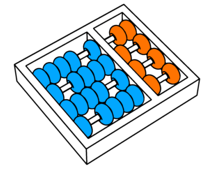
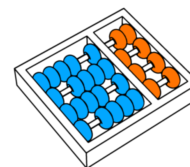


Gustavo Rodrigues Galvão

“Uma Ferramenta de Auditoria para Algoritmos de Rearranjo de Genomas”

CAMPINAS
2012



Universidade Estadual de Campinas
Instituto de Computação

Gustavo Rodrigues Galvão

"Uma Ferramenta de Auditoria para Algoritmos de Rearranjo de Genomas"

Orientador(a): **Prof. Dr. Zanoni Dias**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA POR GUSTAVO RODRIGUES GALVÃO, SOB ORIENTAÇÃO DE PROF. DR. ZANONI DIAS.

Assinatura do Orientador(a)

CAMPINAS
2012

FICHA CATALOGRÁFICA ELABORADA POR
ANA REGINA MACHADO - CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA - UNICAMP

	Galvão, Gustavo Rodrigues, 1988-
G139f	Uma ferramenta de auditoria para algoritmos de rearranjo de genomas / Gustavo Rodrigues Galvão. – Campinas, SP : [s.n.], 2012.
	Orientador: Zanoni Dias. Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.
	1. Biologia computacional. 2. Algoritmos aproximados. 3. Teoria da computação. I. Dias, Zanoni,1975-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

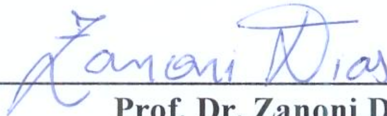
Título em inglês: An audit tool for genome rearrangement algorithms
Palavras-chave em inglês:
Computational biology
Approximation algorithms
Computer theory
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Zanoni Dias [Orientador]
Maria Emília Machado Telles Walter
João Meidanis
Data de defesa: 20-12-2012
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 20 de Dezembro de 2012, pela Banca examinadora composta pelos Professores Doutores:

Profª. Drª. Maria Emília Machado Telles Walter
CIC / UNB

Prof. Dr. João Meidanis
IC / UNICAMP



Prof. Dr. Zanoni Dias
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Uma Ferramenta de Auditoria para Algoritmos de Rearranjo de Genomas

Gustavo Rodrigues Galvão[1]

20 de Dezembro de 2012

Banca Examinadora:

- Prof. Dr. Zanoni Dias (Orientador)
- Profa. Dra. Maria Emília Machado Telles Walter
  Departamento de Ciência da Computação – CIC – UnB
- Prof. Dr. João Meidanis
  Instituto de Computação – Unicamp
- Prof. Dr. Nalvo Franco de Almeida Junior (Suplente)
  Faculdade de Computação – UFMS
- Prof. Dr. Guilherme Pimentel Telles (Suplente)
  Instituto de Computação – Unicamp

[1]Suporte financeiro de: Bolsa do CNPq (processo 135212/2010-3) 08/2010 – 01/2011 e Bolsa da CAPES (processo 01-P-01965/12) 06/2012 – 07/2012.

Resumo

Ao longo da evolução, mutações globais podem alterar a ordem dos genes de um genoma. Tais mutações são chamadas de eventos de rearranjo. Em Rearranjo de Genomas, estimamos a distância evolutiva entre dois genomas calculando-se a distância de rearranjo entre eles, que é o tamanho da menor sequência de eventos de rearranjo que transforma um genoma no outro. Representando genomas como permutações, nas quais os genes aparecem como elementos, a distância de rearranjo pode ser obtida resolvendo-se o problema combinatório de ordenar uma permutação utilizando o menor número de eventos de rearranjo. Este problema, que é referido como Problema da Ordenação por Rearranjo, varia de acordo com os tipos de eventos de rearranjo considerados.

Nesta dissertação, focamos nosso estudo em dois tipos de eventos: reversões e transposições. Variações do Problema da Ordenação por Rearranjo que consideram esses eventos têm se mostrado difíceis de serem resolvidas otimamente, por isso a maior parte dos algoritmos propostos – os quais denominamos genericamente por algoritmos de rearranjo de genomas – são aproximados e é esperado que os próximos avanços ocorram nesse sentido. Em razão disso, desenvolvemos uma ferramenta que avalia as respostas desses algoritmos. Para ilustrar sua aplicação, nós a utilizamos para avaliar as respostas de 16 algoritmos de rearranjo de genomas aproximados relativos a 6 variações do Problema da Ordenação por Rearranjo.

Além da ferramenta, este trabalho traz outras contribuições. Desenvolvemos um algoritmo exato para calcular distâncias de rearranjo que é mais eficiente em termos de uso de memória do que qualquer outro algoritmo que encontramos na literatura. Apresentamos conjecturas que dizem respeito à forma como as distâncias de rearranjo se distribuem. Validamos conjecturas referentes ao diâmetro, que é o maior valor alcançável pela distância de rearranjo entre uma permutação qualquer e a identidade considerando-se todas as permutações com o mesmo número de elementos. Apresentamos demonstrações formais para o fator de aproximação de alguns dos algoritmos avaliados. Por fim, mostramos que o fator de aproximação de 7 dos 16 algoritmos avaliados não podem ser melhorados, o que contradiz algumas hipóteses levantadas na literatura, e conjecturamos que o fator de aproximação de outros 6 algoritmos também não podem.

Abstract

During evolution, global mutations may modify the gene order in a genome and such mutations are called rearrangement events. In Genome Rearrangements, we estimate the evolutionary distance between two genomes by computing the rearrangement distance between them, which is the length of the shortest sequence of rearrangement events that transforms one genome into the other. Representing genomes as permutations, in which genes appear as elements, the rearrangement distance can be obtained by solving the combinatorial problem of sorting a permutation using a minimum number of rearrangement events. This problem is referred to as Rearrangement Sorting Problem and varies accordingly to the types of rearrangement events considered.

In this dissertation, we focus on two types of rearrangement events: reversals and transpositions. Variants of Rearrangement Sorting Problem involving these events have been shown to be difficult to solve optimally, therefore most of the proposed algorithms – which we denominate generically as genome rearrangement algorithms – are approximations, which have been the expected direction to follow. For this reason, we developed a tool that evaluates the results of these algorithms. To illustrate its application, we used it to evaluate the results of 16 genome rearrangement algorithms regarding 6 variants of Rearrangement Sorting Problem.

Besides this tool, we developed an exact algorithm for computing rearrangement distances that is more efficient in terms of memory than any algorithm we have found in literature. Additionally, we presented conjectures on how the rearrangement distance are distributed and validated them regarding their diameter, which is the greatest value that the rearrangement distance between a permutation and the identity can reach considering all permutations with the same number of elements. Moreover, we presented formal proofs on the approximation ratio of some of the evaluated algorithms and showed that the approximation ratio of 7 out of the 16 evaluated algorithms cannot be improved, which contradicts some hypothesis raised in literature. Lastly, we conjectured that the approximation ratio of another 6 algorithms also cannot be improved.

Agradecimentos

À minha família, em especial aos meus pais, Fátima e Claudio, pelo apoio e amor incondicionais.

À minha namorada, Ticiane Todo, não só pelo apoio e incentivo, mas também pela paciência durante todo esse tempo.

Ao meu orientador, Zanoni Dias, pela oportunidade e confiança em mim depositada, pela orientação e colaboração durante o desenvolvimento desta dissertação e, acima de tudo, pela incansável disposição para ajudar-me.

