EFFICIENT SOLUTION TO THE PROBLEM GENOME SORTING BY REVERSALS

A Thesis

Submitted in partial fulfilment of the requirements for the award of the Degree of

MASTER OF TECHNOLOGY

IN

COMPUTER SCIENCE & ENGINEERING

BY

BEETHIKA TRIPATHI

MT/CS/10017/2013

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

BIRLA INSTITUTE OF TECHNOLOGY

MESRA-835215, RANCHI

2015

DECLARATION CERTIFICATE

This is to certify that the work presented in the thesis entitled **“Efficient Solution to the Problem Genome Sorting by Reversals”** in partial fulfilment of the requirement for the award of Degree of **Master of Technology in Computer Science and Engineering** of Birla Institute of Technology Mesra, Ranchi is an authentic work carried out under my supervision and guidance.

To the best of my knowledge, the project report embodies result of original work and activities carried out by student and the content of this thesis does not form a basis for the award of any previous degree to anyone else.

Date:

Dr. Amritanjali
Dept. of Computer Science & Engineering
Birla Institute of Technology
Mesra, Ranchi

Head
Dept. of Computer Sc & Engineering
Birla Institute of Technology
Mesra, Ranchi

Dean
(Post Graduate Studies)
Birla Institute of Technology
Mesra, Ranchi

CERTIFICATE OF APPROVAL

The foregoing thesis entitled "**Efficient Solution to the Problem Genome Sorting by Reversals",** is hereby approved as a creditable study of research topic and has been presented in satisfactory manner to warrant its acceptance as prerequisite to the degree for which it has been submitted.

It is understood that by this approval, the undersigned do not necessarily endorse any conclusion drawn or opinion expressed therein, but approve the thesis for the purpose for which it is submitted.

(Internal Examiner) **(External Examiner)**

(Chairman)
Head of the Department

ACKNOWLEDGEMENT

The successful completion of thesis is like a golden feather for any student. It was next to impossible to finish this project without the guidance and support of few people whom I would like to extend my gratitude.

I would like to express my deepest appreciation and sincere gratitude to my thesis guide, **Dr. Amritanjali**, Department of Computer Science and Engineering, Birla Institute of Technology, Mesra, Ranchi, for her expert guidance constant encouragement, invaluable direction and meticulous attention during my thesis work. Her wide knowledge and logical way of thinking have been of great value for me. Her understanding, encouraging and personal guidance have provided a good basis for present thesis. Always active, she would come up with innovative ideas to steer our work in the right direction. Her enthusiasm was a source of inspiration to me. Working with her was a memorable experience.

I am also thankful to **Dr. Sandip Dutta**, Head of the Department of Computer Science and Engineering, Birla Institute of Technology, Mesra, Ranchi for supporting us with our work. He has always been a constant source of motivation and encouragement to me.

Many of my friends have helped me in different ways during the course of this work. While some of them provided help with the technical aspects of the work, others helped me with their ideas that improved my understanding of the scope of my work. I am thankful to all of my friends.

Last but not least, I would like to state that I am forever indebted to my parents, who were a constant source of inspiration for me.

BEETHIKA TRIPATHI

Date: MT/CS/10017/2013

ABSTRACT

Genetic rearrangement over a period of time leads to evolution. Reversal is one of the most commonly observed evolutionary events. In this a segment of a chromosome gets flipped, thereby reversing the order and orientation of the genes in that segment. By comparing the shared gene order in two genomes that have evolved by reversals events, their evolutionary scenario is reconstructed. Sorting by reversals is defined as the problem of finding the minimal sequence of reversals that can transform the shared gene order of one of the given genome into that of another. Till now lots of study has been done on finding the reversal distance (minimum number of reversals) between two given genomes and generating corresponding sequence of reversals. However, for most of the genomes there can be several such sequences and the solution space increases exponentially with reversal distance and number of shared genes. Each sequence describes a probable evolutionary scenario. The sequence closest to actual scenario can be deduced by applying some biological constraints. Very large number of possible sequences makes their analysis difficult and listing them requires lots of memory and time. Application of biological constraints like conserved intervals can help to discard reversals that are less probable, reducing the solution space. This constraint is applied to generate solutions that are preserving the conserved intervals. Solutions are grouped into equivalence classes for compact representation. Reduction of solution space not only decreases computation time but also makes their analysis easier.

Genomic sequences are sorted using breadth first approach and depth first approach. On comparing results obtained proves the effectiveness of the later approach over the former one. After applying the biological constraints the results are again compared which shows the significant amount of reduction in solution spaces. Moreover, depth first approach requires less memory and is easier to parallelize.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

1.1. INTRODUCTION

This work is a part of comparative genomics and bioinformatics which addresses the problem of genome rearrangement in molecular evolution. Genetic rearrangement over a period of time leads to evolution. All species have said to be evolved from the common ancestor. When a genome replicates into another genome there can be some change or error this leads to mutation and formation of a new species.

Mutations can be due to changes in the external factors like climate, shift in the geographic locations, or it can be due to changes in the internal factors like reaction with certain types of chemicals. These mutations over a course of time lead to evolution. This process results in diversity at the level of species, organisms or molecules.

There can be enormous amount of possible mutations that can convert a genomic sequence into other genomic sequence. Studying these mutations can reveal the actual scenario that might have happened during a period of time.

Comparative genomics is a field of biological research in which various genomic features like genes, gene order, DNA sequence and other genomic structures are compared. Mainly the common features of two organisms will be encoded within a DNA so certain commonality is established between the two and the study is carried out. Bioinformatics is an interdisciplinary field and uses computational power to solve biological problems. Though it is difficult for a computer to adopt human intuitiveness however computers are good at performing calculative tasks. So fast and accurate computations and human intuitiveness can be used to resolve various biological problems.

In this thesis the work mainly focuses on how evolution takes place and it finds all the possible mutations that might convert a given genome into another genome. There are mainly two types of models that are used for comparing two genomes to analyze their evolutionary relationship:

Gene Sequence based Model: Here point (gene-level) mutations that are responsible for change in the nucleotide sequence of genes, e.g. insertion, deletion, substitution.

Gene Order based Model: Here large-scale (genome-level) mutations are studied those are responsible for changes in the order of genomic markers (single gene or group of genes) in the genome, e.g. reversals, translocations, transpositions, duplication, deletion, fusion and fission.

Gene order based model is developed over the last few years as more and more DNA sequence data of complete genomes has become available. Moreover Genome-level mutations occur slowly than gene-level mutations. Closely related species have essentially the same genes however the gene order may be different. Comparing nucleotide sequences of genes provided no evolutionary information that is why Gene Order based model is preferred over Gene Sequence based model.

Study of genome rearrangements is done as it helps in understanding how the genomes have undergone changes during the course of evolution. The chromosomes got broken apart and juggled around by some rearrangement operations during evolution. Given two genomes the evolutionary scenario can be reconstructed by transforming the gene order of one genome (source) into that of another genome (target) by performing different types of rearrangement operations. As rearrangements are rare events, so scenarios that minimize their number are more likely to be close to reality (Parsimony Hypothesis-the best scenario is the one that requires the fewest evolutionary changes).

Various rearrangement operations are reversal, transposition, translocation, fission and fusion. Reversal is amongst the most frequently occurring genomic rearrangement operation that has been observed amongst closely related species [1]. There can be many sequences through which a genome can be transferred into the other however the most optimum solutions need to be established.

The problem of sorting by reversals [1] is a well-known and a hot topic of research for more than a decade. It is used for describing the evolutionary scenario of a chromosome in two species that have evolved from a common ancestor by reversals rearrangement operations.

There are many techniques in which the problem can be approached and many software tools are available like GRIMM [2-3], SoRT^2 [4], baobabLuna [5], UniMoG [6] etc to sort permutations using reversals but several limitation are associated with them. For example GRIMM, SoRT^2 provides just a single solution that converts the given source genome into the target genome which is useless as this might be the actual scenario but chances are very rare.

Therefore all possible solutions must be targeted though baobabLuna which finds out all the possible solutions but fails in dealing with all types of permutations there are certain peculiar structures like ‘hurdles’ and ‘fortress’ when present in the input permutation then it fails. All these tools were implemented based on Hannenhalli and Pevzner (HP) model [1]. There is another tool UniMoG which is based on Double Cut and Join model [7] but it again fails in enumerating all possible evolutionary scenario.

In this work all the possible evolutionary scenarios are listed efficiently even in the presence of hurdles and fortress. Moreover certain heuristics are applied which could biologically limit the solutions that are not possible at all. Thus reduces the solutions and makes it easier for the biologists to carry out their analysis.

1.2. AIMS AND OBJECTIVES

Aims: Finding efficient and complete solution to the problem of Genome Sorting by Reversals. Underlying objectives of thesis are enumerated as follows:

- Improving efficiency with respect to time and memory.
- Enumerating all solutions in the presence of hurdles and fortress.
- Listing solutions closer to actual evolutionary scenario by filtering out less probable solutions.

1.3. APPLICATIONS

- This could help in study of various diseases that occurs due to mutation in genomic sequence like progeria.
- This helps in construction of phylogenetic tree which could further help in the determination of evolution history.
- The reasons for various types of mutations can be predicted.
- The stage of the disease can be predicted.
- It can help in the identification of new genes.
- Study the mutations that lead to origin of a disease.

1.4. ORGANIZATION OF THE THESIS

Chapter 1 is the introduction and provides a background of this thesis. Along with it the similar types of available tools and their shortcomings, aims and objective, application of this work and the organization of the thesis is also discussed. Chapter 2 gives a literature survey and a brief discussion of the algorithms given by various authors. Chapter 3 provides an overview of the research methodology. It covers the details of various concepts used for the thesis work. Proposed algorithms have been explained in detail with the help of examples. Chapter 4 provides the implementation details of the software package. The result of testing the proposed methodology is described in chapter 5. Chapter 6 states the conclusions drawn from this work. Chapter 7 deals with the future scope of the work and suggests possible directions for further research.

CHAPTER 2
LITERATURE REVIEW

Unichromosomal genomes without duplication of genes are considered in our study. Genomes are represented by the list of homologous markers (usually genes or blocks of contiguous genes) between them, as shown below:

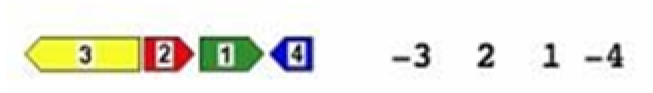


Figure 2.1: Representation of genome using homologous markers.

2.1. SORTING BY REVERSALS

When the permutation is unsigned the sorting by reversal problem is NP-hard [8-9]. In 1995 Hannenhalli and Pevzner for the first time gave a proof of solving it in polynomial time for the signed versions of the problem. Since then, many solutions were given with speed and memory utilization improvements.

Signed integers are used to represent the markers where sign denotes the orientation of markers on one genome with respect to the other genome. The signed permutation $\pi = (\pi_1 \ldots \pi_n)$ of size n is used for representation where it can take values from $-n$ to n without repetition of absolute value of integer. Here π_i represents the element at position i in π. π_T represents the identity permutation (1… n). A reversal ρ on an interval [i,j] of a permutation π where $1 \leq i \leq j \leq n$, is defined as $\rho = \{|\pi_i|, |\pi_{i+1}|, \ldots, |\pi_{j-1}|, |\pi_j|\}$,in sorted order. The reversal operation is denoted by $\pi \circ \rho$ where it reverses the order and flips the signs of the elements of ρ. The operation is represented as:

$$\pi \circ \rho = (\pi_1\ \pi_2 \ldots \pi_{i-1} -\pi_j -\pi_{j-1} \ldots -\pi_{i+1} -\pi_i\ \pi_{j+1} \ldots \pi_{n-1}\ \pi_n)$$

For example $\pi = \{1\ 4\ {-3}\ 2\ {-5}\}$, $\rho = \{2,3,4\}$ then $\pi \circ \rho = \{1\ {-2}\ 3\ {-4}\ {-5}\}$.

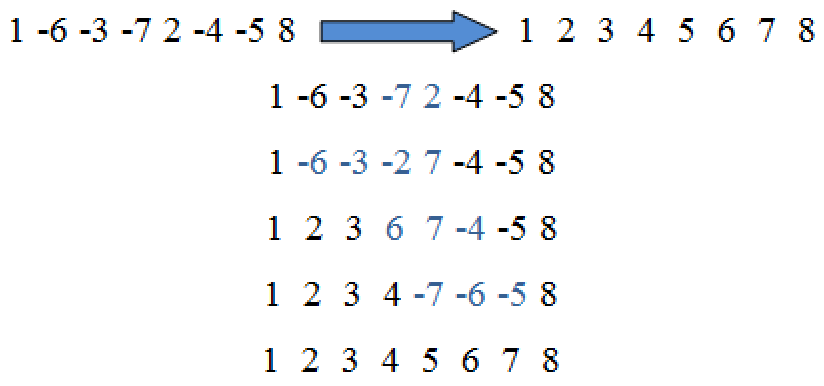


Figure 2.2: Sorting genomic permutation using reversals as the rearrangement operation. Highlighted portion shows the reversals applied on the particular permutation.

