

Figure 4(b) shows that `Random` and `Stack` have a small variation in the average memory. In the considered interval, `Stack` consumes around 50 Megabytes of memory. `Random` initially uses more memory, reaching 130 Megabytes at $n = 40$. This memory usage decreases up to $n = 60$, value where `Random` starts to show the same behavior as `Stack`.

The sliding window algorithms have a bigger memory consumption than the ones shown by `Random` and `Stack`. While `Window 4` has a more stable memory usage, `Window 5` and `Window 6` have an ascending curve of memory utilization.

While the random algorithms can eventually produce the same trace more than once, `Stack` is a deterministic algorithm and outputs every trace just once. Because of this, when using `Stack` we can print the traces avoiding to keep them in memory with the purpose of controlling duplicated traces.

The higher memory usage of `Random` is related to the interval where it outputs more traces. When the number of enumerated traces decreases, the amount of memory that we need to keep the traces in memory also diminishes. As a consequence, the memory consumption reaches a level low enough for the maintenance of the objects which are being used to produce the enumeration.

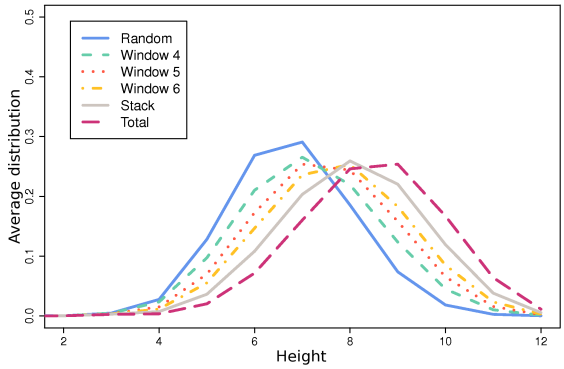
`Window 4`, `Window 5` and, `Window 6` keep in memory the set of enumerated traces and the sets of j-traces ($i \leq j < i + w$) which are going to be combined with the generated w-traces (on each iteration j). To reduce the memory consumption we could print all enumerated traces but, as a result of this, we must add a post-processing step of elimination of copies of traces.

When we perform a partial enumeration of a big set of traces, we must observe if the result is unbiased. It means that we must check if the output of the algorithms covers the space of solutions uniformly.

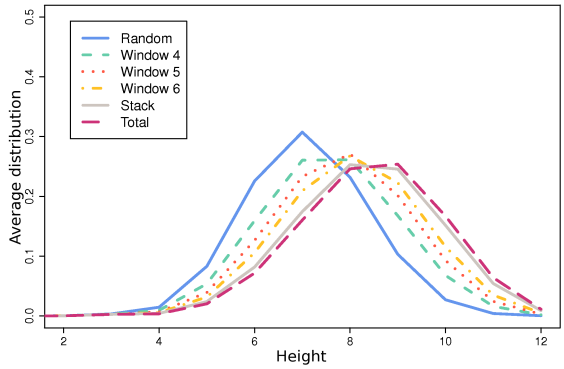
To evaluate this characteristic, we got the set of 500 random permutations with $n = 15$ and $d(\pi) = 12$ and we calculated the distribution of traces, according to their heights, using the complete set of traces and the output of the proposed algorithms.

For each permutation and for each height value, we counted the number of traces that appeared on the complete set of traces. After that, for each triplet (A, P, T), we estimated the distribution of traces by calculating the ratio *number of traces with height H / total number of traces of the permutation P*. Finally, for each triplet (A, T, H) we calculated the average distribution of traces that have height H and were enumerated by A during the time limit L among the set of 500 random permutations. Figure 5 shows the distribution of traces of the sets which were calculated using the following time limits: 6, 18 and 30 seconds.

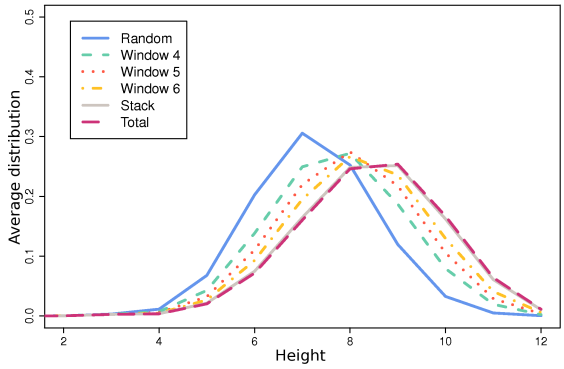
Figure 5 shows that the curves of the proposed algorithms exhibit different levels of approximation to the reference curve (`Total`) according to the time limit.



(a) 6 seconds



(b) 18 seconds



(c) 30 seconds

Figure 5: Average distribution of the traces according to their heights.