A todos os meus amigos e amigas, que sempre estiveram ao meu lado.

À Scopus Tecnologia e aos meus antigos chefes, Leila e Fabio, por terem feito o possível para que eu pudesse conciliar o meu emprego com o Mestrado.

Ao Felipe Rodrigues da Silva e ao João Carlos Setubal por terem possibilitado que parte dos cálculos das distâncias de rearranjo fosse realizado utilizando computadores da Embrapa e do Virginia Bioinformatics Institute.

Ao CNPq e à CAPES, pelo apoio financeiro.

Finalmente, gostaria de dedicar este trabalho à minha filha, Sofia.

Sumário

Resumo ix

Abstract xi

Agradecimentos xiii

1 Introdução 1

1.1 Conceitos Básicos 4

1.1.1 Genomas como Permutações 4

1.1.2 Algoritmos Aproximados 6

1.2 Variações do Problema da Ordenação por Rearranjo 7

1.2.1 Variações que Consideram Apenas Reversões 7

1.2.2 Variações que Consideram Apenas Transposições 9

1.2.3 Variações que Consideram Reversões e Transposições 9

1.3 Motivação e Objetivos 10

1.4 Organização da Dissertação 12

2 A Ferramenta de Auditoria 13

2.1 Algoritmo para Calcular as Distâncias de Rearranjo 13

2.1.1 Funções de Indexação e Desindexação 15

2.1.2 O Algoritmo 18

2.1.3 Implementação e Discussão 22

2.2 Cálculo das Distâncias de Rearranjo 26

2.2.1 Distribuição das Distâncias de Rearranjo 26

2.2.2 Base de Dados de Distâncias de Rearranjo 38

2.3 GRAAu: *Genome Rearrangement Algorithm Auditor* 40

3 Algoritmos de Rearranjo de Genomas 47

3.1 Problema da Ordenação por Reversões 47

3.2 Problema da Ordenação por Reversões de Prefixo 51

3.3 Problema da Ordenação por Reversões de Prefixo com Sinal 57
3.4 Problema da Ordenação por Reversões Curtas 65
3.5 Problema da Ordenação por Transposições 71
3.6 Problema da Ordenação por Transposições de Prefixo 78

4 Auditoria dos Algoritmos de Rearranjo de Genomas 83
4.1 Problema da Ordenação por Reversões 83
4.2 Problema da Ordenação por Reversões de Prefixo 85
4.3 Problema da Ordenação por Reversões de Prefixo com Sinal 90
4.4 Problema da Ordenação por Reversões Curtas 94
4.5 Problema da Ordenação por Transposições 100
4.6 Problema da Ordenação por Transposições de Prefixo 104

5 Considerações Finais 113
5.1 Publicações 116
5.2 Trabalhos Futuros 116

Referências Bibliográficas 118

Lista de Tabelas

1.1 Estado da arte das variações do Problema da Ordenação por Rearranjo consideradas nesta dissertação. 11

2.1 Modelos de rearranjo e valores de n considerados para o cálculo das distâncias de rearranjo. 26
2.2 Valores exatos e limitantes conhecidos para os diâmetros de S_n e $S_n^{\pm}$. 27
2.3 Distribuição da distância de reversão de prefixo com sinal em $S_n^{\pm}$. 28
2.4 Distribuição da distância de reversão com sinal e transposição em $S_n^{\pm}$. 29
2.5 Distribuição da distância de reversão com sinal, transposição e transreversão do tipo A em $S_n^{\pm}$. 29
2.6 Distribuição da distância de transposição e transreversão do tipo A e do tipo B em $S_n^{\pm}$. 30
2.7 Distribuição da distância de reversão em S_n. 30
2.8 Distribuição da distância de transposição em S_n. 31
2.9 Distribuição da distância de reversão e transposição em S_n. 31
2.10 Distribuição da distância de reversão de prefixo em S_n. 32
2.11 Distribuição da distância de transposição de prefixo em S_n. 33
2.12 Distribuição da distância de reversão de prefixo e transposição de prefixo em S_n. 33
2.13 Distribuição da distância de reversão curta em S_n. 34
2.14 Diâmetro, diâmetro transversal e longevidade de $S_n^{\pm}$. 35
2.15 Diâmetro, diâmetro transversal e longevidade de S_n. 36
2.16 Diâmetro, diâmetro transversal e longevidade de S_n. 37
2.17 Conjecturas para os valores de $D_M(n)$, $T_M(n)$ e $L_M(n)$. 38

4.1 Resultado da auditoria do Algoritmo 11. 84
4.2 Resultado da auditoria do Algoritmo 13. 84
4.3 Resultado da auditoria do Algoritmo 14. 86
4.4 Resultado da auditoria do Algoritmo 15. 86
4.5 Resultado da auditoria do Algoritmo 16. 87
4.6 Resultado da auditoria do Algoritmo 17. 91

4.7 Resultado da auditoria do Algoritmo 18. 91
4.8 Resultado da auditoria do Algoritmo 19. 92
4.9 Resultado da auditoria do Algoritmo 20. 95
4.10 Resultado da auditoria do Algoritmo 23. 95
4.11 Resultado da auditoria do Algoritmo 25. 96
4.12 Resultado da auditoria do Algoritmo 26. 101
4.13 Resultado da auditoria do Algoritmo 27. 101
4.14 Resultado da auditoria do Algoritmo 28. 102
4.15 Permutações π de tamanho $3m+1$, $m \in \{5, 6, 7\}$, tais que $\frac{p(\pi)}{d_t(\pi)} = \frac{3m}{m+1}$. Note que $d_t(\pi) \geq \frac{b_t(\pi)}{3} \geq m+1$. 104
4.16 Resultado da auditoria do Algoritmo 29. 104
4.17 Resultado da auditoria do Algoritmo 30. 105

Lista de Figuras

2.1 Ganho de desempenho médio e máximo observado ao executarmos a implementação de 32 bits 10 vezes para cada par (n, t), sendo n o tamanho das permutações e t o número de *threads*, sempre considerando um modelo de rearranjo composto apenas por reversões. 25

3.1 Grafo de *breakpoints* $G_{rp}(\pi)$ da permutação $\pi = (4\ 2\ 1\ 3)$. 54

4.1 Comparação entre os algoritmos 11 e 13 com base nos resultados produzidos pelo GRAAu. 85
4.2 Comparação entre os algoritmos 14, 15 e 16 com base nos resultados produzidos pelo GRAAu. 88
4.3 Grafos de *breakpoints* das permutações que são produzidas pelo Algoritmo 15 ao ordenar a permutação $\pi = (1\ 7\ 8\ 2\ 4\ 3\ 9\ 5\ 6)$. 90
4.4 Comparação entre os algoritmos 17, 18 e 19 com base nos resultados produzidos pelo GRAAu. 93
4.5 Comparação entre os algoritmos 20, 23 e 25 com base nos resultados produzidos pelo GRAAu. 97
4.6 Comparação entre os algoritmos 26, 27 e 28 com base nos resultados produzidos pelo GRAAu. 103
4.7 Comparação entre os algoritmos 29 e 30 com base nos resultados produzidos pelo GRAAu. 106

Capítulo 1

Introdução

Um dos desafios modernos da Ciência é tentar desvendar como ocorreu a evolução das espécies. Dado que a evolução pode ser vista como um processo de ramificação, em que novas espécies surgem a partir de modificações ocorridas nos organismos vivos, o estudo da história evolutiva de um grupo de espécies é comumente realizado por meio da construção de árvores, nas quais os nós representam as espécies e as arestas representam relações evolutivas. As relações evolutivas entre um determinado grupo de espécies são chamadas de filogenia, enquanto que as árvores são chamadas de árvores filogenéticas.