A sequence or i−sequence of reversals is represented as $\rho_1\rho_2...\rho_i$ is valid sequence for a permutation π, then $\pi \circ \rho_1,\rho_2,...\rho_i$ denotes the successive application of the reversals ρ_1, ρ_2,... ρ_i in the order in which they appear. An i−sequence of reversals $\rho_1\rho_2...\rho_i$ sorts a permutation π into a permutation π' if $\pi \circ \rho_1\rho_2...\rho_i = \pi'$. The length of the shortest sequence of reversals that sorts a permutation π into π_T is called the reversal distance and is denoted by $d(\pi)$ which is calculated from the breakpoint graph [10]. Breakpoint Graph is a graph constructed from the source and target permutations which is used for finding the reversal distance and an optimal sequence of reversals for sorting the source permutation.

2.2. SOLUTION SPACE OF SORTING BY REVERSALS

An optimal i-sequence is represented as $s = \rho_1\rho_2...\rho_i$ (a valid i-sequence of reversals for π), if $d(\pi \circ s) = d(\pi)$-i. s is called an optimal sorting sequence for π and π_T when $i = d(\pi)$. For example s = {1}{2}{1,2,3} is an optimal 3 sequence for π={-3 2 1 -4}.

Siepel [11] proposed an algorithm that listed all sorting reversals (ASR) in $O(n^3)$ time complexity. Using his algorithm those reversals are computed that will bring the given permutation one step closer to the target. All such 1-sequences are listed. Then each of them are applied to the original permutation to form the new set permutations and each of those permutations have reversal distance $d(\pi)$-1. Now new set of reversals are computed for each permutation. This 1-sequence when combined with the predecessor ρ optimal 2-sequence are generated. In this way the algorithm is repeated till $d(\pi)$-sequences are generated reducing the distance to zero and each of them sort permutation π to π_T. For example {1}{2}{1,2,3}{4} is one such solution that sorts π=(-3 2 1 -4) to π_T=(1 2 3 4).

Later an improved version of the aforementioned algorithm was proposed by Swenson *et al.*, whose average-case time complexity was $O(n^2)$ [12]. The solutions generated were huge in number and keeps on increasing as the size of permutation and the reversal distance increases. It was experimentally found that, the permutation (−4 −11 6 −9 −2 1 −8 3 −10 7 −5) has 6345019 solutions. Though all the solutions could be easily obtained by this algorithm but it requires lot of time as well as memory to compute and store them.

The concept of traces was given by Bergeron *et al.* [13] by grouping the similar sequences into equivalence class and representing them using normal form of traces. Two sequences are said to be equivalent if one can be obtained from another by sequence of commutations of non-overlapping reversals. For example, the sequences of reversals (words) {2} { 2, 3 ,4} { 3, 4, 5} and { 2, 3, 4} {2} {3, 4, 5} are equivalent because the reversals {2} and {2, 3 ,4} commute. As opposed to it none of these sequences of reversals are equivalent to {2} {3, 4, 5} {2, 3, 4} because the reversals {2,3,4} and {3,4,5} overlap.

An equivalence class of optimal sequences of reversals under this equivalence relation is called trace. For any trace there is a unique representation called the normal form. This is done by finding out the commuting pairs, these are the non-overlapping pairs or one set completely contained in another, their positions can be interchanged and belong to same equivalence class these when arranged in lexicographic order forms 1 sub-word of the normal form of trace. The other overlapping pairs are considered as another sub-word. All such sub-words are found out, these sub-words when arranged in lexicographic order forms the normal form of trace representing the equivalence class. For example for the permutation π={1 4 -3 2 -5} there are two normal form traces possible {2}{2,3,4}{4}{5} and {2,3,5}{3} > {2,4,5} > {3,4,5}. In first trace all {2}, {2,3,4}, {4} and {5} are commuting so position of any of them could be interchanged to give 24 different solutions. In the second trace {2,3,5},{2,4,5} and {3,4,5} overlap with each other resulting in three sub-words. Only reversal {3} commutes with all other reversals, so by interchanging {3} with others we get 4 solutions for this trace. In short total 28 solutions are represented just by two traces greatly reducing the solution set.

Bergeron though gave the concept however, no algorithm was given. Braga *et al.* [14-15], combined the algorithm of Siepel and with the concept of traces. By using the normal form representation, it is capable of enumerating the set of all traces that represents all possible solutions.

Baudet et al. [16] used the approach of exploring the solution space in depth first manner using stack and listed normal forms of traces to represent classes of sorting sequences this drastically reduced the memory usage. But their algorithm could not find the total number of solutions. Amritanjali et al. [17] implemented the depth first approach using recursion. However, the implementation till now failed to handle input permutations with hurdles.

For most of the genomes there can be several sequences and the solution space increases exponentially with reversal distance and number of shared genes. Very large number of possible sequences makes their analysis difficult and listing them requires lots of memory and time. Application of biological constraints [18-22] like common intervals can help to discard reversals that are less probable, reducing the solution space. This constraint has been applied to generate solutions that are preserving the conserved segments. Now each sequence describes a probable evolutionary scenario close to actual scenario.

CHAPTER 3
RESEARCH METHODOLOGY

3.1. PRELIMINARIES

Breakpoint graph is constructed to find out all the possible sequences that can sort the source permutation to the target permutation. Edges in the breakpoint graph show the adjacencies in the source and target permutation. Gray edge in the graph connects vertices corresponding to adjacent genes in target and black edge connects vertices corresponding to adjacent genes in source. The steps for constructing graph are as follows:

1. For each positive element +x form node 2x-1 and 2x and for each negative element -x form node 2x and 2x-1.
2. Append node 0 at the start and node 2n + 1 at the end.
3. Join node 2x and 2x+1 with a black edge and join with element value 2x and 2x+1 with a gray edge.

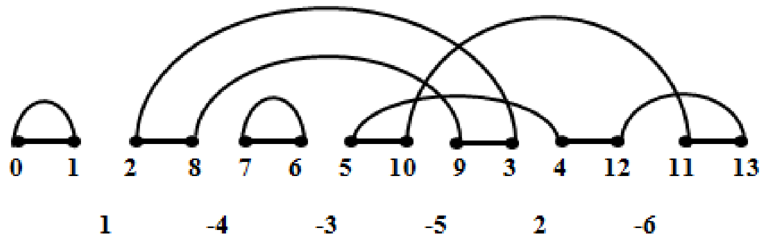


Figure 3.1: Breakpoint Graph.

Various structures are involved in the breakpoint graph that is used to find all possible pairs to perform reversals. **Cycle** is a sequence of connected nodes (n_0, n_1, n_2, …, n_{2x+2}), such that x ≥ 0, $n_{2x+2} = n_0$. It consists of alternate gray and black edges. A trivial cycle is a cycle with only one gray and black edge. In a cycle those Black edges having different directions are called **oriented pairs**. Such a cycle having at least one oriented pairs are called **oriented cycle**. When all the black edge of cycle have same direction then they are called **unoriented cycle**.

Those cycles whose gray edges overlap are referred to as **components**. The trivial component i.e. having trivial cycle is termed as **adjacencies**. **Oriented component** are those non-trivial component that have at least one oriented cycle. Rest all are **unoriented components**.

Reversal Distance is the minimum number of reversals that are required to transform source genome into target genome. The cut points for applying a reversal are taken on a pair of black edges of same cycle or different cycles. Performing a reversal operation can result in splitting a cycle, merging two cycles or no change in the cycle. Let Δc denote the change in the number of cycles after performing a rearrangement operation, then $\Delta c = \{-1, 0, +1\}$. Based on the value of Δc, a reversal operation is classified as **split** ($\Delta c = 1$), **neutral** ($\Delta c = 0$) or **joint** ($\Delta c = -1$).

An **optimal reversal** reduces the reversal distance by 1. The initial set of optimal reversals is called as 1-sequences of optimal reversal. The d-sequence of such optimal reversal is called as **sorting reversal** which sorts the source genome into the target genome in d steps. All such possible sequences form the **solution space**.

Each solution describes a probable evolutionary scenario. Biologists can delineate the actual evolutionary scenario by analyzing the solutions generated. However, the huge size of solution space makes their analysis difficult. The concept of traces is used to group the equivalent i-sequences of reversals, where $1 < i \leq d$. Two i-sequences are equivalent if one i-sequence can be transformed into another by exchanging commutative reversals. Two reversals are commutative when the reversal interval of one is completely contained in the other or they are disjoint. These reversals when arranged in lexicographic order forms 1 sub-word of the normal form of trace. The other overlapping pairs are considered as another sub-word. All such sub-words are found out, these sub-words when arranged in lexicographic order forms the **normal form of trace** representing the equivalence class. The final solution space consists of only d-traces. This results in compact representation of the solution space.

For example for the permutation $\pi=\{1\ 4\ -3\ 2\ -5\}$ there are two normal form traces possible {2}{2,3,4}{4}{5} and {2,3,5}{3} > {2,4,5} > {3,4,5}. In first trace all {2}, {2,3,4}, {4} and {5} are commuting so position of any of them could be interchanged to give 24 different solutions. In the second trace {2,3,5},{2,4,5} and {3,4,5} overlap with each other resulting in three sub-words. Only reversal {3} commutes with all other reversals, so by interchanging {3} with others we get 4 solutions for this trace. In short total 28 solutions are represented just by two traces greatly reducing the solution set.

3.2. ENUMERATING SOLUTION SPACE FOR PERMUTATIONS WITHOUT UNORIENTED COMPONENTS

On constructing the breakpoint graph of the permutation if it contains only oriented cycles then it can be sorted by applying split reversals. Split reversal is performed by reversing the elements between the oriented pair. This increases the number of cycle by one. Reversal distance is calculated with the help of breakpoint graph by counting total number of cycles (A modified version is described later).

It can be calculated as,

$$d(\pi) = (n+1) - c,$$

where n is number of genes in π and c is number of disjoint cycles. For example permutation, π = (1 -5 4 -3 2), n = 5 and c = 3, so d(π) = 6 - 3 = 3.

Thus each time number of cycle increases by one so distance reduces by 1 thus finally sorting the permutation when the distance reduces to 0.

For example, in the permutation π = (1 -5 4 -3 2 6), the pairs (-5, 4), (4, -3), (-5, 6) and (-3, 2) are oriented pairs. Reversing any of the segments (4), (2), (-5, 4, -3, 2) of π unite the corresponding oriented pairs and transforms them into adjacency.

The initial step of processing the permutation like generating breakpoint graph and calculating 1-sequences is shown below:

PROCESS_PERMUTATION procedure is given as follows:

1. Construct the breakpoint graph.
2. Calculate the reversal distance d(π).
3. Generate all the 1-sequence of reversals and store in lexicographic order in root node.

3.2.1. All Sorting Sequences

Tree structure is used to store the 1-sequenes where each node is composed of its elements containing a reversal ρ. The solutions are all the branches of the tree and are of d(π) length. This approach enumerates all the possible sequences that could sort the given permutation to an identity permutation. The complexity of the algorithm is $O(n^3)$ and produces up to $O(n^2)$ solutions. The algorithm is given below:

The **SEQUENCE_GENERATION** Procedure is given as follows:

```
1. Initialize root node using PROCESS_PERMUTATION.
2. Add all the reversals in the root node to the solution set.
3. For each i-sequence in the solution set
        a. For each reversal in the i-sequence
               Apply reversal on the permutation
        b. For the new permutation initialize next level node by calling
           PROCESS_PERMUTATION.
        c. For each reversal ρ in the node
            i.  Check the distance is reduced by 1 after applying ρ.
            ii. Append ρ to i-sequence to form (i+1)-sequence and store them in new
                solution set.
4. Solution set is replaced with new solution set.
5. Step 5 and 6 is repeated till the distance reduces to 0.
6. Solution set is the output.
```

The initial set of optimal reversals is called as 1-sequences of optimal reversal. When these reversals are applied to the source permutation we get new permutations that are one step closer to the target permutation. For each of these next 1-sequences of optimal reversal are computed using the same procedure. When these 1-sequences are concatenated to their corresponding preceding we get 2-sequences of optimal reversals. Finally, after d-iterations all the d-sequences of optimal reversals are obtained. They are the possible parsimonious sequences of reversals that can transform source permutation into the target permutation in least number of steps.

