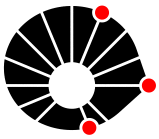

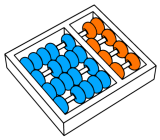

Machine Learning Applied to Sorting Permutations by Reversals and Transpositions

Flavio Altiner Maximiano da Silva *Andre Rodrigues Oliveira* *Zanoni Dias*

Relatório Técnico - IC-PFG-17-03
Projeto Final de Graduação
2017 - Julho

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

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

Machine Learning Applied to Sorting Permutations by Reversals and Transpositions

Flavio Altinier Maximiano da Silva* Andre Rodrigues Oliveira†
Zanoni Dias†

Abstract

The problem of determining relationship trees between genomes is fundamental when studying life evolution on the planet. As mutations are rare, it is believed that when one genome transforms into another, it probably used the fewest operations possible. If we represent genomes as numeric permutations, we reduce that problem to the one of sorting permutations using specific operations. In this work we use two of the most common genome mutations: reversals and transpositions. We propose a machine learning approach where a classifier is trained on a set of features of small permutations and then used to sort bigger permutations. Results show that this method is competitive when compared to others in literature, specially when dealing with small permutations or considering the others' maximum approximation factors.

1 Introduction

Sequence comparison of different genomes is an important study for deriving evolutionary relationships between genes. In this sense, computational molecular biology is becoming widely used in genetics, for it is capable of aiding researchers by highlighting affiliations between genes which could be difficult for a human to perceive.

Still, classical algorithms usually focus on individual nucleotides mutations (which are the most common), but neglect other global rearrangements, i.e., movements of full fragments of the DNA filament. In this project, we focus primarily on those global mutations, which operate on long fragments of genes.

Two of the most common mutations which occur during genome replication are reversals and transpositions. When a fragment of the DNA filament gets reversed in the final replica, that mutation is called a *reversal*. On the other hand, if two fragments of DNA change places during the replication process (but do not get reversed), that mutation is called a *transposition*.

Phylogenetics (the study of evolutionary history and relationships between species) relies strongly on the Principle of Parsimony: the idea that, given a set of possible explanations for a fact, the simplest explanation is most likely to be correct. As mutations are relatively rare,

*flavio.maxi@gmail.com
†Institute of Computing, University of Campinas, Brazil.

when scientists try to put together an evolutionary relationship tree between species, they try to make it so that species have the fewest common ancestors as possible. For example, it is much more likely that two species that have both prominent incisor teeth have evolved from the same common ancestor which developed that trait, rather than believe that trait evolved twice, from different species.

Determining the minimum number of mutations necessary for one genome to transform into another, however, is not an easy problem to solve. In this work, we try to address that problem when it is known that only reversal and transposition mutations have occurred.

Consider the numeric sequence $(1, 2, \ldots, n)$ of size n as the original genome. Given any permutation of the elements of that sequence as the final genome, the problem consists in creating an efficient sorting algorithm (that ideally uses the least number of operations possible) for the sequence where only reversal and transposition operations are allowed.

Another common approach to the representation of genomes is using the *orientation* of the genes, in which case we represent the genome with $+$ and $-$ symbols for each gene, e.g., $(+1, -2, \ldots, -n)$. That is called in the literature a *signed permutation*; if we do not have information about the orientation of the genes, these signs are omitted and it is called an *unsigned permutation*. In this work, we focused primarily on unsigned permutations.

Many have addressed the problem of sorting permutations by genome rearrangements using different algorithms. Using only reversal operations, first Kececioglu and Sankoff [15] proposed a 2.0-approximation algorithm which focused on removing reversal breakpoints (a concept later described in this document). Later on, Bafna and Pevzner [2] created a 1.75-approximation using the *breakpoints graph* of the permutation, and Christie [10] a 1.5-approximation using its *reversals graph*. The current state of the art was developed by Berman et al. [4]: a 1.375-approximation algorithm. Also worth mention are the works by Berman and Karpinski [5] (in which they proved that the best possible heuristic is at most a 1.0008-approximation) and by Caprara [8] (who proved the problem is NP-hard).

When working with signed permutations, though, the problem of sorting by reversals has been proved to be polynomial. Hannenhalli e Pevzner [14] were the first to design a polynomial algorithm, with complexity $O(n^4)$. The state of the art today was proposed by Tannier et al. [19]: an exact algorithm with $O(n^{\frac{3}{2}}\sqrt{\log n})$ complexity. If the operations themselves are not needed, just the number of operations necessary for sorting, there is a linear time algorithm proposed by Bader et al. [1].

When we consider only transpositions, we can again cite Bafna and Pevzner [3], who presented a 1.5-approximation algorithm and showed derivations on lower bounds on transposition distances (number of transpositions necessary to rearrange the permutation). Later on, Christie [11] created a simpler algorithm with the same approximation factor. The state of the art was proposed by Elias and Hartman [13], in 2006: a 1.375-approximation heuristic. It was only in 2011 that Bulteau, Fertin and Rusu [6] were able to prove that sorting by transpositions is also NP-hard, with a reduction of the SAT problem.

Most algorithms in the literature consider some features of the permutations for sorting, and try to find other permutations which can be reached by performing one operation on it, trying to incur precise alterations on those features. It is only natural to think that machine learning algorithms might be more efficient on reaching that goal; the literature,

however, falls short when it comes to that.

In this work, we propose a machine learning approach for sorting permutations by reversals and transpositions, which is trained on small and manageable permutations and can be adapted for bigger permutations, and investigate how it performs when compared to other heuristics presented in literature.

2 Definitions

In this section we expatiate on common operations and definitions when managing genomes.

Genomes can be represented by permutations (n-tuples where n is the number of genes), where each gene is represented by a number, and we assume that all genes are different. In other words, a permutation π of size n is represented by $\pi = (\pi_1\ \pi_2\ \dots\ \pi_n)$, with $\pi_i \in \{1, 2, \dots, n\}$ and $\pi_i \neq \pi_j \iff i \neq j$.

In this sense, we define ι as the *identity permutation*, i.e., $\iota = (1\ 2\ \dots\ n)$.

The *composition* of two permutations π and σ is given by $\pi \cdot \sigma = (\pi_{\sigma_1}\ \pi_{\sigma_2}\ \dots\ \pi_{\sigma_3})$. We define the *inverse* of a permutation π as π^{-1}, such that $\pi^{-1} \cdot \pi = \iota$.

The *extended notation* of a permutation $\pi = (\pi_1\ \pi_2\ \dots\ \pi_n)$, denoted by π_e, is defined as $\pi_e = (0\ \pi_1\ \pi_2\ \dots\ \pi_n\ n+1)$, i.e., two elements are added to π: 0 before the first element of the permutation, and $n+1$ after the last element. Defining the extended notation of a permutation is necessary for performing operations further described later in this section.