Devido a impossibilidade de descobrir a história evolutiva verdadeira, os cientistas propõem métodos de inferí-la. Tal inferência pode ser feita baseando-se em diferentes tipos de dados, desde geográficos, ecológicos e morfológicos até moleculares, como o DNA. Dados moleculares possuem a grande vantagem de serem exatos e reprodutíveis, pelo menos dentro de erros experimentais, além de serem fáceis de obter [34, Capítulo 12]. Cada nucleotídeo de uma sequência de DNA é, por si próprio, uma característica bem definida, enquanto que um dado morfológico, por exemplo, precisa ser codificado em uma característica, o que acarreta problemas de interpretação, discretização e etc [34, Capítulo 12].

Dentre os métodos de inferência filogenética existentes baseados em dados moleculares, iremos focar naqueles que se baseiam na noção de distância evolutiva entre as espécies. Tais métodos constroem a árvore filogenética relativa a um grupo de espécies da seguinte maneira: primeiramente, a distância evolutiva entre as espécies é determinada de tal maneira a gerar uma matriz de distâncias M tal que a entrada $M_{i,j}$ contém a distância evolutiva entre as espécies i e j; depois, a árvore filogenética é computada a partir dessa matriz utilizando-se um algoritmo específico, como o *Neighbor-Joining* [53]. Para mais detalhes a respeito de outros métodos de inferência filogenética, incluindo uma análise detalhada de cada um deles, recomendamos a leitura do livro de Gascuel [34, Capítulo 12].

Olhando para a maneira como uma árvore filogenética é construída por esses métodos, podemos notar que um ponto chave está na abordagem utilizada para determinar a distância evolu-

tiva entre as espécies. Uma abordagem bem aceita é a dada pela área de Rearranjo de Genomas. Pesquisadores dessa área propõem que o grau de divergência entre espécies diferentes é fruto da ocorrência de mutações globais, chamadas de eventos de rearranjo, que modificam a ordem de segmentos do genoma conservados entre elas. Tal proposição é fundamentada em inúmeros trabalhos que evidenciam a ocorrência de eventos de rearranjo, dentre os quais destacamos os trabalhos pioneiros de Dobzhansky e Sturtevant [17, 56].

Sob a perspectiva de Rearranjo de Genomas, a distância evolutiva entre duas espécies é estimada pela quantidade de eventos de rearranjo que levou o genoma de uma a se transformar no genoma da outra. Como é impossível precisar esta quantidade, lança-se mão do princípio da máxima parcimônia, o qual sugere que o número mínimo de eventos fornece uma boa aproximação da realidade. Assim, o tamanho da menor sequência de eventos de rearranjo que transforma o genoma de uma espécie no genoma da outra, denominado como distância de rearranjo, é adotado como a distância evolutiva entre elas.

O genoma de um indivíduo é constituído por cromossomos, os quais podem ser representados, simplificadamente, como um conjunto ordenado de genes orientados, sendo essa orientação determinada pela localização do gene na fita dupla de DNA. Um evento de rearranjo conservativo ocorre quando um ou mais cromossomos do genoma são quebrados em segmentos e unidos de tal forma que o conjunto inicial de genes permanece inalterado, mas a ordem ou a orientação de um subconjunto desses genes se altera. São exemplos de eventos de rearranjo conservativos:

- **Reversões**: ocorrem quando a ordem e a orientação de uma sequência contígua de genes de um cromossomo é invertida.
- **Transposições**: ocorrem quando uma sequência contígua de genes de um cromossomo é destacada e inserida em outra posição no mesmo cromossomo. A ordem e a orientação dos genes não são alteradas.
- **Translocações**: ocorrem quando uma sequência contígua de genes de um cromossomo é trocada com uma sequência contígua de genes de outro cromossomo. Uma translocação pode ou não inverter a ordem e a orientação dos genes.
- **Fusões**: ocorrem quando dois cromossomos se unem para formar um só cromossomo.
- **Fissões**: ocorrem quando um cromossomo se divide em dois cromossomos.

Um evento não-conservativo não possui a propriedade de preservar o conjunto inicial de genes. São exemplos de eventos de rearranjo não-conservativos:

- **Remoções**: ocorrem quando um conjunto de genes desaparece do genoma.
- **Inserções**: ocorrem quando um conjunto de genes aparece no genoma.

- **Duplicações**: ocorrem quando um conjunto de genes é duplicado no genoma.

Dependendo do grupo de espécies para o qual estamos querendo construir a árvore filogenética, podemos considerar um conjunto de eventos de rearranjo específico a fim de determinarmos a distância de rearranjo. Por exemplo, se estamos considerando um grupo de espécies cujos genomas são unicromossomais, isto é, formados por apenas um cromossomo, não há razão para considerarmos translocações. Em contrapartida, se estamos considerando espécies mais complexas, cujos genomas são formados por múltiplos cromossomos, então passa a fazer sentido que as consideremos. O conjunto de eventos de rearranjo escolhido para determinarmos a distância de rearranjo é chamado de modelo de rearranjo.

Não existe nenhuma evidência biológica concreta sobre como identificar um bom modelo de rearranjo. Em razão disso, estudos envolvendo o cálculo da distância de rearranjo vem ocorrendo de maneira exploratória. Inicialmente, os pesquisadores consideraram modelos de rearranjo simples, compostos por apenas um tipo de evento de rearranjo. Recentemente, os pesquisadores tem considerado modelos mais sofisticados, não somente pelo fato de serem compostos por mais do que um tipo de evento de rearranjo, mas também pelo fato de pesos serem atribuídos a cada um deles. Esses pesos servem para modelar a frequência com que tais eventos ocorrem: quanto maior o peso, menor a frequência.

Neste trabalho, nos atemos a modelos de rearranjo compostos por eventos de rearranjo conservativos, doravante denominados também como operações, que ocorrem em um único cromossomo, ou seja, nos restringimos aos modelos de rearranjo compostos por reversões ou transposições. Apesar desta restrição nos limitar a uma visão não tão abrangente do processo evolutivo, ela é motivada, em primeiro lugar, pela necessidade de definirmos um escopo realista para esta dissertação de mestrado e, em segundo lugar, pelo fato dessas operações estarem entre as mais estudadas na literatura. Além disso, consideramos que os eventos de rearranjo possuem pesos iguais.

Em adição às restrições anteriores, que implicam em um estudo focado em genomas unicromossomais, assumimos que os cromossomos são lineares e partilham um mesmo conjunto de genes tal que este conjunto não possui genes duplicados. A razão dessa simplificação é que ela nos permite modelar um genoma como uma permutação de número inteiros, na qual cada número representa um gene do genoma. Se a orientação dos genes é conhecida[1], cada número possui um sinal, positivo ou negativo, indicando a orientação do gene que ele representa. Ainda que essa simplificação nos desvie da realidade biológica, ressaltamos que permutações constituem a representação clássica de genomas na área de Rearranjo de Genomas. Além disso, existem exemplos biológicos para os quais permutações monstram-se ser um modelo adequado, tal como descrito por Kececioglu e Sankoff [43].

[1]Existem dois métodos de se obter a ordem dos genes em um genoma: mapeamento físico e sequenciamento. Somente o segundo método fornece a informação sobre orientação dos genes [61].