3.2.2. All Sorting Traces with count of number of sorting sequences in each trace

When all the sorting sequences are enumerated it is observed that many of the solutions are redundant with a slight variation in the positioning of the reversals. All such reversals were grouped in equivalence class and represented with the help of normal form trace. In order to show number of sequences that were similar count is used which counts the number of solutions in the equivalence class. The algorithm to enumerate all sorting traces with count of number of sorting sequences in each trace is given below:

The **TRACE_GENERATION** Procedure is given as follows:

1. Initialize root node using PROCESS_PERMUTATION.
2. Add all the reversals in the node to the trace set and initialize its count as 1.
3. For each i-trace in the trace set
    a. For each reversal in the i-trace
        Apply reversal on the permutation
    b. For the new permutation initialize next level node by calling PROCESS_PERMUTATION.
    c. For each reversal ρ in the node
        i. Check the distance is reduced by 1 after applying ρ.
        ii. Find out the position of the last overlapping sub-word in trace with ρ.
        iii. Append ρ at the next sub-word while maintaining the lexicographic ordering, if it was the last sub-word then append at the end by forming a new sub-word.
        iv. Now check if (i+1)-trace formed is already present in new trace set.
        v. If present then just increase the count of the trace otherwise add it to the new trace set.
4. Trace set is replaced with new trace set.
5. Step 5 and 6 is repeated till the distance reduces to 0.
6. Trace set is the output.

Each i-trace is formed and checked in the set of traces if there is a match found so the i-trace is merged with the already present one and the count of both the i-trace are added to form the new count. This is done till distance reduces to 0 at that stage all the unique traces with their count representing the total number of similar sequences that are present are outputted.

3.2.3. All Sorting Traces without counting number of sorting sequences in each trace

As there is not much significance of counting traces so the traces could be generated in a more efficient manner where only those branches are explored where new solutions could be found. For each equivalence class there will be only one trace in the normal form order. So, no need to compare with the other sorting traces in the solution space to check if the generated trace is a new one or not. The time spent in matching with other sorting traces is saved with the limitation that total number of sorting sequences cannot be counted. Only unique traces are explored this is possible by exploring only those branches in which the reversals append at the end of the normal form i-trace.

There are two approaches to solve this type of problem, Breadth First Approach and Depth First Approach. They are described below:

Using Breadth First Approach

The above approach (described in section 3.2.2) is modified by exploring only those traces in which the reversal appends at the end as this will always leads to the generation of new trace so computation time can be saved in matching the trace with the existing traces. This is done by checking if the last non-commuting sub-word is at the end of the trace then form a new sub-word with ρ, otherwise if the last non-commuting sub-word is at the second-last position in the trace and ρ has the highest lexicographic order in the last sub-word then append at the end of that sub-word.

The **TRACE_GENERATION** Procedure of BFA is given as follows:

```
1. Initialize root node using PROCESS_PERMUTATION.
2. Add all the reversals in the node to the trace set.
3. For each i-trace in the trace set
        a. For each reversal in the i-trace
               Apply reversal on the permutation
        b. For the new permutation initialize next level node by calling
           PROCESS_PERMUTATION.
        c. For each reversal ρ in the node
             i. Check the distance is reduced by 1 after applying ρ.
            ii. If ρ has to be appended at the end of trace (as described above) then do so.
                Otherwise, move to next ρ.
           iii. Add the trace to the new trace set.
4. Trace set is replaced with new trace set.
5. Step 5 and 6 is repeated till the distance reduces to 0.
```

6. Trace set is the output.

The way in which trace is generated makes BFA much faster than ever. As many of the unnecessary branch exploration is foregone. The pictorial representation of how BFA works is depicted below:

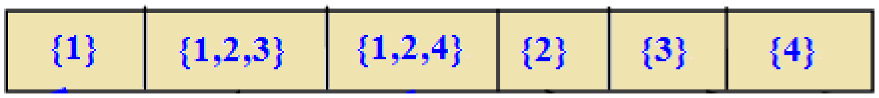


Figure 3.2 (a): Root node initialized with 1-sequences.

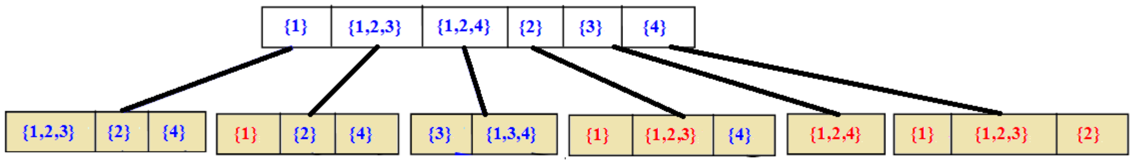


Figure 3.2 (b): Level 2 nodes initialized with 1-sequences after applying 1-sequence at level 1.

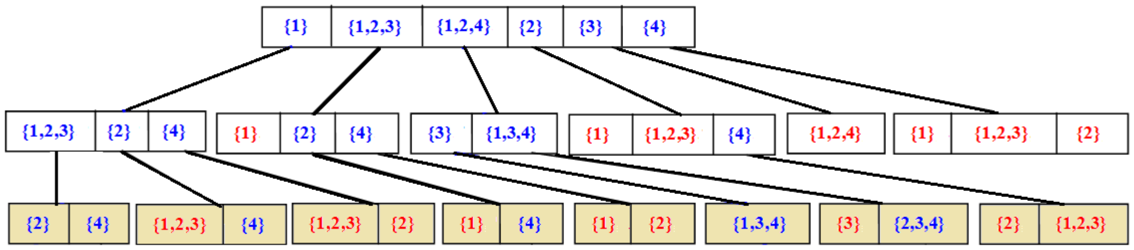


Figure 3.2 (c): Level 3 nodes initialized with 1-sequences after applying 1-sequence at level 1 & 2.

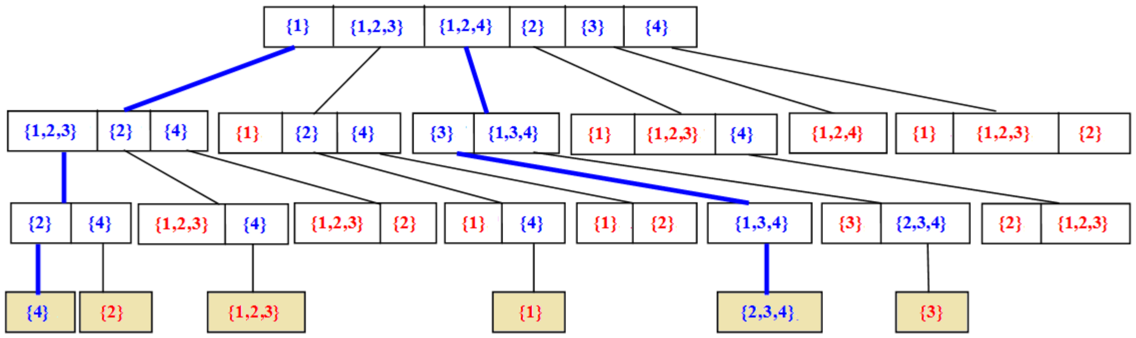


Figure 3.2 (d): Level 4 nodes initialized with 1-sequences after applying 1-sequence at level 1, 2 & 3

Figure 3.2: Above tree structure depicts how sorting takes place for permutation (-3, 2, 1, -4) using BFA and level-wise change in tree structure is shown. Highlighted level represents current level, nodes in red color represent branch terminator reversals and bold blue lines represent the trace that sorts the permutation.

Branch terminator reversals are those reversals which are optimal 1-sequences but, when combined with predecessor trace, lead to traces that were inserted in another branch of the tree, which means they are redundant so terminates further exploration.

Using Depth First Approach

In order to generate (i+1)-trace it is not necessary to calculate all i-trace. From here the concept of depth first approach came into existence, when the tree was processed in depth first manner instead of breadth first manner, this improved the memory usage.

All the 1-sequences are generated and are stored in lexicographic manner in the root node of the tree. Now take the first reversal apply it on the given permutation π and obtain new permutation π' and form 1-trace, again compute 1-sequences for it and store them in lexicographic order in the next node. Find the position of the current 1-sequence in the trace generated so far, which has to be added in normal form preserving order. If it has to be appended at the last position then continue the process of generating 1-sequences and expanding the tree, otherwise discard the 1-sequence and continue with the next reversal. Repeat this till the height of the tree becomes equal to $d(\pi)$ the trace generated is the new trace.

Once the branch is explored perform backtracking to explore the new branch. Repeat this till all the branches have been explored.

The **TRACE_GENERATION** Procedure of DFA is given as follows:

1. Initialize root node using PROCESS_PERMUTATION.
2. For each reversal
    a) Find the traces generated by the reversal by calling recursive procedure EXPAND.

The routine TRACE_GENERATION initializes the root node with the 1-sequences in the lexicographic order. It then calls the recursive procedure EXPAND to explore the path originating from here.

The **EXPAND** procedure is given as follows:

1. Apply reversal on given genome to get new genome sequence.
2. If level is equal to $d(\pi)$ add it to the trace set and return.

3. Generate next 1-sequence of reversals and store in lexicographic order in new node at next level.
4. For each ρ in node

   If ρ is appended at the end of the predecessor sequence so leads to a new trace then call EXPAND to further explore the branch otherwise skip the reversal and continue with the next reversal.

5. Return to the TRACE_GENERATION procedure.

A sequence is appended to the predecessor sequence in a particular order i.e. first all the commuting reversals gets arranged in lexicographic order then the non-commuting reversals in lexicographic order. Therefore while appending the 1-sequence to the predecessor sequence if it gets appended at the end then it might lead to a new trace so could be expanded further otherwise if it gets appended somewhere in the middle it means that the sequence had already been processed and no need to expand further. In the latter case just skip the reversal as will lead to the formation of redundant trace.

Since many of the traces that could be formed are redundant so identifying them at an earlier stage and terminating the path there reduces the amount of computation and memory consumption. The time spent in matching with other sorting traces is saved with the limitation that total number of sorting sequences cannot be counted. Also, DFA saves memory as only one branch is being explored at a time. The pictorial representation of how DFA works is depicted below:

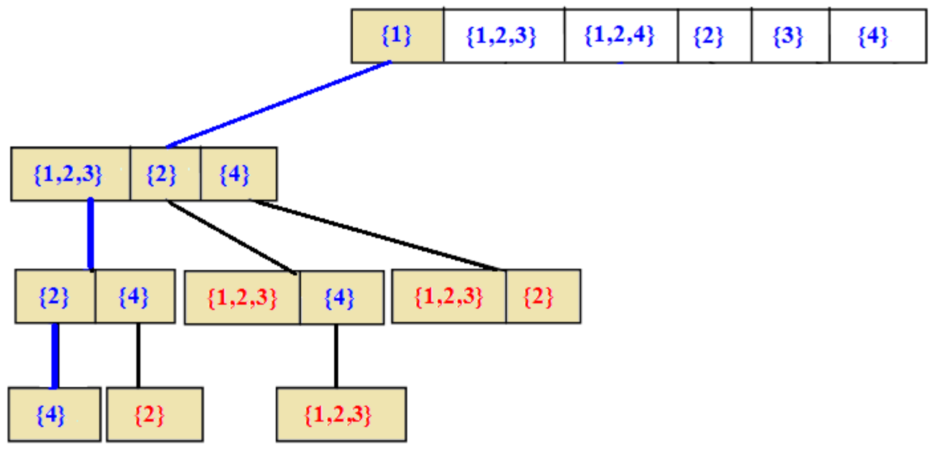


Figure 3.3 (a): Root node initialized with 1-sequences and exploration of first branch giving first solution.

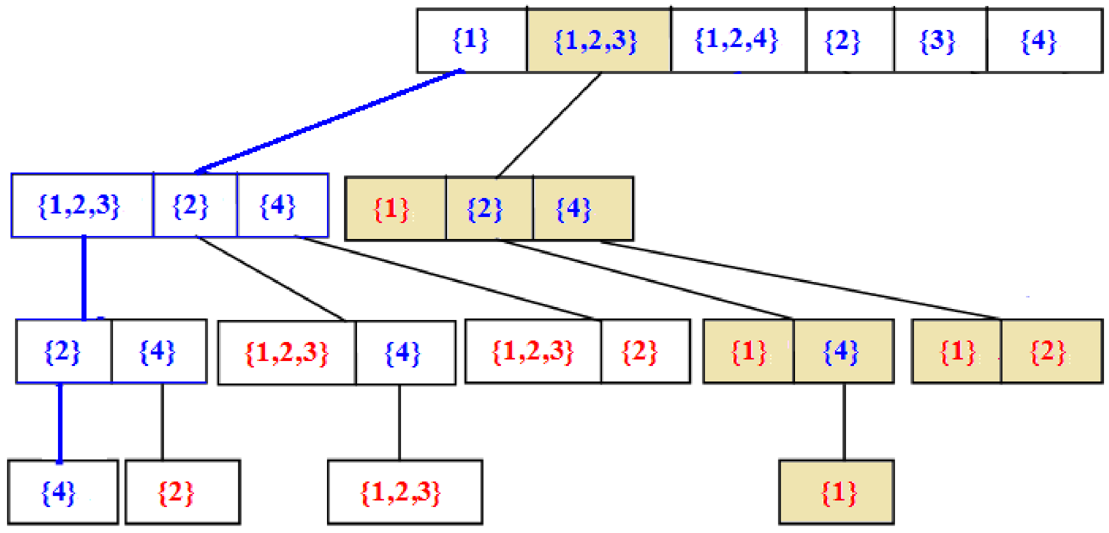


Figure 3.3 (b): Exploring next branch which gets terminated and couldn't reach d(π) level.