`Random` is the algorithm that outputs less traces and it has the tendency to produce low height traces. Because of this, it has the curve that is more distant from the reference curve.

Among the sliding window algorithms, `Window 6` has the best approximation because it can enumerate more traces than `Window 4` and `Window 5`.

`Stack` showed the best results. As this testing set is composed of small permutations, `Stack` can enumerate most of the traces and, consequently, it shows the best approximation to the reference curve.

Analyzing Figure 5, we believe that, except for `Random`, the proposed algorithms can produce unbiased sets of traces. If it was not true, the distribution curve of the algorithms would not show the tendency of getting closer to the reference curve. The quality of the approximation is directly related to the available execution time.

6 Conclusions

A partial enumeration of traces is a feasible strategy for the production of alternative evolutionary scenarios for permutations whose enumeration of the complete set of traces demands a huge computation time.

`Stack` is a strategy that is suited for small permutations. The time lost while exploring dead branches degrades its performance substantially.

`Random` shows a performance similar to the one shown by `Stack`. However, it is a random algorithm and, because of this, it has the advantage of producing distinct sets of traces for each execution while `Stack` always outputs the same set.

The sliding window algorithm is capable of enumerating more traces than the other two strategies. However, the size of the window must be chosen according to the reversal distance of the processed permutations. The available memory is also an important factor that must be taken into account while using this strategy.

`Random` has an interesting advantage over the other algorithms. As it does not use Algorithm 3.1, it can be easily adapted for the utilization of biological constraints during the selection of random reversals.

A future step of this work is to create a method of classification of the produced traces with the objective of ranking the traces (according to some biological criteria) and of giving some support to the biologist who wishes to study alternative evolutionary scenarios.

7 Acknowledgments

Christian Baudet was supported by CAPES – Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (4676/08-4). Zanoni Dias is partially sponsored by CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico (483177/2009-1, 473867/2010-9 and 200815/2010-5).

This work is also supported by the ERC Advanced Grant SISYPHE awarded to Marie-France Sagot, INRIA, France.

References

[1] D. A. Bader, B. M. E. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001.

[2] G. Badr, K. Swenson, and D. Sankoff. Listing all parsimonius reversal sequences: New algorithms and perspectives. In E. Tannier, editor, *Proceedings of the 8th Annual RECOMB Satellite Workshop on Comparative Genomics (RECOMB-CG 2010)*, volume 6398 of *Lecture Notes in Bioinformatics*, pages 39–49, Ottawa, Canada, October 2010. Springer-Verlag Berlin Heidelberg.

[3] C. Baudet and Z. Dias. An improved algorithm to enumerate all traces that sort a signed permutation by reversals. In *Proceedings of the 25th Symposium On Applied Computing (ACM SAC 2010)*, Sierre, Switzerland, March 2010. 5 pages, Bioinformatics Track.

[4] A. Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. In *Proceedings of the 12th Annual Symposium of the Combinatorial Pattern Matching (CPM'2001)*, volume 2089 of *Lecture Notes in Computer Science*, pages 106–117, Jerusalem, Israel, 2001.

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

[6] M. D. V. Braga. *Exploring the Solution Space of Sorting by Reversals When Analyzing Genome Rearrangements*. PhD thesis, Université Lyon 1, France, 2008.

[7] M. D. V. Braga, C. Gautier, and M.-F. Sagot. An asymmetric approach to preserve common intervals while sorting by reversals. *Algorithms for Molecular Biology*, 4(1):16, 2009.

[8] M. D. V. Braga, M.-F. Sagot, C. Scornavacca, and E. Tannier. The solution space of sorting by reversals. In *Proceedings of the International Symposium on Bioinformatics Research and Applications 2007 (ISBRA 2007)*, volume 4463 of *Bioinformatics Research and Applications*, pages 293–304. Springer Berlin / Heidelberg, 2007.

[9] M. D. V. Braga, M.-F. Sagot, C. Scornavacca, and E. Tannier. Exploring the solution space of sorting by reversals with experiments and an application to evolution. *Transactions on Computational Biology and Bioinformatics*, 5(3):348–356, 2008.

[10] P. Cartier and D. Foata. *Problèmes combinatoires de commutation et réarrangements*. Number 85 in Lecture Notes in Mathematics. Springer-Verlag, Berlin, 1969.

[11] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1):1–27, January 1999.

[12] A. C. Siepel. An algorithm to enumerate sorting reversals. *Journal of Computational Biology*, 10(3–4):575–597, 2003.

[13] K. M. Swenson, V. Rajan, Y. Lin, and B. M. E. Moret. Sorting signed permutations by inversions in $O(n \log n)$ time. *Journal of Computational Biology*, 17(3):489–501, 2010.

[14] E. Tannier, A. Bergeron, and M.-F. Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155:881–888, April 2007.