Modelando genomas como permutações, podemos enunciar o *Problema da Distância de Rearranjo* da seguinte forma: dados dois genomas, calcule o número mínimo de operações pertencentes a um determinado modelo de rearranjo necessárias para transformar a permutação que representa um genoma na permutação que representa o outro. Como transformar uma permutação em outra por meio da aplicação de um número mínimo de operações é equivalente a ordenar uma permutação pela aplicação das mesmas operações (vide próxima seção), o Problema da Distância de Rearranjo é comumente reduzido ao chamado *Problema da Ordenação por Rearranjo*. Algoritmos desenvolvidos com intuito de resolver algum desses dois problemas são os que denominamos por *Algoritmos de Rearranjo de Genomas*.

1.1 Conceitos Básicos

Esta seção é destinada à formalização dos conceitos básicos concernentes a esta dissertação. A maior parte do conteúdo apresentado é baseado no livro de Fertin e colegas [23], que consideramos ser a principal referência na área de Rearranjo de Genomas.

1.1.1 Genomas como Permutações

Na literatura de Rearranjo do Genomas, podemos encontrar muitas abordagens para modelar genomas. Basicamente, o que difere uma abordagem de outra são as suposições feitas sobre os genomas. Assumindo que os genomas consistem-se de um único cromossomo linear, partilham o mesmo conjunto de genes e não contêm genes duplicados, podemos modelá-los como permutações de números inteiros (com ou sem sinal), nas quais cada número representa um gene do genoma.

Uma permutação sem sinal π é uma função bijetora sobre o conjunto $\{1, 2, \ldots, n\}$. Uma notação clássica para permutações sem sinal é a chamada notação em duas colunas

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi_1 & \pi_2 & \ldots & \pi_n \end{pmatrix},$$

$\pi_i \in \{1, 2, \ldots, n\}$ para $1 \leq i \leq n$. Esta notação indica que a imagem do elemento $i \in \{1, 2, \ldots, n\}$ é π_i ou, em outras palavras, $\pi(1) = \pi_1$, $\pi(2) = \pi_2$, $\ldots$, $\pi(n) = \pi_n$. Em Rearranjo de Genomas, geralmente é utilizada a notação chamada de notação em uma coluna $\pi = (\pi_1\ \pi_2 \ldots \pi_n)$, a qual possui o mesmo significado. Nós dizemos que a permutação π possui tamanho n.

Uma permutação com sinal π é uma função bijetora sobre o conjunto $\{-n, \ldots, -2, -1, 1, 2, \ldots, n\}$. Sua notação em duas colunas é dada por

$$\pi = \begin{pmatrix} -n & \ldots & -2 & -1 & 1 & 2 & \ldots & n \\ -\pi_n & \ldots & -\pi_2 & -\pi_1 & \pi_1 & \pi_2 & \ldots & \pi_n \end{pmatrix},$$

$\pi_i \in \{1, 2, \ldots, n\}$ para $1 \leq i \leq n$. Utilizando a notação em uma coluna, ocultamos o mapeamento dos elementos negativos uma vez que $\pi(-i) = -\pi(i)$ para todo $i \in \{1, 2, \ldots, n\}$ e também escrevemos $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_n)$. Por abuso de notação, também dizemos que a permutação π possui tamanho n. Desse modo, podemos tratar permutações com ou sem sinal de maneira unificada, fazendo diferenciações quando necessário.

A composição entre duas permutações π e σ, representada por $\pi \circ \sigma$, resulta na permutação γ tal que $\gamma(i) = \pi(\sigma(i))$, ou seja, $\gamma = (\pi_{\sigma_1}\ \pi_{\sigma_2}\ \ldots\ \pi_{\sigma_n})$. Essa operação permite-nos modelar não somente genomas como permutações, mas também eventos de rearranjo, de tal forma que um evento de rearranjo ρ transforma um genoma π no genoma $\pi \circ \rho$. Na Seção 1.2, definiremos formalmente os eventos de rearranjo estudados nesta dissertação.

A operação de composição induz uma estrutura de grupo sobre o conjunto formado por todas as permutações de um determinado tamanho uma vez que as seguintes propriedades são satisfeitas:

- **Fechamento:** dadas duas permutações π e $\sigma \in S_n$ ($S_n^{\pm}$), nós temos que $\pi \circ \sigma \in S_n$ ($S_n^{\pm}$);
- **Associatividade:** dadas três permutações π, σ e γ, nós temos que $\pi \circ (\sigma \circ \gamma) = (\pi \circ \sigma) \circ \gamma$;
- **Existência do elemento neutro:** a permutação identidade $\iota = (1\ 2\ \ldots\ n)$ é o elemento neutro, pois dada uma permutação $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_n)$, nós temos que $\pi \circ \iota = \pi$;
- **Existência do elemento inverso:** dada uma permutação $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_n)$, seja π^{-1} uma permutação tal que $\pi^{-1}_{\pi_i} = i$ para $1 \leq i \leq n$. Nós chamamos π^{-1} de permutação inversa de π e temos que tanto $\pi \circ \pi^{-1}$ quanto $\pi^{-1} \circ \pi$ resultam na permutação identidade.

O grupo formado pelo conjunto de todas as $n!$ permutações sem sinal de tamanho n juntamente com a operação de composição é chamado de grupo simétrico e é denotado por S_n. O grupo formado pelo conjunto de todas as $n!2^n$ permutações com sinal de tamanho n juntamente com a operação de composição é chamado de grupo simétrico com sinal e é denotado por $S_n^{\pm}$.

Seja G um subconjunto de S_n ($S_n^{\pm}$) tal que qualquer permutação de S_n ($S_n^{\pm}$) pode ser obtida pela composição das permutações de G. As permutações pertencentes a G são chamadas de geradores de S_n ($S_n^{\pm}$). Ademais, se $\pi^{-1} \in G$ para toda permutação $\pi \in G$, então nós dizemos que G é simétrico. Um modelo de rearranjo M é um conjunto formado por geradores de S_n ($S_n^{\pm}$) tal que todos os geradores pertencentes a M modelam eventos de rearranjo conservativos e M é simétrico.

O Problema da Distância de Rearranjo pode ser definido formalmente da seguinte forma: dadas duas permutações π e σ, determine o número mínimo de geradores $\rho_1, \rho_2, \ldots, \rho_t$ pertencentes a um modelo de rearranjo M tal que $\pi \circ \rho_1 \circ \rho_2 \circ \cdots \circ \rho_t = \sigma$. Este número corresponde à distância de rearranjo entre as permutações π e σ com respeito a M, denotada por $d_M(\pi, \sigma)$. Como M é simétrico, nós temos que $d_M(\pi, \sigma) = d_M(\sigma, \pi)$ pois

$$\pi \circ \rho_1 \circ \rho_2 \circ \cdots \circ \rho_t = \sigma \iff \sigma \circ \rho_t^{-1} \circ \rho_{t-1}^{-1} \circ \cdots \circ \rho_1^{-1} = \pi.$$

A distância de rearranjo entre as permutações π e σ com respeito a M é igual à distância de rearranjo entre as permutações $\sigma^{-1} \circ \pi$ e ι com respeito a M uma vez que

$$\pi \circ \rho_1 \circ \rho_2 \circ \cdots \circ \rho_t = \sigma \iff (\sigma^{-1} \circ \pi) \circ \rho_1 \circ \rho_2 \circ \cdots \circ \rho_t = \iota.$$

Isso significa que podemos restringir a análise do Problema da Distância de Rearranjo ao caso em que uma das permutações é a permutação identidade. Sendo assim, definimos a distância de rearranjo de uma permutação π com respeito a M, denotada por $d_M(\pi)$, como sendo a distância de rearranjo entre π e ι com respeito a M. Ou seja, $d_M(\pi) = d_M(\pi, \iota)$.