Given two permutations π and σ and a set of operations $M = \{p_1, p_2, \dots, p_k\}$, the problem of transforming π into σ consists of performing the least possible operations of M on π so that it becomes equal to σ. The number of operations needed is called the *distance* between π and σ, and is represented by $d(\pi, \sigma)$. To make notations simpler, we make $d(\pi, \iota) = d(\pi)$.

Note that the problem of finding the distance between permutations π and σ is equivalent to the problem of finding the distance between some permutation α and the identity ι, where $\alpha = \sigma^{-1} \cdot \pi$. That is true because $d(\pi, \sigma) = d(\sigma^{-1} \cdot \pi, \sigma^{-1} \cdot \sigma) = d(\alpha, \iota) = d(\alpha)$. So, if we want to find the distance between two permutations π and σ, that problem is exactly the same as sorting $\sigma^{-1} \cdot \pi$. For example, consider $\pi = (1\ 3\ 2\ 5\ 4)$ and $\sigma = (2\ 4\ 5\ 1\ 3)$: we have that $\sigma^{-1} = (4\ 1\ 5\ 2\ 3)$, what makes $\alpha = \sigma^{-1} \cdot \pi = (5\ 1\ 4\ 3\ 2)$. So the distance between π and σ is the same as the sorting distance for α.

For the symmetric group S_n (the set of all permutations of size n), the maximum distance in S_n is called the *diameter* of the symmetric group, and is represented by $D(n)$.

This *permutation sorting* work consists then on using machine learning models to try to approximate $d(\pi, \iota) = d(\pi)$.

2.1 Reversal

A reversal operation $p_r(i, j)$, with $1 \leq i < j \leq n$ reverts a fragment of the permutation, i.e., $p_r(i, j)$ applied to $\pi = (\pi_1\ \dots\ \pi_{i-1}\ \underline{\pi_i\ \dots\ \pi_j}\ \pi_{j+1}\ \dots\ \pi_n)$ generates $\pi \cdot p_r(i, j) = (\pi_1\ \dots\ \pi_{i-1}\ \underline{\pi_j\ \dots\ \pi_i}\ \pi_{j+1}\ \dots\ \pi_n)$.

Consider, for example, $\pi = (1\ 3\ 2\ 5\ 4\ 7\ 6)$ and the reversal operation $p_r(2, 5)$. We have then that $\pi \cdot p_r(2, 5) = (1\ \underline{4\ 5\ 2\ 3}\ 7\ 6)$.

2.2 Transposition

A transposition operation $p_t(i, j, k)$ with $1 \leq i < j < k \leq n + 1$ transposes two adjacent fragments of the permutation, the first from i to $j - 1$ and the second from j to $k - 1$, i.e., $p_t(i, j, k)$ applied to $\pi = (\pi_1 \ \ldots \ \pi_{i-1} \ \underline{\pi_i \ \ldots \ \pi_{j-1}} \ \underline{\pi_j \ \ldots \ \pi_{k-1}} \ \pi_k \ \ldots \ \pi_n)$ generates $\pi \cdot p_t(i, j, k) = (\pi_1 \ \ldots \ \pi_{i-1} \ \underline{\pi_j \ \ldots \ \pi_{k-1}} \ \underline{\pi_i \ \ldots \ \pi_{j-1}} \ \pi_k \ \ldots \ \pi_n)$.

Consider again, for example, $\pi = (1\ 3\ 2\ 5\ 4\ 7\ 6)$ and the transposition operation $p_t(2, 5, 8)$. By applying that operation on π, we have that $\pi \cdot p_t(2, 5, 8) = (1\ \underline{4\ 7\ 6}\ \underline{3\ 2\ 5})$.

2.3 Breakpoints

Breakpoints are divided into two categories: *reversal breakpoints* and *transposition breakpoints*.

Reversal breakpoints count the number of elements which are not side-by-side with their identity-neighbors. Formally, a reversal breakpoint happens when, for element π_i and π_{i+1} of permutation π, $|\pi_i - \pi_{i+1}| \neq 1$. The identity is the only permutation with no reversal breakpoints. For example, given $\pi = (3\ 4\ 1\ 2\ 6\ 5)$, we have two reversal breakpoints: one between elements $(4, 1)$ and another between elements $(2, 6)$.

Transposition breakpoints happen when an element breaks sequence. Formally, a transposition breakpoint happens when, for elements π_i and π_{i+1} of permutation π, $\pi_{i+1} - \pi_i \neq 1$. The identity is, again, the only permutation with no transposition breakpoints. Consider the same example permutation $\pi = (3\ 4\ 1\ 2\ 6\ 5)$: it has three transposition breakpoints, between elements $(4, 1)$, elements $(2, 6)$ and elements $(6, 5)$.

2.4 Strips

Strips are fragments of a permutation that do not present breakpoints. Given a permutation π, a strip of π is a sequence of elements $\pi_i \ldots \pi_j$, with $0 < i \leq j \leq n + 1$ such that:

1. (π_{i-1}, π_i) is a breakpoint;
2. Either $j = n + 1$, or $j < n + 1$ and (π_j, π_{j+1}) is a breakpoint.
3. There is no breakpoint between $(\pi_k, \pi_{k+1})\ \forall\ k \mid i \leq k \leq j - 1$

If elements of a strip, in order, form an ascending sequence, it is called an *increasing strip*. Otherwise, it is a *decreasing strip*. Finally, if a strip contains only one element, it is considered a special case of an increasing strip, called *singleton*.

For example, take the permutation $\pi = (1\ 4\ 5\ 6\ 3\ 2\ 7)$. Considering only reversal breakpoints, π has four strips: two singletons $\langle 1 \rangle$ and $\langle 7 \rangle$, the increasing strip $\langle 4, 5, 6 \rangle$, and the decreasing strip $\langle 3, 2 \rangle$. Taking only transposition breakpoints into account, on the other hand, there are five strips in π: singletons $\langle 1 \rangle$, $\langle 3 \rangle$, $\langle 2 \rangle$ and $\langle 7 \rangle$, and the increasing strip $\langle 4, 5, 6 \rangle$.

2.5 Longest Increasing and Decreasing Subsequences

First, we must define *subsequence*: let $X = [x_1\ x_2\ \ldots x_n]$ be a sequence of integers. A sequence $Y = [y_1\ y_2\ \ldots y_m]$ is a subsequence of X if $Y \subseteq X$, $m \leq n$ and order is preserved, i.e., $\forall\ y_i, y_j$, with $i < j$, $\exists\ x_k, x_l$ with $k < l$ such that $y_i = x_k$ and $y_j = x_l$. For example, consider the permutation $X = (4\ 2\ 3\ 5\ 1\ 7\ 6\ 8)$: one possible subsequence of X is $Y = (2\ 3\ 5\ 1\ 7)$.