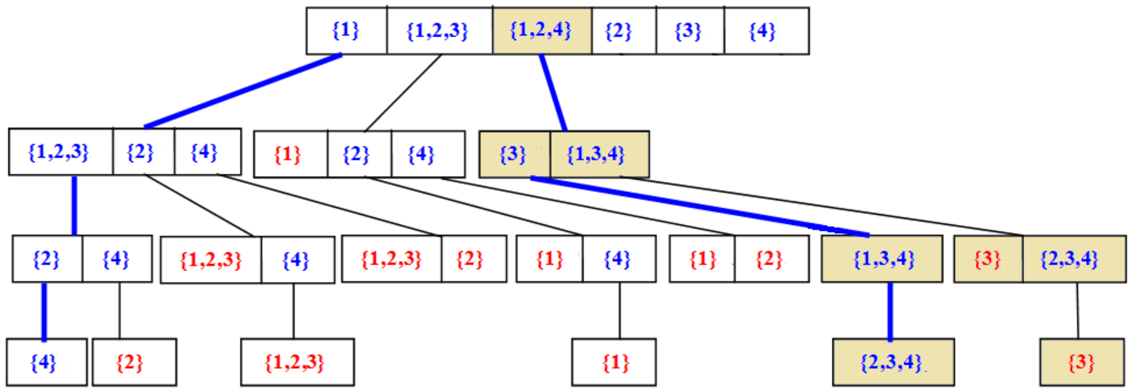


Figure 3.3 (c): On exploring next branch reaches d(π) level so another trace generated.

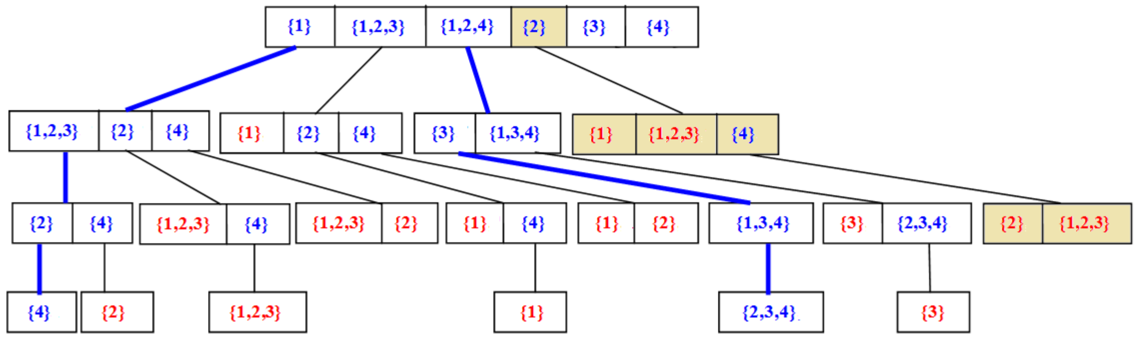


Figure 3.3 (d): Exploring next branch which gets terminated and couldn't reach d(π) level.

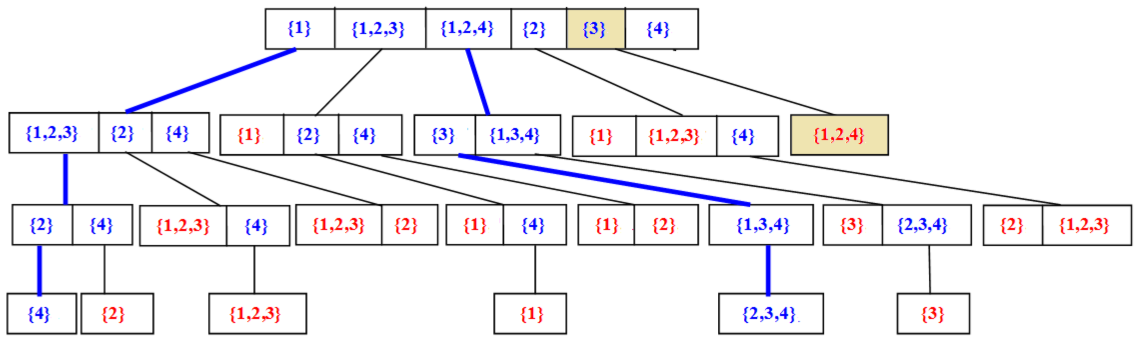


Figure 3.3 (e): Exploring next branch which gets terminated and couldn't reach d(π) level.

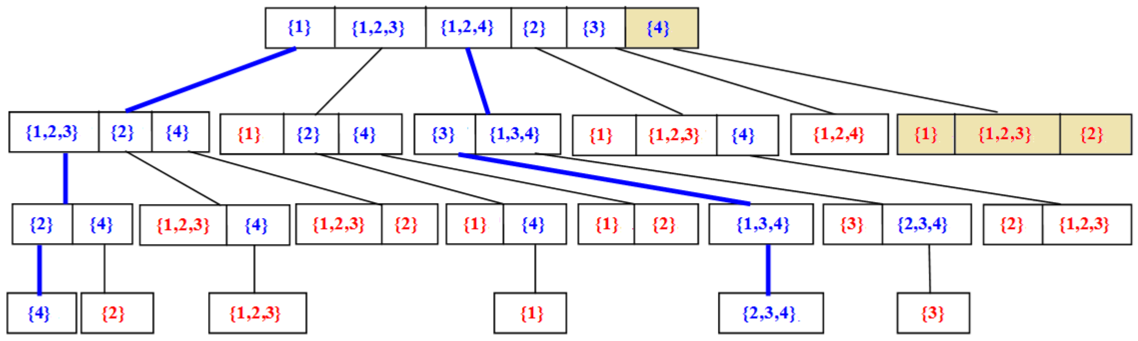


Figure 3.3 (f): Last branch explored which got terminated before reaching d(**π**) level

Figure 3.3: Above tree structure depicts how sorting takes place for permutation (-3, 2, 1, -4) using DFA and branch-wise change in tree structure is shown. Highlighted branch represents current branch, nodes in red color represent branch terminator reversals and bold blue lines represent the trace that sorts the permutation.

3.3. ENUMERATING SOLUTION SPACE FOR PERMUTATIONS WITH UNORIENTED COMPONENTS

A **hurdle** is an unoriented component that does not separate two other unoriented components. If removing the hurdle transforms a nonhurdle into hurdle then it is called **superhurdle** otherwise **simple**. Such a nonhurdle is said to be **protected**. A **fortress** is present in the graph if all the hurdles are superhurdles and they are odd in number.

An unoriented component seperates two other unoriented component if all its black edges lies between any black edges of the other two, then it forms the **separating component.**

These hurdles and fortress costs an extra effort for removing them otherwise permutation could not be sorted. So the reversal distance is modified as the number of steps required to sort the sequence increases. The Reversal Distance can be calculated as,

$$d(\pi) = (n+1) - c + h + f,$$

where n is number of genes in π and c is number of disjoint cycles, h is the number of hurdles and f is a flag that tells fortress exist or not in the breakpoint graph. h and f are 0 when all the components in the graph are oriented components.

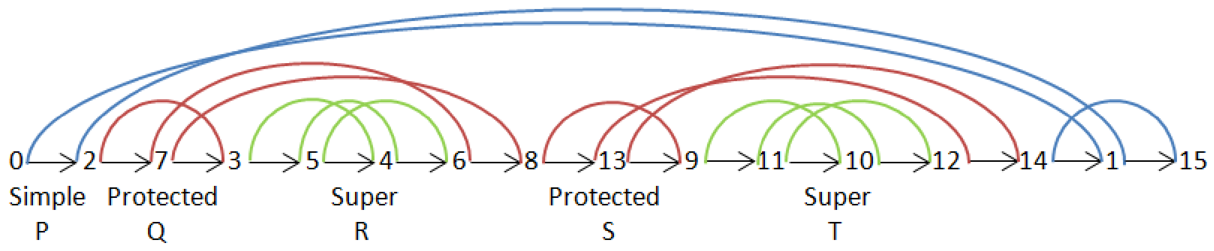


Figure 3.4: Breakpoint graph for an example signed permutation (2, 7, 3, 5, 4, 6, 8, 13, 9, 11, 10, 12, 14, 1, 15). It has 5 components P, Q, R, S and T.

Hurdle graph is constructed for the unoriented components of the breakpoint graph in order to determine their type as simple hurdle or super hurdle or protected nonhurdle. It is constructed by simply traversing the breakpoint graph and joining all the adjacent unoriented components, then on the basis of degree of the vertex and whether the vertex is involved in cycle or not the type of unoriented component is decided. The hurdle graph for the breakpoint graph of Figure 3.4 is shown in Figure 3.5.

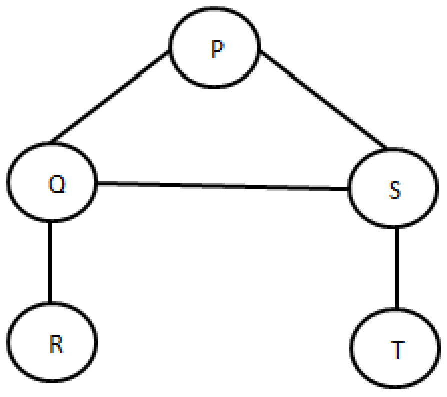


Figure 3.5: Hurdle graph corresponding to the breakpoint graph of Figure 3.4 Component P is a simple hurdle, components Q and S are protected non-hurdles, and components R and T are super hurdles.

Algorithm to identify hurdles, nonhurdles is given below:

1. For every vertex in the Hurdle Graph
    a. If it has degree 2 and belongs to cycle or has degree less than 2 and does not belongs to cycle.
    It is a hurdle and vertex is marked as simple hurdle.

    If its degree is 1 check the degree of its adjacent vertices if they have degree 3 and belongs to cycle or have degree 2 and does not belongs to cycle, then the vertex is marked as super hurdle.

    b. If it has degree more than 2 and belongs to cycle or has degree 2 and does not belongs to cycle.
    It is not a hurdle and vertex is marked as protected non hurdle.

There are two types of reversals that can overcome hurdle and fortress they are cut reversal and merge reversal and are described below:

Cut Reversal: When the extremities of the reversal are in the same cycle of the unoriented component. It is **neutral**, transforms unoriented component into oriented component. It either eliminates hurdle or fortress but does not change number of cycle, such that $\Delta h+\Delta f=-1$. To find cut reversal when:

1. If f=0
   Perform cut on all the pair of edges of the cycles of the simple hurdle so that they can be converted into oriented component. Number of simple hurdle must be either more than 1 or can be 1 when number of super hurdle even, so that fortress formation can be avoided.
   Here Δh is reduced by 1 and Δf remains unchanged so that $(\Delta h + \Delta f)$ is reduced by 1.
2. If f=1
   Perform cut on all the pair of edges of the cycles of the super hurdle or protected hurdle so that super hurdle can be converted into simple hurdle component and fortress formed can be resolved. Here Δf is reduced by 1 and Δh remains unchanged so that $(\Delta h + \Delta f)$ is reduced by 1.

Merge Reversal: When extremities of the reversal are in the cycles of different components, thus rearranging the components and increasing the number of cycle and decreasing the number of hurdles. Here $\Delta c=-1$, $\Delta h+\Delta f=-2$ so that $-\Delta c+\Delta h+\Delta f=-1$

Find Merge Reversal when fortress is absent.

1. If number of simple hurdle is less or equal to 1.

Apply merge on all edges between pairs of:

i. Super hurdles.
ii. Super hurdle and benign component with separating hurdle.
iii. Benign component and separating hurdle.

2. If number of simple hurdle is 2.

Apply merge on all edges between pairs of:

i. Super hurdles.
ii. Super hurdle and benign component with separating hurdle.
iii. Benign component and separating hurdle.
iv. Simple hurdle when number of super hurdle is even.