Ordenar uma permutação significa transformá-la na permutação identidade por meio da composição de geradores. Desse modo, o Problema da Ordenação por Rearranjo é formalmente definido da seguinte maneira: dada uma permutação γ, determine a menor sequência de geradores $\rho_1, \rho_2, \ldots, \rho_k$ pertencentes a um modelo de rearranjo M tal que $\gamma \circ \rho_1 \circ \rho_2 \circ \cdots \circ \rho_k = \iota$.

Agora podemos visualizar porque o Problema da Distância de Rearranjo é fortemente vinculado ao Problema da Ordenação por Rearranjo: claramente, ao resolvermos este, teremos também resolvido aquele. Note que o inverso não é verdadeiro, pois a solução para o Problema da Distância de Rearranjo nos diz apenas qual é o tamanho da menor sequência de geradores. Apesar disso, a literatura costuma considerar esses problemas como equivalentes, dando maior ênfase ao Problema da Ordenação por Rearranjo. Nós iremos adotar essa abordagem por razões de simplicidade.

Ao definirmos o Problema da Ordenação por Rearranjo, fixamos um modelo de rearranjo. Isso significa que, dependendo do modelo de rearranjo considerado, temos uma variação diferente do Problema da Ordenação por Rearranjo. Se, ao contrário, tivéssemos definido o modelo de rearranjo como um parâmetro de entrada, então teríamos formulado uma espécie de Problema da Ordenação por Rearranjo genérico, o qual aceitaria como caso específico qualquer variação do Problema da Ordenação por Rearranjo. Todavia, graças ao resultado de Even e Goldreich [21], sabemos que o problema genérico é NP-Difícil.

1.1.2 Algoritmos Aproximados

Algoritmos aproximados ou aproximativos são algoritmos que não necessariamente produzem uma solução ótima para um determinado problema de otimização, mas produzem uma solução com uma certa garantia de aproximação em relação à solução ótima. A motivação para estudá-los advém principalmente da provável impossibilidade de encontrarmos algoritmos polinomiais ótimos para problemas de otimização NP-Difíceis.

De acordo com Cormen e colegas [14, Capítulo 35], um algoritmo aproximado para um problema de otimização possui um fator de aproximação $\alpha(n)$ se, para qualquer instância de

entrada de tamanho n, o custo C da solução produzida pelo algoritmo está dentro de um fator $\alpha(n)$ da solução ótima C^*. Isto é, se o problema é de maximização, então $\frac{C^*}{C} \leq \alpha(n)$. Ao contrário, se o problema é de minimização, então $\frac{C}{C^*} \leq \alpha(n)$.

Se um algoritmo aproximado possui fator de aproximação $\alpha(n)$, nós dizemos que ele é um algoritmo $\alpha(n)$-aproximado. No nosso caso, seja A um algoritmo aproximado para uma determinada variação do Problema da Ordenação por Rearranjo tal que, para cada permutação $\pi \in S_n$ ($S_n^{\pm}$), ele retorna uma sequência de geradores (pertencentes a um modelo de rearranjo M) de tamanho $A(\pi)$. Então, A é dito ser um algoritmo $\alpha(n)$-aproximado se $\frac{A(\pi)}{d_M(\pi)} \leq \alpha(n)$ para toda permutação $\pi \in S_n$ ($S_n^{\pm}$).

Uma pergunta natural a se fazer a respeito de um algoritmo aproximado é: o fator de aproximação desse algoritmo pode ser melhorado? Para responder essa pergunta positivamente, é necessário demonstrar que o algoritmo possui um fator de aproximação melhor. Por outro lado, para respondê-la negativamente, é necessário demonstrar que existe uma infinidade de instâncias de entrada para as quais a razão entre a resposta do algoritmo e a solução ótima (ou o inverso, no caso do problema ser de maximização) é igual ao fator de aproximação do algoritmo.

Caso seja possível responder aquela pergunta negativamente para um determinado algoritmo aproximado, nós dizemos que o fator de aproximação desse algoritmo é justo. Além de servirem para mostrar que não é possível melhorar o fator de aproximação de um algoritmo aproximado, as instâncias de entrada que demonstram a justeza do fator de aproximação possuem outra finalidade, como destaca Vazirani [59, Capítulo 1]. Elas oferecem uma visão crítica de como o algoritmo funciona e frequentemente levam à ideias que dão origem a algoritmos com fatores de aproximações melhores.

1.2 Variações do Problema da Ordenação por Rearranjo

Nesta seção, faremos uma breve revisão bibliográfica das variações do Problema da Ordenação por Rearranjo que iremos tratar mais detalhadamente nesta dissertação. Embora algumas dessas variações não tenham sido originalmente motivadas pela área de Rearranjo de Genomas, tendo inclusive pouca relevância do ponto de vista biológico, elas foram absorvidas pela área e atualmente são apresentadas como problemas de Rearranjo de Genomas.

1.2.1 Variações que Consideram Apenas Reversões

Uma reversão $r(i, j)$, $1 \leq i < j \leq n$, transforma a permutação sem sinal $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_{i-1}\ \underline{\pi_i\ \pi_{i+1}\ \ldots\ \pi_{j-1}\ \pi_j}\ \pi_{j+1}\ \ldots\ \pi_n)$ na permutação sem sinal $\pi \circ r(i, j) = (\pi_1\ \pi_2\ \ldots\ \pi_{i-1}\ \underline{\pi_j\ \pi_{j-1}\ \ldots\ \pi_{i+1}\ \pi_i}\ \pi_{j+1}\ \ldots\ \pi_n)$, ou seja, $r(i, j)$ é a permutação sem sinal $(1\ 2\ \ldots\ i-1\ j\ j-1\ \ldots\ i+1\ i\ j+1\ \ldots\ n)$. A variação do Problema da Ordenação por Rearranjo que considera um

modelo de rearranjo composto apenas por reversões é chamada de Problema da Ordenação por Reversões.

Caprara [9] demostrou que o Problema da Ordenação por Reversões é NP-Difícil. Watterson e colegas [65] foram os primeiros a apresentar um algoritmo para este problema, sendo este um algoritmo $\frac{n-1}{2}$-aproximado. Kececioglu e Sankoff [43] foram os primeiros a apresentar um algoritmo de aproximação com fator constante. Eles apresentaram um algoritmo 2-aproximado. Posteriormente, Bafna e Pevzner [2] construíram um algoritmo de aproximação com fator 1.75. A próxima evolução foi dada por Christie [12] ao desenvolver um algoritmo com fator de aproximação 1.5. Por fim, o melhor resultado conhecido até o presente momento foi apresentado por Berman, Hannenhalli e Karpinski [6]. Eles desenvolveram um algoritmo 1.375-aproximado.

Uma reversão com sinal $rs(i,j)$, $1 \leq i \leq j \leq n$, transforma a permutação com sinal $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_{i-1}\ \underline{\pi_i\ \pi_{i+1}\ \ldots\ \pi_{j-1}\ \pi_j}\ \pi_{j+1}\ \ldots\ \pi_n)$ na permutação com sinal $\pi \circ rs(i,\,j) = (\pi_1\ \pi_2\ \ldots\ \pi_{i-1}\ \underline{-\pi_j\ -\pi_{j-1}\ \ldots\ -\pi_{i+1}\ -\pi_i}\ \pi_{j+1}\ \ldots\ \pi_n)$, ou seja, $rs(i,j)$ é a permutação com sinal $(1\ 2\ \ldots\ i-1\ -j\ -(j-1)\ \ldots\ -(i+1)\ -i\ j+1\ \ldots\ n)$. A variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto apenas por reversões com sinal é chamada de Problema da Ordenação por Reversões com Sinal.