The *Longest Increasing Subsequence* (LIS) of a permutation is the subsequence which has the maximum possible number of elements in increasing order. A permutation may have more than one LIS. Consider again the example $X = (4\ 2\ 3\ 5\ 1\ 7\ 6\ 8)$: one possible LIS of X is $Y = (2\ 3\ 5\ 6\ 8)$.

The *Longest Decreasing Subsequence* (LDS) of a permutation, on the other hand, is the longest subsequence with elements in decreasing order. A permutation may also have more than one LDS. Considering the same $X = (4\ 2\ 3\ 5\ 1\ 7\ 6\ 8)$, one possible LDS of X is $Y = (4\ 2\ 1)$.

2.6 Entropy

Entropy measures the disorganization of a permutation. We can calculate the entropy of an element π_i of permutation π as $entropy(\pi_i) = |\pi_i - i|$, i.e., the distance between π_i and its position in the identity. It is easy then to define the entropy of π as:

$$entropy(\pi) = \sum_{i=1}^{n} entropy(\pi_i)$$

For example, consider permutation $\pi = (4\ 2\ 3\ 5\ 1\ 6)$. To get π's entropy, we first iterate on the elements getting their entropy:

- Entropy of $\pi_1 = |4 - 1| = 3$
- Entropy of $\pi_2 = |2 - 2| = 0$
- Entropy of $\pi_3 = |3 - 3| = 0$
- Entropy of $\pi_4 = |5 - 4| = 1$
- Entropy of $\pi_5 = |1 - 5| = 4$
- Entropy of $\pi_6 = |6 - 6| = 0$

So, in the end we have that $entropy(\pi) = 3 + 1 + 4 = 8$.

2.7 Left and Right Coding

The left code of an element π_i of a permutation π is defined as the number of elements greater than π_i that are positioned left of it in π. Likewise, the right code of π_i is the number of elements smaller than π_i positioned right of it in π.

The *left code* of a permutation π is then defined as the array of left-codes for each element of π (similarly for the *right code* of a permutation).

The *left plateau* of a permutation is the size of the maximal sequence of elements which have the same not-null code (similarly for the *right plateau*).

For example, take permutation $\pi = (4\ 2\ 3\ 5\ 1\ 6)$. Now, for each element, we populate an array L with the left code of each element in π:

- Left code of $\pi_1 = 0$
- Left code of $\pi_2 = 1$ (because $4 > 2$ and is positioned left of it)
- Left code of $\pi_3 = 1$ (because $4 > 3$ and is positioned left of it)
- Left code of $\pi_4 = 0$
- Left code of $\pi_5 = 4$ (because $4, 2, 3$ and 5 are left of it)
- Left code of $\pi_6 = 0$

So now we have that the left code of permutation π is given by $L = (0\ 1\ 1\ 0\ 4\ 0)$. With that, we can say that the left plateau of π, denoted $pl(lc(\pi)) = 2$, because we have the following not-null sequences of the same value: $(1\ 1)$ and (4).

Using the same example to get the right plateau of π, first we must calculate the right code of π, $R = (3\ 1\ 1\ 1\ 0\ 0)$. The right plateau of π is then $pl(rc(\pi)) = 2$, because we have two not-null sequences of the same value: (3) and $(1\ 1\ 1)$.

2.8 Correct Prefix and Suffix

Given a permutation π, its *correct prefix* is defined as the number of elements, from left to right, that are in the correct position before finding one in the wrong place. For example, consider permutation $\pi = (1\ 2\ 3\ 5\ 4\ 6)$. The correct prefix of π is then equal to 3, because we can find three elements in the correct position before finding one incorrect.

Similarly, the *correct suffix* of a permutation is the number of elements in the right place, from right to left, until finding one in the wrong position. For example, if $\pi = (3\ 2\ 4\ 1\ 5\ 6)$, the correct suffix of π is 2 (elements 5 and 6).

The identity is the only permutation with *correct prefix* = *correct suffix* $= n$.

2.9 Inversions

Given a permutation π, there is an inversion between elements π_i and π_j if $|\pi_i| > |\pi_j|$ and $j > i$, i.e., π_j should appear before π_i, but is actually right of π_i.

The number of inversions of π, denoted by $inv(\pi)$ is defined as the total number of inversions present in the permutation, when comparing every possible pair of elements.

For example, take $\pi = (2\ 5\ 3\ 4\ 1\ 6)$. We can, for each element π_i, calculate $inv(\pi_i)$:

- $inv(\pi_1) = 1$, because element 1 is to the right of it.

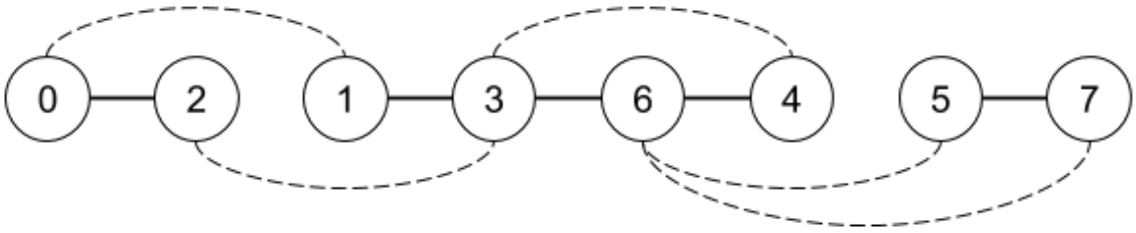

Figure 1: Breakpoints graph of permutation (2 1 3 6 4 5), where solid lines represent black labeled edges, and dashed lines represent gray labeled edges.

- $inv(\pi_2) = 3$, because $3, 4$ and 1 are to the right of it.
- $inv(\pi_3) = 1$, because element 1 is to the right of it.
- $inv(\pi_4) = 1$, because element 1 is to the right of it.
- $inv(\pi_5) = 0$.
- $inv(\pi_6) = 0$.

So we have that $inv(\pi) = 1+3+1+1 = 6$. In this example we calculated the number of inversions element by element (a $O(n^2)$ algorithm, comparing every pair of elements). We can, however, calculate $inv(\pi)$ faster, using a modified version of merge-sort [17].

2.10 Cycles

The concept of cycles is fundamental in the study of sorting permutations, and many of the features we use to train the machine learning sorting algorithm depend heavily on cycles.