3. If number of simple hurdle is more than 2.

Apply merge on all edges between pairs of:

i. Super hurdles.
ii. Super hurdle and benign component with separating hurdle.
iii. Benign component and separating hurdle.
iv. Simple hurdle.

Find Merge Reversal when fortress is present.

1. When double super hurdle is present, apply merge on all edges between pairs of:
   i. Super hurdles which form double super hurdle.
   ii. Super hurdle and benign component with separating super hurdle which form double super hurdle.
2. When double super hurdle is present, apply merge on all edges between pairs of super hurdle and protected non hurdle.
3. Apply merge on all edges between pairs of super hurdle and benign component in the same protected non hurdle.

3.3 APPLYING BIOLOGICAL CONSTRAINTS

As the solution space is too large some heuristics needs to be applied in order to find out more relevant solutions. So various biological constraints could be applied on the reversals to logically limit the reversals that are practically not possible or whose probability of occurrence is very less. Common Interval is one such phenomenon in which the clusters of co-localised genes between the two genomes are listed. This common interval is said to have been inherited from the common ancestor and the chances of occurrence of sequence that breaks this block is very less. So we could discard solutions having such reversals, resulting into a much reduced solution set and biologically more feasible.

A pair interval ($[x_\pi, y_\pi]$, $[x_{\pi T}, y_{\pi T}]$) is called a common interval between the permutation π and π_T if it satisfies the following condition:

$$\{ \pi_i \mid i \in [x_\pi, y_\pi]\} = \{ \pi_{Ti} \mid i \in [x_{\pi T}, y_{\pi T}]\}$$

If ith element of a permutation π is j i.e. $\pi_i = j$ then $\pi^{-1}j = i$ means that element j is present at ith index of π. $I_{\pi\pi T}$ is denoted as $\pi_T^{-1}\pi$ (i.e. $I_{\pi\pi T}(i) = \pi_T^{-1}(\pi(i))$. So $I_{\pi\pi T}(i)=j$ means ith element of π is located at jth position of π_T.

For interval [x,y] of π:

$$l(x, y) = \min I_{\pi\pi T}(i) \text{ where } i \in [x, y]$$

$$u(x,y) = \max I_{\pi\pi T}(i) \text{ where } i \in [x,y]$$

$$f(x, y) = u(x, y) - l(x, y) - (y - x)$$

When $f(x, y) = 0$ then ($[x_\pi, y_\pi]$, $[x_{\pi T}, y_{\pi T}]$) represents the common interval. An efficient algorithm to enumerate all common intervals between two permutations has been proposed in[16].

For example the set of common interval between the permutations $\pi = \{-7\ 4\ 5\ 6\ -3\ 8\ -2\ -1\}$ and $\pi_T = \{1...8\}$ are I={{1,2},{2..8},{3…6},{3…8},{4,5},{4…7} {5,6}}

The filter process is done as soon as the 1-sequences are generated. Each of the reversals is checked if they break the common interval or not. A common interval breaks when one end of reversal lie inside the interval and other end outside the interval and the interval is not completely contained within the reversal, such that after performing the reversal the elements of the common interval are not contiguous. Those reversals that break the interval are not added to the set of 1-sequences. This filtration process is done at each step of the algorithm as soon as the 1-sequences are generated. So the process keeps on reducing the solution set generating the solutions that define most probable evolutionary scenario.

Let θ represent one of the intervals amongst the set of common intervals. A reversal ρ breaks an interval θ if ρ overlaps with θ and θ is not completely contained in ρ. For example in the permutation (-2 -5 4 -7 3 -8 6 -1) having θ = {2,…,8} we observe that the reversal ρ = {1, 3, 4, 6, 7, 8} breaks the interval.

The common intervals between the source and the target genome are found at the starting. Those sorting sequences that do not break any common interval are called as perfect sorting sequence.

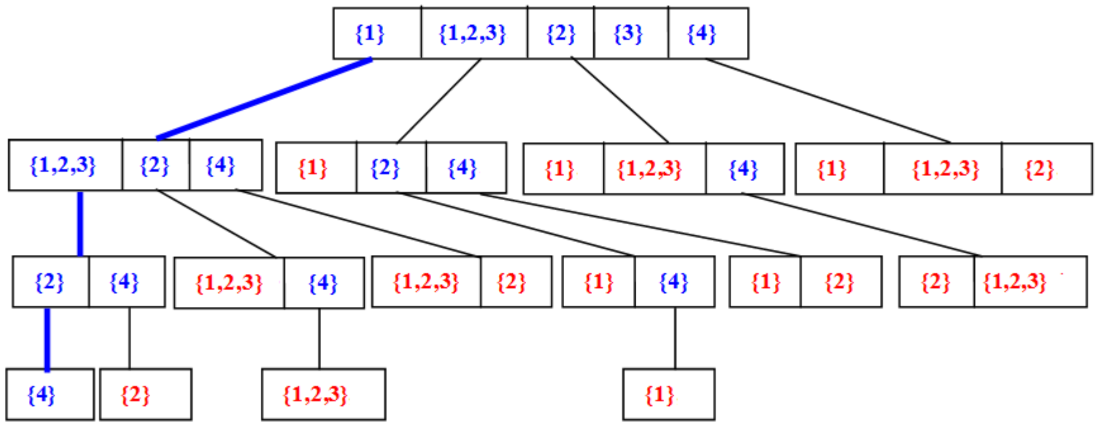


Figure 3.6: Tree structure of the traces that sort the permutation (−3, 2, 1, −4) after filtering the sequences that break common interval.

As in figure 3.6, it is clear that the traces have reduced from two to just one. In this way the solution set can be reduced which could make the analysis easier for the biologists.

3.4. THEORETICAL COMPLEXITY

Total number of sequences that could be generated is $O(N. n^{kmax})$. For finding all 1-sequence takes $O(n^3)$ time and processing prefix takes $O(n^4)$ time this is computed as:

No of 1-sequences generated * Adding them to previous prefix

Comparisons to merge equivalent traces takes $O(n^3 \log(N. n^{kmax})$ which is computed as:

No of sequences * (No of 1-seq * comparing with all traces)

So, theoretical complexity for generating traces with counting number of solutions in each case is $O(N. n^{kmax + 4})$ where,

- n is number of elements in the sequence.
- N is total number of traces.
- The width (k) of a trace t is defined as the biggest subset of reversals of t such that every pair of reversals in this subset commutes.
- kmax is the max width at the particular i-trace.

  Time saved in generating traces without counting all sorting sequences is of the order $O(N.n^{kmax+3} \log(N.n^{kmax}))$.

3.5. SUMMARY

At an initial stage all the solutions for the problem *Sorting by Reversals* was generated and results were verified with pre-existing software baobabLuna. After that the solutions were grouped into traces for compressed representation of solution space and again the results were verified. On the basis of it the BFA and DFA approaches were given and were modified for a more efficient performance (discussed in chapter 5). The well-known hurdle problem was solved for providing a solution to those permutations that couldn’t be sorted till now. Later, due to the exponential size of the solution space it was difficult to carry out analysis so, with the help of biological constraints practically impossible solutions were removed.

CHAPTER 4
IMPLEMENTATION DETAILS

4.1. CLASSES AND THEIR DESCRIPTION

Breakpoint graph is the basis of the sorting process. It has been implemented with 3 views: Edge view, Cycle view and Component view. The components are classified as oriented and unoriented components. The unoriented components are further categorized as simple hurdle, superhurdle or protected non-hurdle. ArrayList is used to store the 1-sequences at each node which can dynamically grow. Each 1-sequence is stored in a SortedSet of integers. Whenever a new element is added to the set it is stored in an order that is predefined. This is used so that the elements of 1-sequence are automatically sorted on every addition of element so workload of processing them later is reduced. Trace is represented as ArrayList of SortedSet of integers. Using these data structures we have implemented the depth-first approach. HashMap has been used to efficiently process separating hurdles for benign components. It provides direct access to any value with help of its key, which is a benign component in our case. To add common interval constraint the code, the common intervals are stored as ArrayList of ArrayList of integers.

Breadth First approach and Depth first approach were implemented to find out number of solutions and number of traces. Biological constraints were applied to reduce the number of solutions as well as the computational time. So following are the implementation details of various classes that were employed to perform the given task. The code for enumerating all plausible sorting traces has been implemented in Java. The implemented code has been tested with several random permutations as well as permutations for real genomes. Some have hurdles in the beginning and in some hurdles were detected later and the corresponding reversal sequence was discarded, wherever distance does not reduce.

4.1.1. Generator Class

This class holds the main function

Input:

i. Number of genes.
ii. Gene order in the chromosome.

Output:

i. Final traces to sort the input gene-order using either Breadth first approach or Depth first approach.

Main Function:

i. Create object of BreadthFirstApproach or DepthFirstApproach.
ii. Call sorting method to generate the traces.
iii. Display traces.

4.1.2. BreadthFirstApproach

Functions:

1. **sortingByReversal()**
   i. Expand the input sequence.
   ii. Create Graph object.
   iii. Display distance.
   iv. Create Trace object for each 1-sequence of reversals in the graph and add it to ‘i_trace’ which holds set of solutions.
   v. For each i_trace
      a. Apply the reversals (locateRange())
      b. Find 1-sequence for the new graph, concatenate with i-trace in normal form order to get i+1-trace.
      c. Add i+1-trace to ‘seq1’ temporary solution.
      d. Check if i+1-trace in ‘seq1’ is present in temporary solution set ‘new_i_trace’.
         If present, increment the count of existing trace.

         Else, add it to the ‘new_i_trace’.
   vi. Assign ‘new_i_trace’ to ‘i_trace’.
2. **locateRange()**
   Finds the reversal interval in a permutation and reverse the elements in the interval.
3. **display_i_trace()**
   Display each i_trace and add the count to give total number of solutions.

4. **expandPermutation()**

   Expands the permutation in the form of 2i and 2i+1 along with it appends 0 and 2n+1 where n is the number of genes in the sequence.

4.1.3. DepthFirstApproach

Functions:

1. **sortingByReversal()**
   1. Expand the input sequence.
   2. Create Graph object.
   3. Display distance.
   4. For each 1-sequence of Graph in node.
      1. Create a trace object for it and add it to 'i_trace'.
      2. Call 'branchExpand()' to form the complete trace.
2. **branchExpand()**

   It is a recursive function that forms a complete trace terminates when distance becomes 0 then returns forming a complete trace.

   Parameters:

   i. Initial permutation.
   ii. Particular reversal that has to be applied.
   iii. Trace for which the function called.
   iv. Distance of the graph.

Working:

i. Apply the reversal (locateRange()).
ii. Find 1-sequence for new graph.
iii. Find distance of new graph.
iv. Check if distance = 0
v. The trace is complete if it is not present in solution set i.e. 'i_trace' then add it. Otherwise, increase the count of already present trace.
vi. Return.
vii. Else,

a. Concatenate 1-sequence with i-trace in normal form order to form i+1-trace.
b. Check the position of insertion, if it is at the end that means leading to generation of normal form trace so call 'branchExpand()' on it.
c. Otherwise, if inserted in mid-way
   Check if it is already present in solution set 'i_trace' then increase its count otherwise call 'branchExpand()' on it.

3. **locateRange()**
   Finds the reversal interval in a permutation and reverse the elements in the interval.

4. **display_i_trace()**
   Display each i_trace and add the count to give total number of solutions.

5. **expandPermutation()**
   Expands the permutation in the form of 2i and 2i+1 along with it appends 0 and 2n+1 where n is the number of genes in the sequence.

4.1.4. Graph

Data Members:

1. **Edge[]: BlackEdge**
   Array of black edges in the graph.

2. **cycleList: ArrayList<Integer>**
   List of cycles in the graph.

3. **componentList: ArrayList<Integer>**
   List of components in the graph.

4. **node: TreeSet<TreeSet<Integer>>**
   1-sequences of reversals in lexicographic order for the graph.

5. **n: int**
   Number of genes in the gene sequence.

6. **c: int**
   Number of cycles in the graph.

7. **h: int**

   Number of hurdles in the graph.

8. **s: int**

   Number of superhurdles in the graph

9. **f: int**

   Flag for marking the presence or absence of fortress.

10. **d: int**

    Reversal distance of the graph.

Constructor:

i. Creates a Breakpoint graph which has 3 views: Edge view 'Edge[]', cycle view 'cycleList' and Component view 'componentList'
ii. All the 1-sequences of this particular graph are stored in 'node'.
iii. The distance is calculated and checked if it reduces the previous distance then only continue otherwise it is an unsafe sequence.