Hannanhalli e Pevzner [38] foram os primeiros a resolver o Problema da Ordenação por Reversões com Sinal, apresentando um algoritmo ótimo que executa em tempo $O(n^4)$. Alguns refinamentos foram feitos neste algoritmo ao longo dos anos até que Tannier, Bergeron e Sagot [57] apresentaram um algoritmo $O(n^{\frac{3}{2}}\sqrt{\log n})$, sendo este o melhor resultado conhecido. Bader, Moret e Yan [1] mostraram como calcular a distância de reversão com sinal em tempo linear.

Uma reversão de prefixo $rp(i)$, $2 \leq i \leq n$, é equivalente à reversão $r(1,i)$. A variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto apenas por reversões de prefixo é chamada de Problema da Ordenação por Reversões de Prefixo. Este problema também é conhecido como Problema da Ordenação de Panquecas e foi introduzido por Dweighter [18]. Bulteau, Fertin e Rusu [7] provaram que ele é NP-Difícil e o melhor algoritmo aproximado conhecido para resolvê-lo foi apresentado por Fischer e Ginzinger [24]. Tal algoritmo é uma 2-aproximação.

Uma reversão de prefixo com sinal $rps(i)$, $1 \leq i \leq n$, é equivalente à reversão com sinal $rs(1,i)$. A variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto apenas por reversões de prefixo com sinal é chamada de Problema da Ordenação por Reversões de Prefixo com Sinal. Este problema também é conhecido como Problema da Ordenação de Panquecas Queimadas e foi introduzido por Cohen e Blum [13]. O melhor algoritmo aproximado conhecido para resolvê-lo é uma 2-aproximação apresentada por eles [13]. A complexidade deste problema não é conhecida.

Uma reversão $r(i,j)$ é dita uma k-reversão se $j - i \leq k$. Uma reversão curta é uma k-reversão tal que $k = 3$. A variação do Problema da Ordenação por Rearranjo que conside-

ra um modelo de rearranjo composto apenas por reversões curtas é chamada de Problema da Ordenação por Reversões Curtas. A complexidade deste problema é desconhecida e o melhor algoritmo aproximado conhecido para resolvê-lo é uma 2-aproximação apresentada por Heath e Vergara [40, 60].

1.2.2 Variações que Consideram Apenas Transposições

Um transposição $t(i,\ j,\ k)$, $1 \leq i < j < k \leq n+1$, transforma a permutação $\pi = (\pi_1 \ldots \pi_{i-1}\ \underline{\pi_i \ldots \pi_{j-1}}\ \underline{\pi_j \ldots \pi_{k-1}}\ \pi_k \ldots \pi_n)$ na permutação $\pi \circ t(i,\ j,\ k) = (\pi_1 \ldots \pi_{i-1}\ \underline{\pi_j \ldots \pi_{k-1}}\ \underline{\pi_i \ldots \pi_{j-1}}\ \pi_k \ldots \pi_n)$, ou seja, $t(i,\ j,\ k)$ é a permutação $(1\ 2 \ldots i-1\ j\ j-1 \ldots k-1\ i \ldots j-1\ k \ldots n)$. A variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto apenas transposições é chamada de Problema da Ordenação por Transposições.

O primeiro resultado relevante obtido para o Problema da Ordenação por Transposições é devido a Bafna e Pevzner [3]. Eles apresentaram um algoritmo aproximativo com fator de aproximação 1.5. Elias e Hartman [19] produziram uma algoritmo de aproximação com fator de aproximação 1.375. Recentemente, Bulteau, Fertin e Rusu [8] demonstraram que este problema é NP-Difícil.

Uma transposição de prefixo $tp(i, j)$, $2 \leq i < j \leq n+1$, é equivalente à transposição $t(1, i, j)$. A variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto apenas transposições de prefixo é chamada de Problema da Ordenação por Transposições de Prefixo. Este problema foi introduzido por Dias e Meidanis [16] e o melhor algoritmo aproximado conhecido para resolvê-lo é uma 2-aproximação apresentada por eles [16]. A complexidade deste problema é desconhecida.

1.2.3 Variações que Consideram Reversões e Transposições

Walter, Dias e Meidanis [62] foram os primeiros a considerar tanto a variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto apenas por reversões com sinal e transposições quanto a variação que considera um modelo de rearranjo composto apenas por reversões e transposições. A primeira é chamada de Problema da Ordenação por Reversões com Sinal e Transposições e a segunda é chamada de Problema da Ordenação por Reversões e Transposições. Para este problema, eles apresentaram um algoritmo 3-aproximado, enquanto que, para aquele, um algoritmo 2-aproximado. Rahman, Shatabda e Hasan [52] apresentaram um algoritmo $2k$-aproximado para o Problema da Ordenação por Reversões e Transposições, sendo k o fator de aproximação do algoritmo de decomposição de ciclos. A complexidade de ambos problemas é desconhecida.

Sharmina e colegas [55] estudaram a variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto apenas por reversões de prefixo e transposições

de prefixo, chamada de Problema da Ordenação por Reversões de Prefixo e Transposições de Prefixo ou de Problema da Ordenação de Panquecas com Duas Espátulas. Eles desenvolveram um algoritmo 3-aproximado para resolvê-la. A complexidade do problema não é conhecida.

Alguns pesquisadores também consideraram um outro tipo de operação, chamada reversão-+transposição ou transreversão. Ela é similar a uma transposição, exceto que um dos segmentos transpostos da permutação é invertido. Um transreversão com sinal do tipo A $trs_a(i, j, k)$, $1 \leq i < j < k \leq n+1$, transforma a permutação com sinal π na permutação com sinal $\pi \circ tr_a(i, j, k) = \pi \circ rs(i, j-1) \circ t(i, j, k)$. Uma transreversão com sinal do tipo B $trs_b(i, j, k)$, $1 \leq i < j < k \leq n+1$, transforma a permutação com sinal π na permutação com sinal $\pi \circ trs_b(i, j, k) = \pi \circ rs(j, k-1) \circ t(i, j, k)$.

Gu *et al.* [36] desenvolveram um algoritmo 2-aproximado para a variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto por reversões com sinal, transposições e transreversões com sinal do tipo A, a qual chamamos de Problema da Ordenação por Reversões com Sinal, Transposições e Transreversões com Sinal do Tipo A. Hartman e Sharan [39] desenvolveram um algoritmo 1.5-aproximado para a variação do Problema da Ordenação por Rearranjo que considera um modelo de rearranjo composto por transposições e transreversões com sinal dos tipos A e B, a qual chamamos de Problema da Ordenação por Transposições e Transreversões com Sinal do Tipo A e do Tipo B. A complexidade de ambos os problemas é desconhecida.

1.3 Motivação e Objetivos

Esta breve revisão de algumas variações do Problema da Ordenação por Rearranjo evidencia que, de modo geral, elas são computacionalmente difíceis de serem resolvidas. Consequentemente, a maior parte dos algoritmos propostos para resolvê-las são aproximados (ver Tabela 1.1) e é esperado que os próximos avanços ocorram nesse sentido. Este fato nos motivou a construir uma ferramenta para avaliar as respostas desse tipo de algoritmo. Tal avaliação é importante tanto para efeitos de comparação (por exemplo, elucidar qual algoritmo produz, em média, os melhores resultados) quanto para efeitos de validação (por exemplo, verificar a corretude da implementação de um algoritmo ou averiguar a justeza do fator de aproximação).