To define cycles, we must first define the *breakpoints graph* of a permutation π, denoted by $G_b(\pi)$. $G_b(\pi)$ is formed by a set of vertices $V = \{\pi_0, \pi_1, \ldots, \pi_n, \pi_{n+1}\}$ (where each vertex corresponds to an element in the extended notation of π) and a set E of edges, which can be either labeled as *black* or *gray*. Edges are positioned as follows:

- There is a black edge between vertices π_i and π_{i+1} (with $0 \leq i \leq n$) if there is a reversal breakpoint between π_i and π_{i+1}.
- There is a gray edge between vertices π_i and π_j (for $0 \leq i < j \leq n+1$) if $|\pi_i - \pi_j| = 1$ (i.e., they are adjacent in the identity) and π_i and π_j are not adjacent in π.

An example of a breakpoints graph, for permutation π = (2 1 3 6 4 5), is shown on Figure 1.

The *cycle graph* of π, noted $G(\pi)$, can then be obtained from $G_b(\pi)$. To do so, we must first add one black edge and one gray edge between all elements where $|\pi_i - \pi_{i+1}| = 1$ and they are adjacent in π: these new edges form the set $\bar{c}_b(\pi)$. A visual representation of the

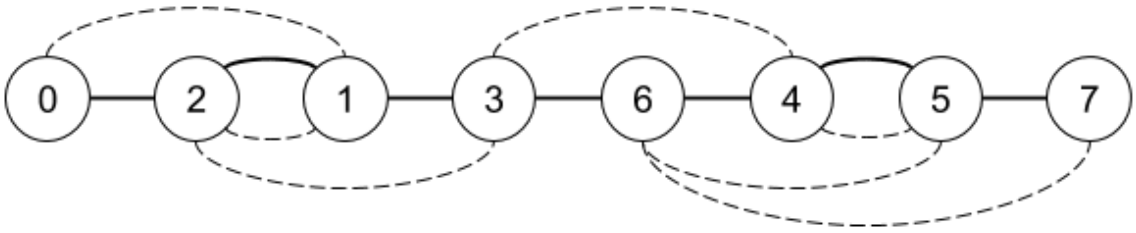

Figure 2: Breakpoints graph of permutation (2 1 3 6 4 5) with the extra edges in $\bar{c}_b(\pi)$, where solid lines represent black labeled edges, and dashed lines represent gray labeled edges.

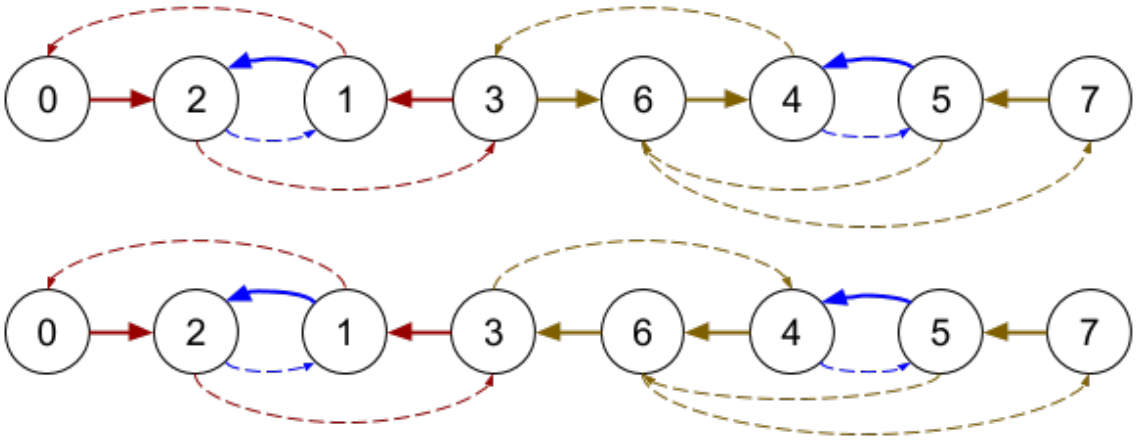

Figure 3: Two possible maximal decompositions of permutation $\pi = (2\ 1\ 3\ 6\ 4\ 5)$ in alternating cycles. Solid lines represent black edges, and dashed lines represent gray edges. Red and yellow oriented edges represent cycles in $G_b(\pi)$, and blue edges are the ones in $\bar{c}_b(\pi)$.

breakpoints graph for the permutation $\pi = (2\ 1\ 3\ 6\ 4\ 5)$ with those new edges added can be seen in Figure 2.

Now, a *cycle graph* of π, denoted by $G(\pi)$, is formed by the maximal decomposition of $G_b(\pi) \cup \bar{c}_b(\pi)$ into alternating cycles, i.e., cycles in which the edges alternate between black and gray. Two possible maximal decompositions for permutation $\pi = (2\ 1\ 3\ 6\ 4\ 5)$ can be seen in Figure 3.

To find such maximal decomposition into alternating cycles has been proved to be a NP-hard problem [7]. The best know approximation, proposed by Chen [9], is $1.4167 + \epsilon$, for $\epsilon > 0$.

Given the cycle graph $G(\pi)$ of permutation π, the black edges are labeled 1 to $n+1$ (so edge (π_i, π_{i-1}) is labeled i).

A *k-cycle* is defined as a cycle with k black edges. The *parity* of a cycle is determined by the parity of k: if the cycle has an even number of black edges, it is called an *even cycle*;

otherwise, it is an *odd cycle*. The identity is the only permutation with $n+1$ odd cycles. A k-cycle with $k \leq 3$ is also called a *short* cycle; if $k > 3$, it is called a *long* cycle.

A cycle C is represented by the vertices in it, in order (e.g., $C = (v_1, v_2, v_3, \ldots)$). It is a convention to make v_1 the right-most element of the cycle (considering original positions in π). A simplified form of C notes only the black edges in it, such as $\{(v_1, v_2), (v_3, v_4), \ldots\}$.

We can also label the black edges in the cycle: if it goes from right to left in π's original order, it is a positive edge. On the other hand, if it goes from left to right, it is a negative edge. As for any cycle v_1 is always the right-most vertex, the edge (v_1, v_2) is always positive.

A k-cycle (with $k \geq 3$) is called *non-oriented* if its black edges are listed in decreasing order; otherwise, it is called an *oriented* cycle.

If all black edges in a k-cycle are positive, it is called a *convergent cycle*. Otherwise, it is a *divergent cycle*.

3 Methodology

Now that all definitions and nomenclatures are explained, we can go on to the methodology of the experiments.

The foundation of this project is built on the idea that we can train a machine learning algorithm to sort small permutations and then use that same algorithm to sort bigger permutations.

3.1 Training Data

The training data used in this project considers all permutations of size $n = 10$. We define $M = \{p_1, p_2, \ldots, p_k\}$ as the set of allowed operations, which are all possible reversals and transpositions. As our permutations have size 10, there are 45 possible reversals and 165 possible transpositions, summing up to a total of $|M| = 210$ operations. Also, it is easy to show that there are $10! = 3628800$ possible permutations of size 10.