Functions:

1. **constructHurdleGraph()**

   i. For each edge in the breakpoint graph that belongs to unoriented component

      c. If it is not first or last edge

         Check previous and next edge and find its component.

         If they belong to some other unoriented component means those components are adjacent then maintain its entry in HurdleGraph.

      d. If first edge

         Check next edge and find its component.

         If it belongs to some other unoriented component then maintain its entry in HurdleGraph.

      e. If last edge

         Check previous edge and find its component.

If it belongs to some other unoriented component then maintain its entry in Hurdle Graph.

2. **findComponenNumberForEdge()**

   Finds the component to which the given edge belongs by checking entry in the EdgeGroup of each component.

3. **hurdleDetection()**

   i. Degree of each unoriented component is calculated.
   ii. Each unoriented component is marked if it forms a cycle in the Hurdle Graph.
   iii. For every component in the Hurdle Graph

      a. If it has degree 2 and belongs to cycle or has degree less than 2 and does not belongs to cycle.
         It is a hurdle and component is marked as simple hurdle.

         If its degree is 1 check the degree of its adjacent nodes if they have degree 3 and belongs to cycle or have degree 2 and does not belongs to cycle, then the component is marked as super hurdle.

      b. If it has degree more than 2 and belongs to cycle or has degree 2 and does not belongs to cycle.
         It is not a hurdle and component is marked as protected non hurdle.

4. **findCycleinGraph()**

   Calls a recursive procedure DFS to generate all the possible paths in the hurdle graph from a particular vertex as the graph is connected graph so all possible paths are covered in it.

5. **calculateDistance()**

   i. Calculates the value of h i.e total number of hurdles.
   ii. Marks f as 1 if fortress is present i.e only superhurdles are present and are odd in number.
   iii. Calculates the reversal distance using the formula $d=(n+1)+h+f-c$.

6. **findReversal()**

   i. For each component that are oriented

      For every oriented cycle

      a. Find every possible diverging pair by considering the edges that are opposite in direction.
      b. 1-sequence is found out between the pairs add it to node if it does not breaks Common Interval (if required).

   ii. Return node

7. **findNeutralReversal()**

   i. If f=0,

      For each simple hurdle if h-s>1 or h-s=1 and s%2=0

      For every cycle

      a. Consider every possible pair of edge.
      b. 1-sequence is found out between the pairs add it to node if it does not breaks Common Interval (if required).

   ii. If f=1,

      For each superhurdle or protected nonhurdle

      For every cycle

      a. Consider every possible pair of edge.
      b. 1-sequence is found out between the pairs add it to node if it does not breaks Common Interval (if required).

8. **findMergeReversal()**

   If f=0,

   a) If number of h-s=1,

   i. If number of s is even. Find all merge reversal between:

      - Pair of super hurdles.
      - Pair of super hurdles and benign components with separating super hurdle.
      - Pair of benign component with two separating super hurdles.

ii. If number of s is odd. Find all merge reversal between:
   - Pair of simple and super hurdles.
   - Pair of super hurdle and benign component with separating hurdle.
   - Pair of benign component with two separating hurdles.

b) If number of h-s=2,

i. If number of s is even. Find all merge reversal between:
   - Pair of hurdles.
   - Pair of hurdle and benign components with separating hurdle.
   - Pair of benign component with two separating hurdles.

ii. If number of h-s is odd. Find all merge reversal between:
   - Pair of simple and super hurdles.
   - Pair of super hurdles and benign components with separating hurdle.
   - Pair of benign component with two separating super hurdles such that at least one is super hurdle.

c) If number of h-s>2, find all merge reversal between:
   - Pair of hurdles.
   - Pair of hurdle and benign components with separating hurdle.
   - Pair of benign component with two separating hurdles.

If f=1,

a) When double super hurdle exist, find all merge reversal between:
   - Super hurdle pair forming double super hurdle.
   - Super hurdle and benign component with separating super hurdle (given they form double super hurdle pair)

b) When double super hurdle does not exist, find all merge reversal between super hurdle and protected non hurdle.

c) Find all merge reversal between super hurdle and benign component in the same protected non hurdle.

9. **find1Sequence()**
    1. Takes cycle and 2 diverging edges as input.
    2. Finds out all the elements between the pair of edges.
    3. Store them in increasing order in one_seq.
    4. Return one_seq
10. **displayGraphEdges()**

    Displays all the edges of the graph.
11. **displayCycle()**

    Displays all the cycles of the graph.
12. **displayComponent()**

    Displays all the components of the graph.

Class Comp

It inherits the Comparator class and overrides the compare function of it.

4.1.5. BlackEdge

Data Members:

1. **Left: Node**

    Represents the left-node of the edge.
2. **Right: Node**

    Represents the right-node of the edge.

4.1.6. Node

Data Members:

1. **Key: int**

    Stores the key integer of the node of graph.

4.1.7. Cycle

Data Members:

1. **type: int**

Stores the type of cycle -1 for unoriented , 0 for trivial and 1 for oriented cycle.

2. **EdgeList: ArrayList<Integer>**

   List of edge numbers that is contained in the cycle.

3. **Direction: ArrayList<Integer>**

   Holds the direction of edge corresponding to in the EdgeList.

4. **min: int**

   Index of leftmost black edge, helps in finding out span of component.

5. **max: int**

   Index of rightmost black edge, helps in finding out span of component.

Functions:

1. **constructCycle()**
   i. For unvisited Edge in EdgeList
      i. Let ‘s’ be the first node of first unvisited edge in EdgeList .
      ii. Search for ‘s’ in the node of each edge, once found traverse that edge and determine its direction (0 if left to right , 1 otherwise). Now the other end of edge becomes ‘s’.
      iii. If ‘s’ is odd then change it to ‘s-1’ otherwise to ‘s+1’ (this is done so that we do not need to keep track of gray edges, they are stored in order which itself represents the gray edge.
      iv. Repeat the process from ii. Step till a visited node is encountered.
      v. For each cycle maintain list of edges it holds, leftmost and rightmost edge and type of cycle i.e. trivial, oriented and unoriented.

4.1.8. Component

Data Members:

1. **cycleGroup: ArrayList<Integer>**

   List of cycle numbers that is contained in the component.

2. **type: int**

   -1 for unoriented , 0 for trivial and 1 for oriented components.

3. **subtype: int**

   Valid only when type = -1 i.e. unoriented component, specifies simple hurdle i.e. 1 or super-hurdle i.e 1 or Protected Nonhurdle i.e. 0

4. **min: int**

   Index of leftmost black edge in leftmost cycle.

5. **max: int**

   Index of rightmost black edge in rightmost cycle.

Function:

1. **constructComponent()**

   For all unvisited cycle in the cycleList

   i. Initialize component as the first unvisited cycle.
   ii. For all intersecting cycle include it in the component.
   iii. When no more intersecting cycle found include it in the componentList.

4.1.9. Trace

Data Members:

1. **i_seq: ArrayList<SortedSet<Integer>>**

   Holds the sequence of sub-words in each trace.

2. **count: int**

   Holds the count of the trace when repeated solutions.

Trace Representation:

A Trace's object holds the 'i_seq' which is the normal form trace till the i^{th} iteration. The sub-words in 'i-seq' are separated by dummy variables {0}. It also holds the count which represents number of similar traces encountered so far.

Functions:

1. **sequenceInsert()**

   i. Takes the sequence to be inserted in the 'i_seq' of current trace object.

ii. Finds out the last sub-word with which the ‘curr_seq’ overlaps (i.e. non commuting), -1 when no such sub-word found.

iii. Mark the start and end position of the next sub-word with the help of dummy variable encountered.

iv. If end position found insert in lexicographic order between the start and end position. (lexicoOrder())

v. But if no end index found means it is the last sub-word so form a new sub-word and insert at the end.

2. **lexicoOrder()**

   Finds out the index which is the correct position for insertion by checking the ‘lexicographic()’ order within the sub-word.

3. **lexicographic()**

   Returns 1 if source element is smaller and has to be appended before target element, returns 0 if source element is larger than target and we still need to find a larger target.

CHAPTER 5
RESULTS AND DISCUSSIONS

5.1. RESULTS

A comparative test was carried out between the Breadth First Algorithm and the Depth First Algorithm while enumerating traces without counting the number of solutions in the equivalence class. It was found out that DFA produces results more efficiently than BFA.

20 random permutations were generated with two factors keeping n=d and n=2d. The permutations were generated by randomly selecting two cut points on an identity permutation of size n and performing reversal between them, then checking the distance increased by 1. In this way permutation of desired distance is generated.

The results were obtained on these 20 permutations and later on their average was taken to obtain a generalized result. The tests were carried out on an AMD A-10 2.5 GHz processor with 8 GB RAM running Windows 8.1 64 bit. JVM was allocated with maximum memory 1.5 GB (using parameter -Xmx1550m). The comparative analysis of memory and time for the two approaches is shown below:

5.1.1. Memory Analysis

Memory used (in bytes) in BFA and DFA is computed with the help of separate thread measuring memory at frequent intervals using the object-Runtime and method- totalMemory() and freeMemory(). The results obtained are tabulated below:

	Average memory used (in bytes)			
	n=2d		**n=d**	
distance	**BFA**	**DFA**	**BFA**	**DFA**
3	176510.40	168275.99	161552.40	150909.99
4	245298.40	242999.99	236944.00	229233.99
5	470494.80	475866.800	511244.39	508421.99
6	614732.00	587420.400	750019.20	714839.60
7	811786.40	774811.200	932090.39	736254.00
8	1393424.00	825480.40	2872953.20	1304331.19
9	3730674.40	1713477.59	1.176104E7	4381798.40
10	16021698.39	6910462.0	66637731.19	27234663.20
11	63956969.19	23313340.79	363134056.67	145726384.37
12	117069896.55	63909832.67	Out of memory	1392756680.50

Table 5.1: Average maximum memory used

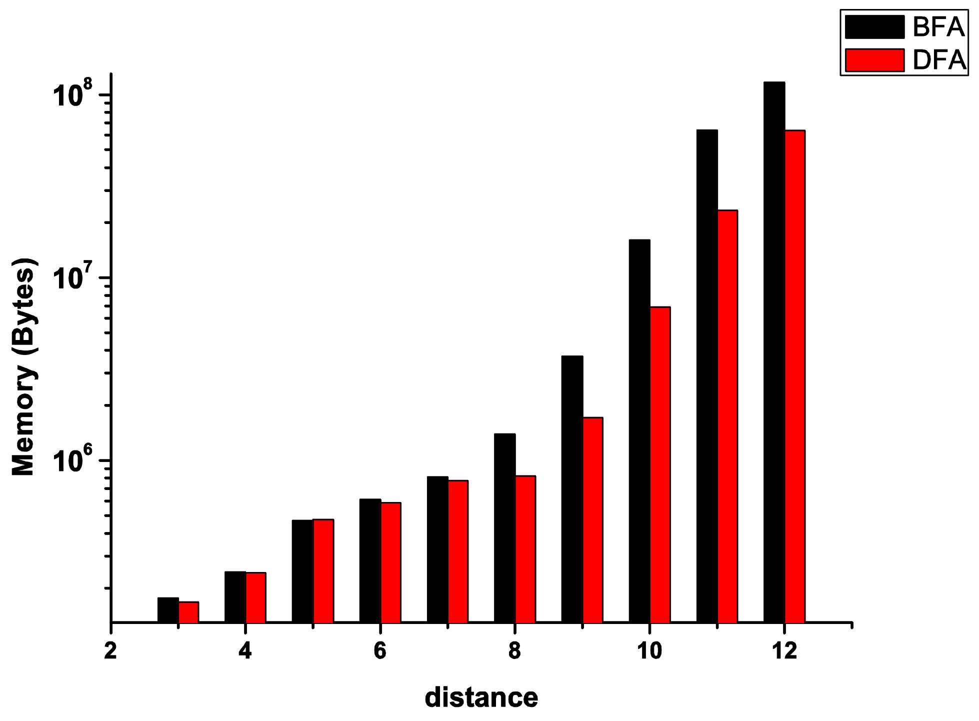


Figure 5.1.1: Memory used when n=2d.