Um método empregado na literatura para avaliar as respostas de um algoritmo de rearranjo de genomas que não produz respostas garantidamente ótimas é a que chamamos de *auditoria*. A auditoria consiste-se em calcular a distância de rearranjo de todas as permutações que possuem até um certo tamanho, compará-las com as respostas retornadas pelo algoritmo para essas permutações e produzir estatísticas que meçam a qualidade das respostas. Esse método foi empregado, por exemplo, por Walter, Dias e Meidanis [63], Walter e colegas [64], Benoît-Gagné e Hamel [5] e Dias e Dias [15].

O processo de auditoria consome uma quantidade considerável de tempo e esforço, em

Tabela 1.1: Estado da arte das variações do Problema da Ordenação por Rearranjo consideradas nesta dissertação.

Modelo de Rearranjo	NP-Difícil	Melhor Solução Teórica
Reversões com Sinal	Não	Algoritmo exato $O(n^{\frac{3}{2}}\sqrt{\log n})$
Reversões	Sim	Algoritmo 1.375-aproximado
Transposições	Sim	Algoritmo 1.375-aproximado
Reversões de Prefixo	Sim	Algoritmo 2-aproximado
Transposições e Transreversões do Tipo A e do Tipo B	?	Algoritmo 1.5-aproximado
Reversões Curtas	?	Algoritmo 2-aproximado
Transposições de Prefixo	?	Algoritmo 2-aproximado
Reversões com Sinal e Transposições	?	Algoritmo 2-aproximado
Reversões com Sinal, Transposições e Transreversões do Tipo A	?	Algoritmo 2-aproximado
Reversões e Transposições	?	Algoritmo $2k$-aproximado
Reversões de Prefixo e Transposições de Prefixo	?	Algoritmo 3-aproximado

sua maioria gastos implementando um algoritmo que calcula as distâncias de rearranjo e, em seguida, computando as distâncias de rearranjo. A ferramenta que nós desenvolvemos mitiga o tempo e o esforço gastos para realizar ambas as tarefas, tornando assim o processo de auditoria mais rápido e menos laborioso. Além de facilitar esse processo, nós esperamos que a ferramenta possa promover uma certa padronização da avaliação de algoritmos de rearranjo de genomas. A razão disso é que, se todos algoritmos forem avaliados utilizando um mesmo método (ou um conjunto de métodos), tornar-se-ia muito mais fácil compará-los, sendo inclusive suficiente avaliá-los apenas uma vez.

Um aspecto teórico não muito explorado nos trabalhos que apresentam algoritmos de rearranjo de genomas aproximados é a justeza do fator de aproximação. Esse fato, somado à necessidade de ilustrar a aplicação da ferramenta, nos motivou a implementar uma série de algoritmos de rearranjo de genomas aproximados propostos na literatura, auditar essas implementações utilizando a ferramenta e discutir a justeza do fator de aproximação desses algoritmos com base nas estatísticas obtidas.

Deste modo, esta dissertação foi realizada tendo-se em vista dois objetivos principais:

- Construir uma ferramenta de auditoria para algoritmos de rearranjo de genomas;
- Avaliar algoritmos de rearranjo de genomas aproximados com a ferramenta e discutir a justeza do fator de aproximação desses algoritmos com base nos resultados obtidos, trazendo para o centro da discussão um tópico pouco abordado na literatura de Rearranjo

de Genomas.

1.4 Organização da Dissertação

O restante desta dissertação está organizada da seguinte maneira. O Capítulo 2 apresenta a ferramenta de auditoria de algoritmos de rearranjo de genomas, dando todos os detalhes de como ela foi desenvolvida. O Capítulo 3 descreve os algoritmos de rearranjo de genomas que auditamos com a ferramenta. O Capítulo 4 apresenta e analisa os resultados que obtivemos ao auditar os algoritmos descritos no Capítulo 3. Por fim, o Capítulo 5 conclui esta dissertação.

Capítulo 2

A Ferramenta de Auditoria

Neste capítulo, iremos apresentar a ferramenta de auditoria de algoritmos de rearranjo de genomas. Nós a chamamos de GRAAu, que é um acrônimo para o nome em inglês *Genome Rearrangement Algorithm Auditor*.

A auditoria é baseada na comparação da distâncias de rearranjo de todas permutações que possuem até um determinado tamanho com as respostas produzidas pelo algoritmo para essas permutações. A Seção 2.1 descreve o algoritmo que desenvolvemos para calcular as distâncias de rearranjo e compara-o com outros algoritmos apresentados na literatura. Graças a esse algoritmo, conseguimos calcular as distribuições das distâncias de rearranjo das permutações em S_n e em $S_n^{\pm}$ para valores de n nunca calculados anteriormente. A Seção 2.2.1 apresenta essas distribuições juntamente com algumas conjecturas que visam capturar o modo como as distâncias de rearranjo estão distribuídas. Por fim, a Seção 2.3 descreve como o GRAAu funciona.

2.1 Algoritmo para Calcular as Distâncias de Rearranjo

Como estamos interessados em variações do Problema de Ordenação que são NP-Difíceis ou que não possuem solução polinomial exata conhecida, podemos pensar basicamente em dois métodos para calcular a distância de rearranjo de todas as permutações de S_n ($S_n^{\pm}$) com respeito a um modelo M:

1. Realizar uma busca em largura em S_n ($S_n^{\pm}$): inicialize uma fila de permutações F com a permutação identidade e atribua a ela distância 0. Enquanto F não estiver vazia, remova a permutação π de F, reporte π e $d_M(\pi)$ e, para cada $\rho \in M$, faça
   - compute a permutação σ tal que $\sigma = \pi \circ \rho$;
   - adicione σ a F caso ela não tenha sido adicionada anteriormente, atribuindo a ela distância $d_M(\pi) + 1$.

2. Desenvolver um algoritmo exato, que executa em tempo exponencial, para a variação do Problema de Ordenação e executar esse algoritmo para todas permutações de S_n ($S_n^{\pm}$). Exemplos de algoritmos exatos propostos na literatura para resolver variações do Problema da Ordenação que são NP-Difíceis dividem-se principalmente entre algoritmos de *branch-and-bound* e algoritmos de programação linear inteira.

Nós acabamos optando pelo método 1 devido, principalmente, aos seguintes motivos:

- **Simplicidade e Corretude.** Implementar uma busca em largura em S_n ($S_n^{\pm}$) é mais simples e, portanto, menos suscetível a erros do que desenvolver um algoritmo de *branch-and-bound* ou de programação linear inteira. Como exemplo, podemos citar o algoritmo de *branch-and-bound* desenvolvido por Kececioglu e Sankoff [43]: a implementação possuía aproxidamente 9.500 linhas de código e, como iremos mostrar na próxima seção, não calculava as distâncias corretamente.

- **Flexibilidade.** Levando em conta que estávamos interessados em considerar diversas variações do Problema da Ordenação por Rearranjo, utilizar um método facilmente adaptável era algo altamente desejável. Este não é o caso do método 2, pois algoritmos de *branch-and-bound* e de programação linear inteira tendem a ser bastante específicos uma vez que dependem de limitantes ou de restrições que são particulares de cada problema.