We have then, for every permutation π^0 of size 10, applied all possible operations of $M = \{p_1, p_2, \ldots, p_k\}$. We denote the result of operation p_m on π^0 as π^m, i.e., $\pi^m = p_m(\pi^0)$. We then calculate $d(\pi^m)$ and compare it to $d(\pi^0)$:

1. if $d(\pi^m) < d(\pi^0)$ (i.e., π^m needs less operations than π^0 to be transformed into ι) then we label $p_m(\pi^0)$ as a *good* operation on π^0.

2. if $d(\pi^m) > d(\pi^0)$ (i.e., π^m needs more operations than π^0 to be transformed into ι) then we label $p_m(\pi^0)$ as a *bad* operation on π^0.

3. otherwise, i.e., $d(\pi^m) = d(\pi^0)$, we label $p_m(\pi^0)$ as a *neutral* operation on π^0.

and those are the labels we use to train our classifier.

After generating π^m, we can compare many characteristics of π^m and π^0, and use them as features for our classifier. Table 1 describes each one of the 30 selected features for the feature vector $F = (f_1, f_2, \ldots, f_{30})$; they are all calculated considering:

$$f_b = \frac{\text{value of } b \text{ in } \pi^m - \text{value of } b \text{ in } \pi^0}{\text{maximum possible value for } b}$$

As we divide every feature by the maximum value b can assume, it is easy to see every feature is normalized to values between -1 and 1. As normalization is intrinsic to the features, the same definition can be extrapolated to bigger permutations and hopefully generate similar classification results.

For example, consider we are generating the final feature vector for permutation $\pi = (10\ 3\ 4\ 5\ 9\ 8\ 6\ 7\ 1\ 2)$ when applying operation $p_t(2, 5, 7)$, transforming in into $\pi^m = (10\ \underline{9\ 8}\ \underline{3\ 4\ 5}\ 6\ 7\ 1\ 2)$. First we must calculate all features for permutation π, then for π^m, then finally use those values to populate the feature vector, as can be seen in Table 2.

In summary, our training data consists of features extracted from every possible operation applied on every possible permutation of size 10, and considers if that is a good, bad or neutral transformation.

3.2 Classifier

The classifier chosen for this project is the Stochastic Gradient Descent Classifier (SGDC) from python's *scikit-learn 0.18.1* package.

As we have a relatively large volume of data ($10! \times 210$ samples, using more than 110GB of storage), it would be unfeasible to use all of this input as training data. Instead, we decided to use mini-batches of data, 1000 samples at a time, to train the classifier (as the SGDC supports online learning, this is not a problem) until reaching convergence.

The loss function chosen for the experiments is the Modified Huber Loss function, also from scikit-learn. This function is less sensitive to outliers than other functions, but proves useful for us for another reason: with it, it is possible to estimate the probability of a label being positive in the test set. With that information at hand when rearranging a permutation, we can choose to apply on it the operation the algorithm is most confident will reduce the distance to the identity.

3.3 Workflow

Having described the core goals and methods of this project, we move on to the pipeline of actions.

First, we train the classifier using the data described in Section 3.1, thus creating a classifier model. Then, we feed that model with an unseen permutation of arbitrary size. The model then increases a counter of rounds by one, applies every possible operation on that permutation, calculates all 30 features for the results, and decides for the one with highest probability of reducing the distance to the identity. If the result is not the identity, we feed this new permutation to the model again; if it is the identity, then the permutation is sorted and we record the number of rounds needed as the result. A graphic representation of the workflow can be seen in Figure 4.

To illustrate the complete workflow, take the permutation $\pi = (10\ 3\ 4\ 5\ 9\ 8\ 6\ 7\ 1\ 2)$ of size 10, for example. First, we take all 210 operations which can be applied on the

Table 1: Description of the characteristics considered when constructing the feature vector, where b is the index of the feature in F.

b	**Description**
1	number of reversal breakpoints
2	number of transposition breakpoints
3	1 + 2
4	length of biggest non-singleton strip
5	number of non-singleton strips
6	sum of the lengths of all non-singleton strips
7	number of non-singleton increasing strips
8	number of decreasing strips
9	sum of the lengths of all non-singleton increasing strips
10	sum of the lengths of all decreasing strips
11	number of elements in the correct position ($\pi_i = i$)
12	number of fixed elements ($\pi_i = i$, every $\pi_j < i$ is to the left of π_i and every $\pi_j > i$ is to the right of π_i)
13	number of inversions
14	length of correct prefix
15	length of correct suffix
16	length of LIS
17	length of LDS
18	entropy
19	size of left plateau
20	size of right plateau
21	number of cycles
22	number of odd cycles
23	number of short cycles
24	number of divergent even cycles
25	number of oriented cycles
26	number of unitary cycles
27	length of biggest cycle
28	number of cycle components
29	number of edges in biggest component
30	number of cycles in biggest component

Table 2: Final feature vector calculations for permutations $\pi = (10\ 3\ 4\ 5\ 9\ 8\ 6\ 7\ 1\ 2)$ and $\pi^m = (10\ 9\ 8\ 3\ 4\ 5\ 6\ 7\ 1\ 2)$.

b	Features for π	Features for π^m	**Final Feature Vector**
1	6	4	-0.18
2	7	6	-0.09
3	13	10	-0.14
4	3	5	0.20
5	4	3	-0.20
6	9	10	0.10
7	3	2	-0.20
8	1	1	0.00
9	7	7	0.00
10	2	3	0.10
11	0	0	0.00
12	0	0	0.00
13	28	34	0.13
14	0	0	0.00
15	0	0	0.00
16	5	5	0.00
17	5	5	0.00
18	36	42	0.12
19	4	4	0.00
20	5	4	-0.11
21	7	8	0.09
22	5	7	0.18
23	6	7	0.09
24	0	0	0.00
25	1	1	0.00
26	5	7	0.18
27	4	4	0.00
28	6	8	0.18
29	6	4	-0.18
30	2	1	-0.20

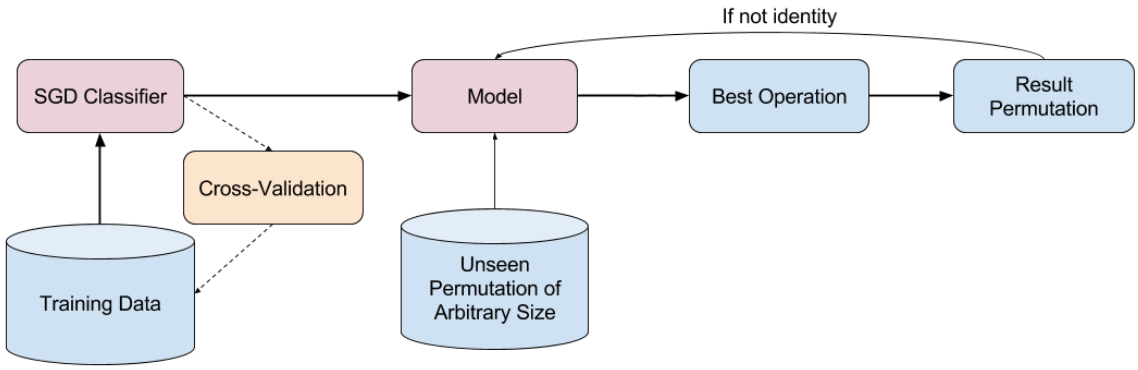

Figure 4: Workflow of the project

permutation, and generate the resulting permutations π^m, with $1 \leq m \leq 210$. Then, for each one of those, we calculate the feature vector f. All these vectors are fed to the classifier, which chooses the one it considers best: the one it is most confident decreases the distance to the identity.