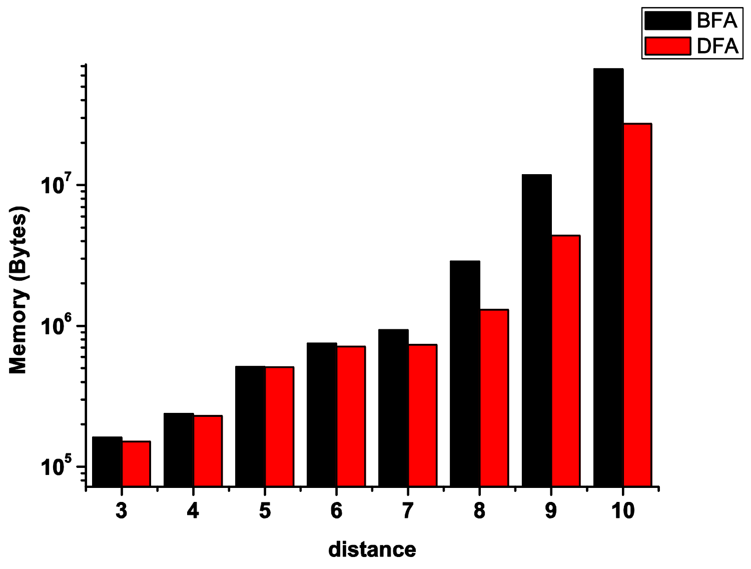


Figure 5.1.1: Memory used when n=d.

Figure 5.1: Graphs for comparing average maximum memory (in bytes) used for BFA and DFA.

From the above graph it is clear that DFA is more memory efficient than BFA, even when BFA encounters out of memory problem DFA continues to run efficiently. This is due to the fact that each time only one branch is explored and before exploring the next branch memory is freed. In this way memory consumption is reduced.

5.1.2. Time Analysis

Elapsed time (in seconds) in BFA and DFA is computed by taking the system's clock value before and after execution with the help of object-System and method-currentTimeMillis(). The results obtained for various data are tabulated below:

	Average time elapsed (in seconds)			
	n=2d		**n=d**	
distance	**BFA**	**DFA**	**BFA**	**DFA**
3	0.00100	0.00050	0.01175	0.00620
4	0.00125	0.00055	0.0031	0.00020
5	0.00180	0.00105	0.00255	0.00159
6	0.00475	0.00405	0.01010	0.00720
7	0.02560	0.02190	0.05860	0.05060
8	0.10855	0.09745	0.31055	0.26915
9	0.54845	0.47550	2.20655	1.80400
10	5.11350	4.33045	13.79930	11.21290
11	16.91955	14.66605	68.43000	55.52300
12	90.52402	85.76299	Out of memory	60.52055

Table 5.2: Average elapsed time

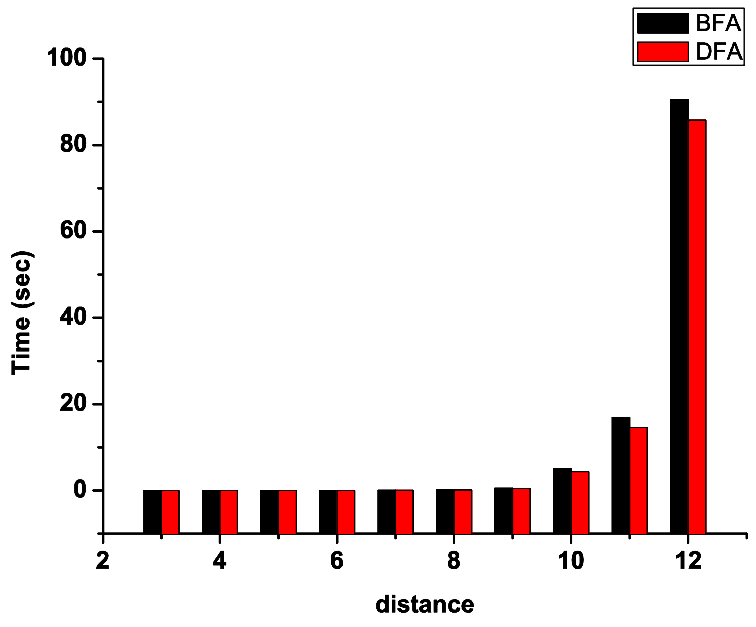


Figure 5.2.1: Time elapsed when n=2d.

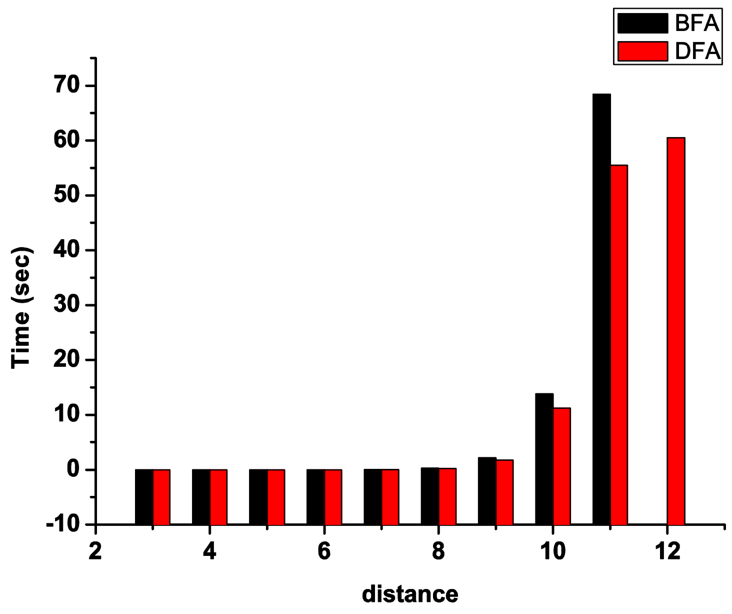


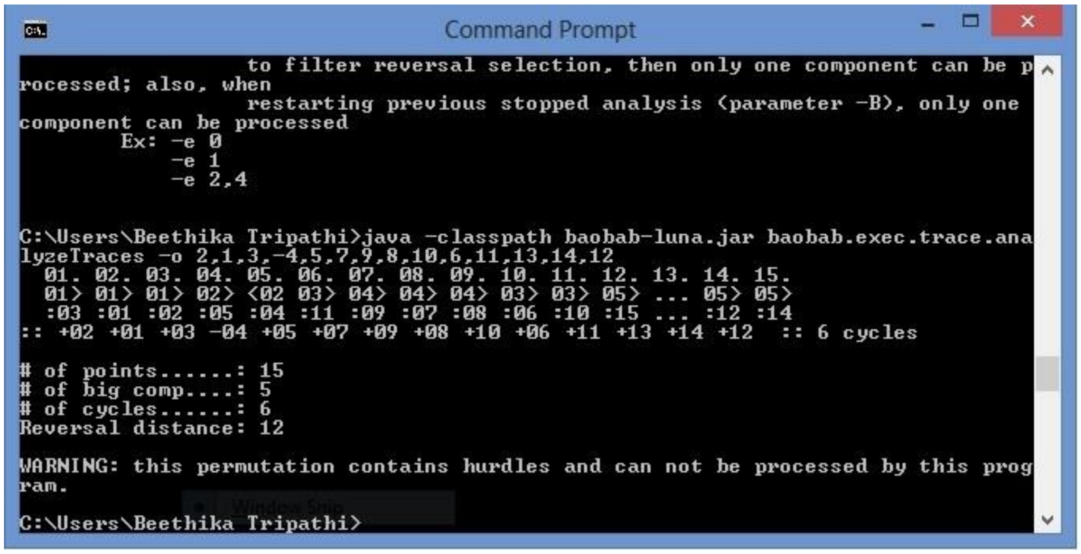
Figure 5.2.1: Time elapsed when n=2d.

Figure 5.2: Graphs for comparing average time elapsed (in sec) for BFA and DFA.

From the above graph it is clear that BFA and DFA are more or less similar with respect to time required for computation.

5.1.3. Examples with hurdles

The baobabLuna tool fails to give solution in the presence of hurdle. Figure 5.3 shows the result of executing baobabLuna for an input permutation having hurdle.



```
                  to filter reversal selection, then only one component can be p
rocessed; also, when
                  restarting previous stopped analysis (parameter -B), only one
component can be processed
         Ex: -e 0
             -e 1
             -e 2,4


C:\Users\Beethika Tripathi>java -classpath baobab-luna.jar baobab.exec.trace.ana
lyzeTraces -o 2,1,3,-4,5,7,9,8,10,6,11,13,14,12
  01. 02. 03. 04. 05. 06. 07. 08. 09. 10. 11. 12. 13. 14. 15.
  01> 01> 01> 02> <02 03> 04> 04> 04> 03> 03> 05> ... 05> 05>
  :03 :01 :02 :05 :04 :11 :09 :07 :08 :06 :10 :15 ... :12 :14
:: +02 +01 +03 -04 +05 +07 +09 +08 +10 +06 +11 +13 +14 +12  :: 6 cycles

# of points......: 15
# of big comp....: 5
# of cycles......: 6
Reversal distance: 12

WARNING: this permutation contains hurdles and can not be processed by this prog
ram.

C:\Users\Beethika Tripathi>
```

Figure 5.3: Executing baobabLuna with input permutation (2,1,3,-4,5,7,9,8,10,6,11,13,14,12).

Our implementation is capable of giving solutions in the presence of hurdles (except double super hurdles) that may be present in the input permutations or may arise later in the process of sorting. Table 5.3 shows the output for various permutations that contained different types of hurdle.

Input permutation	d	Hurdle	Fortress	Traces	Time(sec)	Memory(Bytes)
2,1,3,-4,5,7,9,8,10,6,11,13,14,12	12	Simple, protected nonhurdle	0	172	2.421	903704
2,7,3,5,4,6,8,13,9,11,10,12,14,1	13	Simple, Super, protected nonhurdle	0	1048	8.790	2893504
17,1,3,8,4,6,5,7,9,11,13,12,14,10,15,2,16	16	Super, protected nonhurdle	1	1048	120.631	3850136

Table 5.3: Examples of permutations with hurdle their distance, total number of sequences, total number of traces, time elapsed in seconds and memory used in bytes.

5.1.4. Number of traces with and without CI constraints

When the Biological constraint i.e. Common Interval is applied then the solution set is drastically reduced. These logically removes those traces that are not biologically possible thus making the analysis easier. The variation in the number traces with and without CI constraints is shown below:

	Average number of traces			
	n=2d		n=d	
distance	Without CI	With CI	Without CI	With CI
3	0.95	0.60	0.80	0.39
4	1.50	0.49	1.80	0.35
5	3.10	1.05	4.60	1.00
6	7.75	0.85	15.25	3.45
7	23.40	8.05	64.85	5.89
8	118.45	36.90	358.80	130.85
9	273.65	87.80	1914.75	619.65
10	2051.55	121.30	12550.90	2851.40
11	4749.40	364.60	68369.85	3567.55
12	10193.25	3453.50	84145.05	9532.90

Table 5.4: Variation in average number of traces with and without Common Interval constraint.

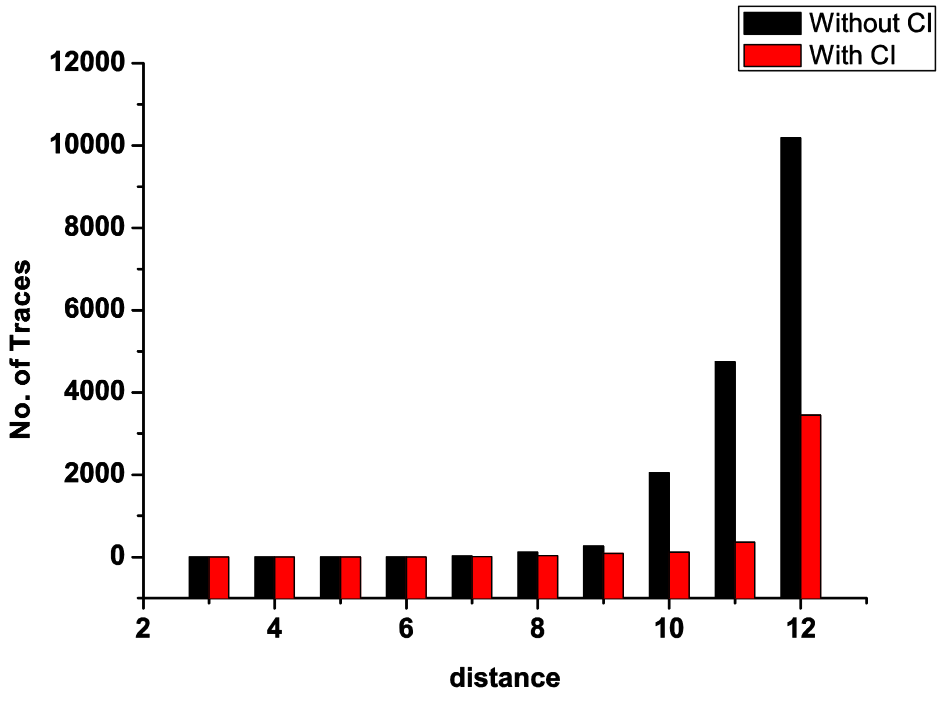


Figure 5.4 (a): Average traces when n=2d.