Let us say, for example, it chooses operation $p_t(2, 5, 7)$, generating $p_t(2, 5, 7) \cdot \pi =$ (10 $\underline{9\ 8}$ $\underline{3\ 4\ 5}$ 6 7 1 2) and increasing the round counter by one. The algorithm checks if the permutation is sorted: as it clearly is not, it follows the same procedures on this new permutation as it did on the last (generate every possible result, their feature vectors and choose the best).

This time, it chooses operation $p_r(1, 10)$, reversing the whole permutation and generating ($\underline{2\ 1\ 7\ 6\ 5\ 4\ 3\ 8\ 9\ 10}$). Next, it chooses operation $p_t(3, 7)$, in which point we have permutation (2 1 $\underline{3\ 4\ 5\ 6\ 7}$ 8 9 10). Finally, it chooses operation $p_t(1, 2)$, returning the identity and finishing the sorting process after four rounds.

3.4 Evaluation

A set T of test permutations was created with permutations of size n, where n are all multiples of 10 between 10 and 100, inclusive. For each n where $10 \leq n \leq 50$, $10,000$ permutations were generated; for $n > 50$, we have $1,000$ different permutations for each n, for performance reasons. Also, every item in T consists of a permutation of ι, where $\frac{n}{10}$ reversals and $\frac{n}{10}$ transpositions were randomly applied. This way, we know that every permutation can be sorted with at most $\frac{2n}{10}$ operations (upper bound).

For each n, we compare our algorithm to six others in literature, regarding three different metrics:

1. Average Distance: the average number of operations necessary to sort the permutations.

2. Champion Percentage: percentage of permutations in which this algorithm returns the best result when compared to the other six algorithms.

3. Maximum Approximation Factor: the lower bound on the distance for unsigned permutations is $l_b = \frac{\text{number of breakpoints}}{3}$, as a transposition may remove up to 3 breakpoints. The Approximation Factor of an algorithm over a permutation is defined as the distance returned by the algorithm, divided by l_b. The Maximum Approximation Factor is the maximum among those values, for a fixed n.

The other six sorting algorithms considered for this study are some of the most popular in literature, with many different approaches and approximation factors.

The first is the greedy *Breakpoints Removal* approach, denoted **BP** [17]. This algorithm usually returns good results, and consists of greedily removing the greatest number of breakpoints possible each iteration, without generating new ones, increasing the length of some strip in the permutation. This heuristic has been proved to be a 3-approximation, at most.

Another approach was proposed by Dias et al. [12], from now on denoted **Dias**, which uses the cycle graph of the permutation. It first tries to increase the number of odd cycles, then the number of even ones, and when that cannot be done anymore with just one operation, it uses a bucket of configurations to decide which path to follow. It is a robust 3-approximation for unsigned permutations that rarely returns results that high.

The third algorithm we use for comparisons was designed by Rahman et al. [18] (denoted **Rahman** from now on), which also uses the cycle decomposition of the permutation. It has a $2k$-approximation factor for unsigned permutations, where k is the approximation factor for the algorithm used to generate the cycle graph.

Other important algorithm in literature is the one designed by Tesler [20], who created the program **GRIMM** to sort signed permutations by reversals (as we use both reversals and transpositions, an algorithm that uses only reversals creates valid solutions). As we are working with unsigned permutations only, we use Lin and Jiang's [16] solution for cycle decomposition to generate the equivalent signed permutations to be fed to **GRIMM**.

As **GRIMM** sorts by reversals, it would be fair to also test an algorithm that sorts exclusively by transpositions. The chosen algorithm is the 1.5-approximation (to the optimal solution that uses only transpositions) designed by Bafna and Pevzner [3], from here on denoted **TRANS**, presented in Section 1.

The last algorithm considered for comparisons also sorts by both reversals and transpositions, and is the one designed by Walter, Dias and Meidanis [21]: a 3-approximation for unsigned permutations that focuses on removing breakpoints. This algorithm will be denoted as **Walter**.

4 Results

While training the classifier with the data described in Section 3.1, it was observed that the algorithm reaches convergence relatively fast: just three or four iterations are sufficient for it to reach an $F\text{-}Score = 0.82 \pm 0.02$.

The following subsections elaborate on the results for the metrics described in Section 3.4. When comparing the machine learning approach described in this document to other algorithms, we address it as the **ML** approach.

4.1 Average Distance

The resulting average distance can be observed in Figure 5. As we increase the size of the permutations, it is clear that three algorithms (**TRANS**, **Walter** and **GRIMM**) do not scale well and get separated from the others (not surprisingly, as **TRANS** and **GRIMM** are limited to using just one kind of operation, they do not perform as well as the others).

The proposed **ML** algorithm shows to be competitive on smaller permutations, as it is the closest to the upper bound value of 4 when $n = 20$: **ML** shows an Average Distance of 4.005, while **Dias** in second place produces 4.039.

As n grows bigger, though, **Dias** and **BP** algorithms outperform **ML**. We believe this performance may be related to the fact that our classifier was trained on small permutations, and some of the features it considered relevant do not scale well. Still, it is a competitive algorithm: when $n = 100$ (and the upper bound distance is 20), **Dias** shows an Average Distance of 20.72, **BP** 21.86 and **ML** comes third with 23.11.

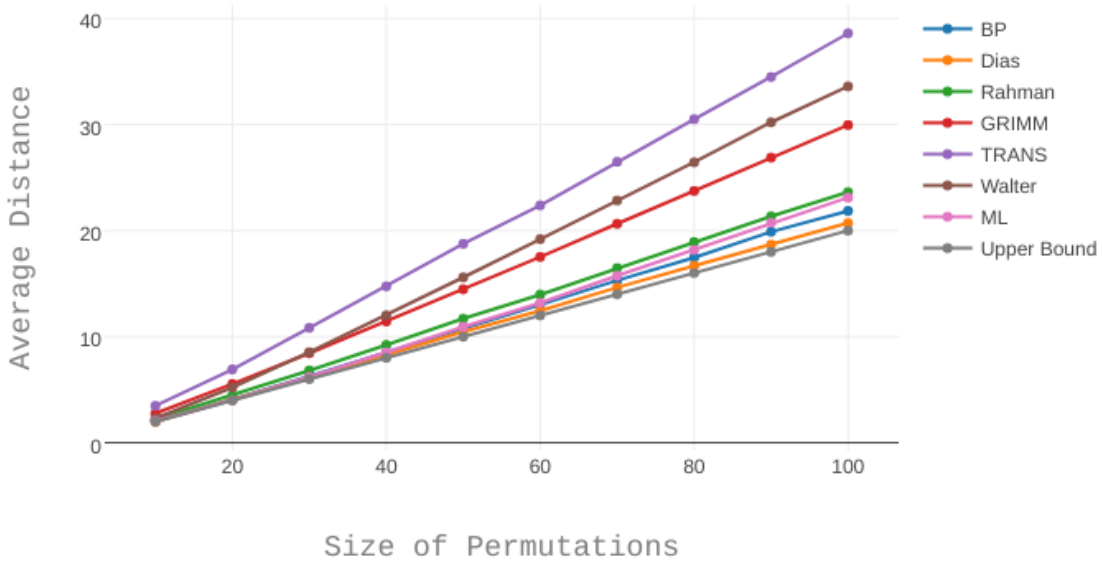

Figure 5: Average Distance returned by each algorithm for different permutations sizes. An extra "Upper Bound" line was added in gray, denoting the upper bound $\frac{2n}{10}$ on the distance.

4.2 Champion Percentage

The Champion Percentage results can be seen in Figure 6. Again, algorithms **TRANS** and **GRIMM** are quickly outperformed by the others (probably due to their operations

limitations). **Walter** shows promise for small permutations, but is also quickly subdued by the others.

As for Average Distance, **ML** shows the best performance among all when $n = 20$, being the best sorting algorithm for 83.85% of the permutations (**Dias** comes in second place with 81.18%).

As n grows, however, **Dias** completely outperforms the others. This shows that, for a fixed n, **Dias** is the most robust algorithm for every kind of permutation, while the others may be good at some specific types. **ML** algorithm finishes up in third again, being the best sorting algorithm for 13% of the permutations when $n = 100$.

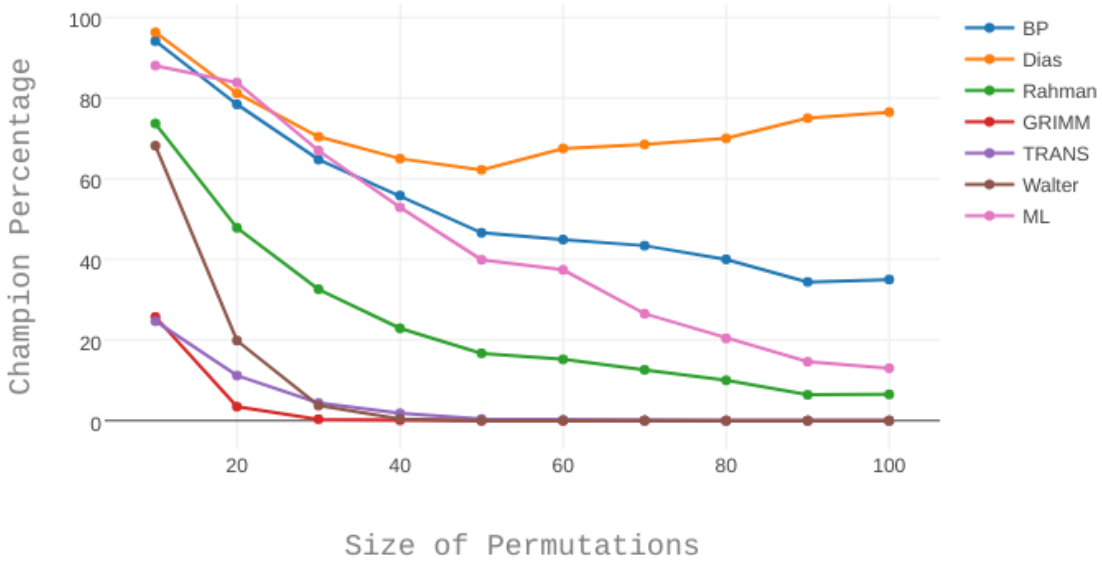

Figure 6: Champion Percentage values for each algorithm for different permutation sizes. When two algorithms reach the same best result, they both count as champions (for that reason, the Champion Percentage may sum up to more than 100% for some permutation sizes).

4.3 Maximum Approximation Factor

The Maximum Approximation Factor line chart can be seen in Figure 7. It is clear that the **TRANS** algorithm has problems with some specific permutations, which make its Maximum Approximation Factor explode to values as high as 9 when $n = 20$. That makes the visualization of other algorithms harder; for that reason, another plot omitting the **TRANS** algorithm can be seen in Figure 8.

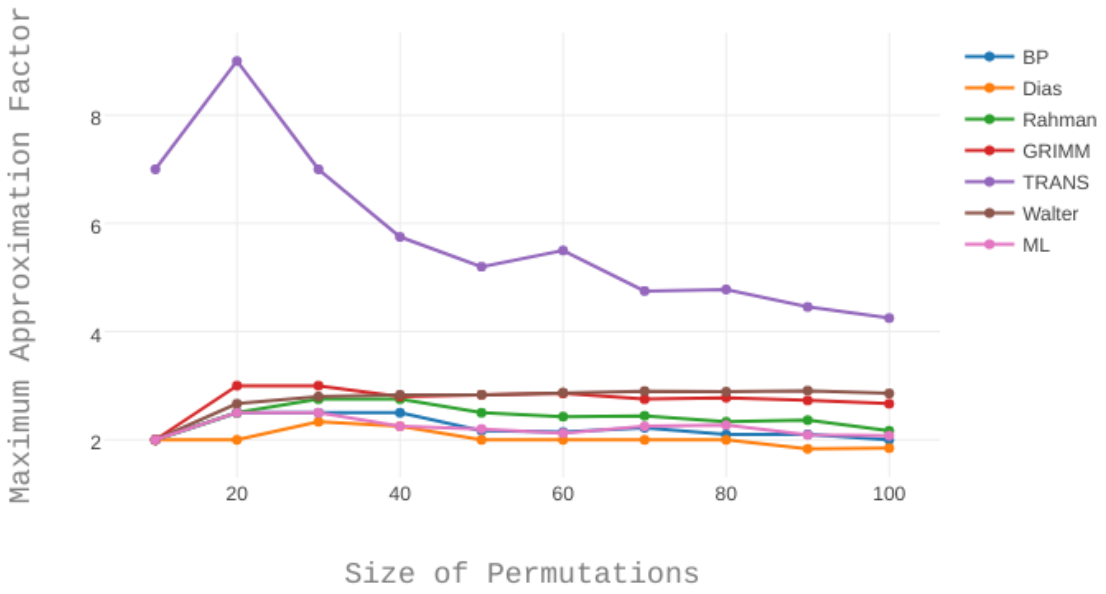

Figure 7: Maximum Approximation Factor for each algorithm for different permutations sizes.