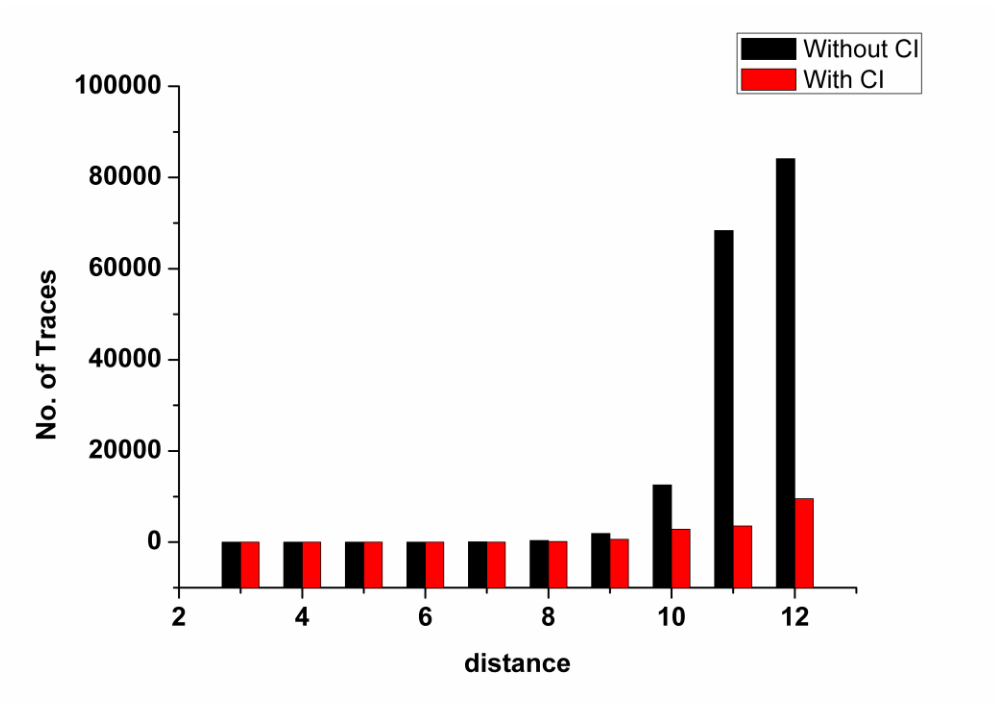


Figure 5.4 (b): Average traces when n=d.

Figure 5.4: Graph for variation in average number of traces with and without Common Interval constraint.

5.1.5. Real examples

Some real examples [23-24] were tested some contained hurdles also, their total sequences, total traces, time elapsed (in sec.) and memory used (in bytes) are calculated and tabulated below:

Input permutation	Species	d	Hurdle	Sequences	Traces	Time	Memory
7,1,2,4,5,3,6,8	conserved blocks in cpDNA of tobacco and Lobelia fervens	6	Simple	1540	23	94	460672
4,6,1,7,2,3,5,8	conserved linkage groups in mouse and human X chromosomes	7	Simple	13088	108	143	628672
-4,-3,-2,1,-13,-15,14,-16 ,8,9,10,-11,12,5,6,7	Mouse and rat	9	No	776907	323	1021	1548328
-6,-5,4,13,14,-15,16,1,-3, 9,-10,11,12,-7,8,-2	Human and mouse	10	No	3362310	218	1254	873296
-13,-4,5,-6,-12,-8,-7,2, 1, -3,9,10, 11, 14,-15,16	Human and rat	10	No	2419750	418	1406	1719968

Table 5.5: Examples of permutation of real species their distance, total number of sequences, total number of traces, time elapsed in seconds and memory used in bytes.

5.2. DISCUSSIONS

Depth first approach is more efficient than breadth first approach especially with respect to memory consumption. It requires lesser memory than the other one. Hurdles can even exist in real world scenario so they must not be ignored.

The universe of sequences and class has been reduced by applying the biological constraints and filtering those sequences which are biologically less probable to occur. This could help the biologists to work on large permutations or permutations with large reversal distance, and a better evolutionary scenario is characterized for the two genomes. The solution space is explored in depth first manner to list the normal form of possible traces.

This version could further be extended to form a parallel version of the algorithm. Data Parallelism can be employed where each branch is handled by a different processor and all the processors are doing the same work. Each branch is explored separately by different processors independently starting from each reversal in the optimal 1-sequences of reversals of the input permutation. This results in better time and space complexity.

Other different types of constraints could be applied depending upon the type of application. The concept of near-perfect trace could be used which allows bounded number of breaks per trace. This is done no perfect sorting sequence exist so a near-perfect sorting sequence is generated.

There are some limitations with biological constraints. It is not necessary that sequence holding the constraints exist so might lead to empty results, which is undesirable. Some relaxations may be done at each point of time to get non empty results but it is very much time consuming and extra efforts have to be applied.

CHAPTER 6
CONCLUSION

6.1. CONCLUSION

This work provides a general solution to the problem of sorting by reversals. Breadth First Approach and Depth First Approach were modified to produce more efficient solutions by discarding those branches that lead to the generation of redundant traces. But still DFA turns out to be much more memory efficient than BFA (as discussed in Section 5.1.1) as after exploring one branch it frees the memory before proceeding to another. DFA can also be easily parallelized so that solutions could be obtained in a much faster manner.

With the preliminary work of Siepel's, sorting reversals for well-known hurdles problem could have been achieved. Dealing with hurdles was a challenging task as the total number of sorting sequences increase drastically when hurdles are present in the breakpoint graph (mentioned in section 3.3). It was observed that many of the i-sequences of neutral or joint reversal belonged to same equivalence class, resulting in drastic reduction of sorting traces in comparison to non-hurdle permutation of same size and distance.

Due to the exponential size of the solution set it was a difficult task for the biologists to analyze each and every solution. Some heuristics need to be applied to logically reduce the solutions. Biological constraints were applied to logically filter those solutions that are not practical as certain traits are inherited from ancestors and those don't change. So any solution that tried to alter such traits was filtered out thus making the solution set much closer to reality.

CHAPTER 7
FUTURE WORK

- This work provides the solution only when genomes transform through reversals and it could be extended to other genomic rearrangement operations.
- If duplicate genes are present in the genome then it has to be removed by listing all the possible exemplars that could be formed and then working on it.
- The concept of near-perfect trace [25] could be implemented that allows bounded number of breaks per trace, such that non-empty result is obtained if there is no perfect sorting sequence.
- Other different types of biological constraints apart from common interval could be applied depending upon the applications.
- Parallel version of DFA algorithm could be implemented, where data parallelism is applied for each branch this could increase the speed of generation of solutions.
- The solution could be modified to deal with permutations of larger size by producing partial solutions at various stages and later on combining those solutions.

LIST OF PUBLICATION

Papers Accepted:

Beethika Tripathi and Amritanjali, “An improved algorithm for reducing the solution space of sorting by reversals”, International Journal of Basic and Applied Biology, vol. 2(2), pp. 23-27, 2014.

Papers under review:

Beethika Tripathi and Amritanjali, “Enumerating All Plausible Traces of Reversals to Sort a Signed Permutation”, CIBCB, IEEE, 2015.

REFERENCES

[1] S. Hannenhalli and P. Pevzener, “Transforming cabbage into Turnip (polynomial algorithm for sorting signed permutation by reversals), J. ACM, vol. 48, pp. 1-27, 1999.

[2] G. Tesler, “GRIMM: genome rearrangements web server”, Bioinformatics, vol. 18(3), 492–493, 2002.

[3] G. Tesler, “Efficient algorithms for multichromosomal genome rearrangements”, Journal of Computer and System Sciences, vol. 65(3), 587-609, 2002.

[4] Y. Huang, C. Huang, C. Y. Tang and C. L. Lu, “SoRT2: a tool for sorting genomes and reconstructing phylogenetic trees by reversals, generalized transpositions and translocations”, Nucleic Acids Res., vol. 38(Web Server issue): W221–W227, 2010.

[5] M. D. V. Braga, “baobabLuna: The solution space of sorting by reversals.", Bioinformatics, vol. 25(14), pp. 1833-1835, 2009.

[6] R. Hilker, C. Sikinger, C.N.S. Pedersen and J.Stoye, “UniMoG—a unifying framework for genomic distance calculation and sorting based on DCJ”, Bioinformatics, vol. 28(19): 2509–2511, 2012.

[7] M. D. V. Braga and J. Stoye, “The solution space of sorting by DCJ”, Journal of Computational Biology, vol. 17(9), pp. 1145-1165.

[8] G. Brightwell and P. Winkler, “Counting linear extensions is #P-complete”, STOC ’91: Proceedings of the twenty-third annual ACM Symposium on Theory of Computing, ACM Press, 1991.

[9] Caprara, A.: "Sorting by reversals is difficult", in: Proc. 1st Ann. Int’l Conf. Comput. Mol. Biol. (RECOMB 1997), ACM Press, New York, pp. 75–83, 1997.

[10] G. Blin, G. Fertin, and C. Chauve, “The breakpoint distance for signed sequences”, Texts in Algorithms, vol. 3, pages 3-16, CompBioNets 2004.

[11] A. C. Siepel, “An algorithm to find all sorting reversals,” in Proc. 6th Ann. Int’l Conf. Comput. Mol. Biol. (RECOMB 2002), pp. 281–290. ACM Press, New York, 2002.

[12] Swenson, K.M., Badr, G., Sankoff, D.: "Listing all sorting reversals in quadratic time.", in: Singh, M. (ed.) WABI 2010. LNCS, vol. 6293, Springer, Heidelberg, pp. 102–110, 2010.

[13] A. Bergeron, C. Chauve, T. Hartman, and K. Saint-Onge, "On the properties of sequences of reversals that sort a signed permutation.", in Proc. JOBIM (2002), pp. 99–108, 2002.

[14] M. D. V. Braga, Exploring the solution space of sorting by reversals when analyzing genome rearrangements, PhD thesis, France, 2009.

[15] V. Diekert, and G. Rozenberg, The Book of Traces, World Scientific, Singapore, 1995.

[16] C. Baudet and Z. Dias, "An improved algorithm to enumerate all traces that sort a signed permutation by reversals", in Proc. ACM Symposium on Applied Computing, pp. 1521–1525, 2010.

[17] Amritanjali and G. Sahoo, "Exploring the Solution Space of Sorting By Reversals: A New Approach", International Journal of Information Technology, vol. 3(2), 2013.

[18] S. Berard , A. Bergeron and C. Chauve, “Conserved structures in evolution scenarios”, RCG 2004, Lecture Notes in Bioinformatics, vol. 3388, 1–15, 2005.

[19] A. Bergeron, J Mixtacki and J Stoye, “Common Intervals and Sorting by Reversals: A Marriage of Necessity”, Bioinformatics, vol. 18(2), pp. 554-563, 2002.

[20] T. Uno and M. Yaguira, “Fast algorithms to enumerate all common intervals of two permutations”, Algorithmica, vol. 26(2), pp. 290-309, 2000.

[21] M. D. V. Braga, C. Gautier and M.-F. Sagot, "An asymmetric approach to preserve common intervals while sorting by reversals", submitted to Algorithms for Molecular Biology, 2009.

[22] Diekmann Y., Sagot M.F. and Tannier E., "Evolution under reversals: parsimony and conservation of common intervals", IEEE/ACM Transactions on Computational Biology and Bioinformatics, vol. 4 (2), 301–309 (A preliminary version appeared in COCOON 2005, Lecture Notes in Computer Science, vol. 3595, (42–51).

[23] 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, vol. 3, 348–356, 2008 (A preliminary version appeared in ISBRA 2007, Lecture Notes in Bioinformatics vol. 4463, 293–304).

[24] S. Hannenhalli and P. Pevzner, "Transforming men into mice (polynomial algorithm for genomic distance problem)", In Proceedings of the IEEE 36th Annual Symposium on Foundations of Computer Science, pages 581-592, 1995.

[25] S. Berard, A. Bergeron, C. Chauve and C. Paul, "Perfect sorting by reversals is not always difficult", IEEE/ACM Transactions on Computational Biology and Bioinformatics, Vol. 4 (1), 4–16, 2007.

APPENDEX

DATA SOURCES

1. GROMM TOOL

   http://grimm.ucsd.edu/GRIMM

2. SoRT^2 TOOL

   http://genome.cs.nthu.edu.tw/SORT2

3. BaobabLuna Java Framework

   http://pbil.univlyon1.fr/software/luna

4. BaobabLuna Tutorials

   http://pbil.univlyon1.fr/software/luna/doc/luna-doc.pdf

5. UniMoG

   http://bibiserv.techfak.uni-bielefeld.de/dcj