In this metric **TRANS**, **Walter** and **GRIMM** also show the worst performances. While **Dias** shows the best performance overall, **ML** actually ties up with it when $n = 40$, with a Maximum Approximation of 2.25.

In this metric **ML** performs well, being comparable to **BP** on most permutation sizes. This shows that while **ML** may not be the best sorting algorithm when analyzing Champion Percentage, it is still competitive on Maximum Approximation, i.e., it appears to be a *versatile* algorithm, which performs well for all permutations of a fixed n size, and is not easier to be tricked by specific permutations than any other algorithm.

5 Conclusions

In this study we propose a machine learning approach to sorting permutations using reversals and transpositions. We train a classifier with 30 different features extracted from small permutations and extrapolate the algorithm to sort bigger permutations, and compare the results with many different algorithms in literature.

Our algorithm shows good results for small permutations, but is outperformed by others when considering bigger permutations. We believe this behavior is due to the classifier's training: it was trained with small permutations' features, hoping they would scale well for bigger permutations, but apparently the model gets confused at some point.

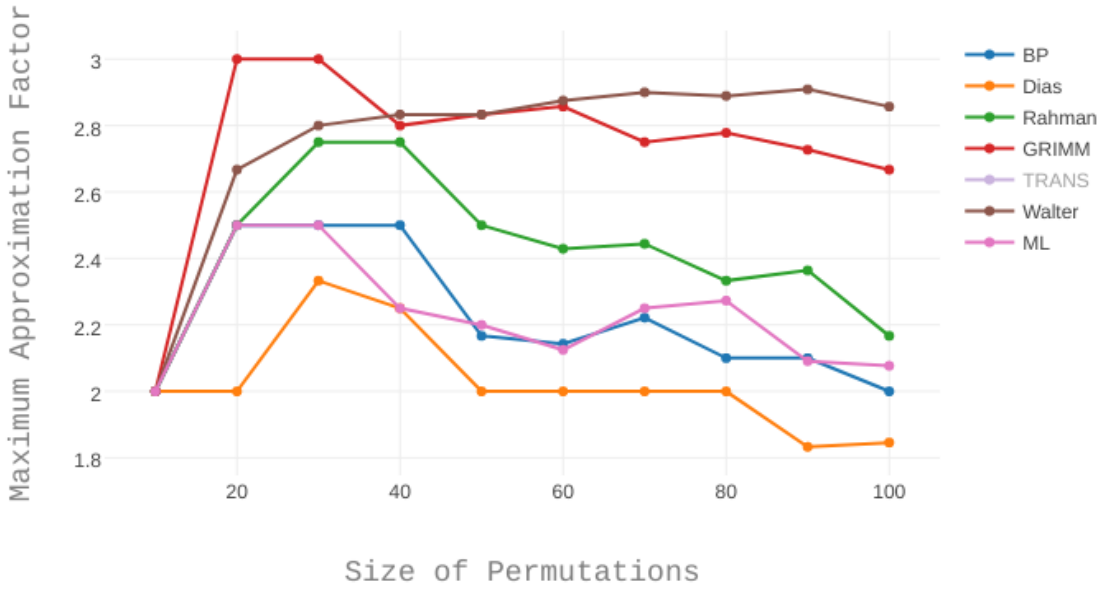

Figure 8: Maximum Approximation Factor for each algorithm for different permutations sizes, with **TRANS** algorithm result removed for better visualization

.

Future work may be focused on trying to use some bigger permutations to train the classifier, and observe if that shows better results when scaling.

References

[1] D. A. Bader, B. M. 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] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

[3] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, 1998.

[4] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-approximation algorithm for sorting by reversals. *In 10th Annual European Symposium on Algorithms (ESA'2002)*, pages 401–408, 2002.

[5] P. Berman and M. Karpinski. On some tighter inapproximability results. In *Proceedings of the 26th International Colloquium on Automata, Languages, and Programming (ICALP'1999)*, volume 1644, pages 200–209. Springer, 1999.

[6] L. Bulteau, G. Fertin, and I. Rusu. Sorting by transpositions is difficult. *SIAM Journal on Discrete Mathematics*, 26(3):1148–1180, 2012.

[7] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the 1st Annual International Conference on Computational Molecular Biology (RECOMB'1997)*, pages 75–83. ACM, 1997.

[8] A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, 1999.

[9] X. Chen. On sorting unsigned permutations by double-cut-and-joins. *Journal of Combinatorial Optimization*, pages 1–13, 2013.

[10] D. A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *Proceedings of the 9th annual ACM-SIAM Symposium on Discrete Algorithms (SODA'1998)*, pages 244–252, 1998.

[11] D. A. Christie. *Genome rearrangement problems*. PhD thesis, University of Glasgow, 1998.

[12] U. Dias, A. R. Oliveira, and Z. Dias. An improved algorithm for the sorting by reversals and transpositions problem. In *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM BCB'2014)*, pages 400–409. ACM, 2014.

[13] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 3(4):369–379, 2006.

[14] S. Hannenhalli and P. A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of 36th Annual Symposium on Foundations of Computer Science (FCS'1995)*, pages 581–592. IEEE, 1995.

[15] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1):180–210, 1995.

[16] G. Lin and T. Jiang. A further improved approximation algorithm for breakpoint graph decomposition. *Journal of Combinatorial Optimization*, 8(2):183–194, 2004.

[17] A. R. Oliveira. O Problema da Ordenação de Permutações por Reversões e Transposições. Master's thesis, University of Campinas, 2015.

[18] A. Rahman, S. Shatabda, and M. Hasan. An approximation algorithm for sorting by reversals and transpositions. *Journal of Discrete Algorithms*, 6(3):449–457, 2008.

[19] E. Tannier, A. Bergeron, and M.-F. Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155(6):881–888, 2007.

[20] G. Tesler. Grimm: genome rearrangements web server. *Bioinformatics*, 18(3):492–493, 2002.

[21] M. E. M. Walter, Z. Dias, and J. Meidanis. Reversal and transposition distance of linear chromosomes. In *String Processing and Information Retrieval (SPIRE'1998)*, pages 96–102. IEEE, 1998.