A nossa maior preocupação ao implementar o método 1 foi como otimizar o uso de memória. Dada a simplicidade do método, o principal ponto passível de otimização nesse sentido era a maneira como as permutações seriam representadas. Ao olharmos para a definição de uma permutação, a primeira representação que nos vem em mente é um vetor de número inteiros. Se cada elemento do vetor utiliza b bits, então uma permutação de tamanho n utiliza nb bits para ser representada. Haveria maneira de utilizar menos bits?

Imagine que fosse possível representar uma permutação em S_n como um número natural pertencente a $\{0, 1, \ldots, n! - 1\}$ e uma permutação em $S_n^{\pm}$ como um número natural pertencente a $\{0, 1, \ldots, 2^n n! - 1\}$. Então, uma permutação em S_n utilizaria $\lceil \lg n! - 1 \rceil$ bits para ser representada e uma permutação em $S_n^{\pm}$ utilizaria $\lceil \lg 2^n n! - 1 \rceil$ bits para ser representada. Supondo que $n \leq 2^{b'}$, $b' \leq b$, nós temos que

$$\lceil \lg n! - 1 \rceil < \lg n! < n \lg n \leq nb'$$

e que

$$\lceil \lg 2^n n! - 1 \rceil < \lg 2^n n! = \lg 2^n + \lg n! = n + \lg n! < n + n \lg n \leq n + nb' = n(1 + b').$$

Isso significa que uma permutação em S_n utilizaria não mais do que nb' bits e uma permutação em $S_n^{\pm}$ utilizaria não mais do que $n(1 + b')$ bits.

Note que a diferença é mais notável na prática. Por exemplo, uma permutação de tamanho 10 pode ser representada como um vetor de *bytes*, portanto utilizando 80 bits. Por outro lado, nós temos que $\lceil \lg 10! - 1 \rceil = 22$ e que $\lceil \lg 2^{10} 10! - 1 \rceil = 32$, logo uma permutação em S_{10} poderia ser representada por um número inteiro sem sinal de 22 bits e uma permutação em $S_{10}^{\pm}$ poderia ser representada por um número inteiro sem sinal de 32 bits. Supondo que os tamanhos dos tipos de dados que representam números inteiros são múltiplos de 16 bits (como ocorre na linguagem C, que foi a utilizada na implementação), tanto uma permutação em S_{10} quanto uma permutação em $S_{10}^{\pm}$ poderia ser representada por um número inteiro sem sinal de 32 bits.

Uma vez convencidos de que representar permutações como número naturais é mais econômico do que representar permutações como vetores de números inteiros, nos restava descobrir como fazê-lo. Mais do que isso, era necessário descobrir tanto uma maneira de mapear uma permutação a um número natural quanto uma maneira de mapear um número natural a uma permutação devido a necessidade de se realizar operações de composição entre permutações. Em outras palavras, precisávamos definir funções bijetoras

$$\begin{gathered} f : S_n \to \{0, 1, \ldots, n! - 1\}, \\ f^{-1} : \{0, 1, \ldots, n! - 1\} \to S_n, \\ g : S_n^{\pm} \to \{0, 1, \ldots, 2^n n! - 1\} \text{ e} \\ g^{-1} : \{0, 1, \ldots, 2^n n! - 1\} \to S_n^{\pm}. \end{gathered}$$

Nós dizemos que as funções f e g *indexam* uma permutação e por isso são referidas como *funções de indexação*. Por conseguinte, as funções f^{-1} e g^{-1} são referidas como *funções de desindexação*.

2.1.1 Funções de Indexação e Desindexação

Antes de prosseguirmos, ressaltamos que não encontramos na literatura algoritmos para computar as funções g e g^{-1}. Por essa razão, desenvolvemos uma maneira de definir g e g^{-1} a partir de f e f^{-1} respectivamente. Primeiro, definimos o *vetor de sinais* $vs(\pi)$ = $[vs(\pi_1), vs(\pi_2), \ldots, vs(\pi_n)]$ de uma permutação $\pi \in S_n^{\pm}$ tal que $vs(\pi_i) = 1$ se $\pi_i < 0$ e $vs(\pi_i) = 0$ se $\pi_i > 0$ para todo $1 \leq i \leq n$. Segundo, definimos a *permutação modular* $m(\pi)$ de uma permutação $\pi \in S_n^{\pm}$ tal que $m(\pi) = (|\pi_1|\ |\pi_2| \ldots |\pi_n|)$. Desse modo, é fácil notar que a permutação $\pi \in S_n^{\pm}$ pode ser representada univocamente pela permutação modular $m(\pi) \in S_n$ e pelo vetor de sinais $vs(\pi)$.

Considerando que o vetor de sinais de uma permutação $\pi \in S_n^{\pm}$ pode ser visto como o número binário $vs(\pi_1)vs(\pi_2)\ldots vs(\pi_n)$, nós definimos a função bijetora $h : V_n \to \{0, 1 \ldots, 2^n - 1\}$ tal que $h(vs(\pi)) = \sum_{i=1}^{n} 2^{i-1} vs(\pi_i)$, sendo $V_n = \{vs(\pi) : \pi \in S_n^{\pm}\}$. Como h foi definida analogamente à forma como convertemos números naturais da base binária para a base decimal, é fácil perceber que a função $h^{-1} : \{0, 1 \ldots, 2^n - 1\} \to V_n$ é definida analogamente à forma como convertemos números naturais da base decimal para a base binária. Assim, definimos g a

partir de f e h tal que $g(\pi) = 2^n f(m(\pi)) + h(vs(\pi))$ para qualquer $\pi \in S_n^{\pm}$. Ademais, g^{-1} está bem definida a partir de f^{-1} e h^{-1} pois, dado $g(\pi)$ de uma permutação $\pi \in S_n^{\pm}$, nós temos que $m(\pi) = f^{-1}(\frac{g(\pi)-r}{2^n})$ e que $vs(\pi) = h^{-1}(r)$, sendo $r = h(vs(\pi)) = g(\pi) \bmod 2^n$.

Segundo Myrvold e Ruskey [51], a abordagem tradicional para o problema de se construir as funções f e f^{-1} é primeiro definir uma ordem para as permutações e depois definir tais funções relativas a essa ordem. Ainda segundo eles [51], construções mais sofisticadas que levam em conta a ordem lexicográfica calculam o *vetor de inversões*, também conhecido como *código de Lehmer* [47], como passo intermediário. Para deixar claro, uma função de indexação é construída levando-se em conta a ordem lexicográfica se, para toda permutação $\pi \in S_n$, $f(\pi)$ é igual ao número de permutações em S_n que são lexicograficamente menores do que π.

Dada uma permutação $\pi \in S_n$, seu vetor de inversões é definido por

$$L(\pi) = [l(\pi_1), l(\pi_2), \ldots, l(\pi_n)] \text{ tal que } l(\pi_i) = |\{\pi_j : \pi_j < \pi_i \text{ e } j > i\}|.$$

O vetor de inversões de uma permutação a representa univocamente [51], de tal forma que podemos obter π a partir de $L(\pi)$ tomando π_i como o $(l(\pi_i) + 1)$-ésimo elemento do conjunto $\{1, 2, \ldots, n\} \setminus \{\pi_1, \pi_2, \ldots, \pi_{i-1}\}$.

Note que tanto um algoritmo ingênuo para computar $L(\pi)$ a partir de π quanto um algoritmo ingênuo para computar π a partir de $L(\pi)$ executam em tempo quadrático no tamanho de π. Myrvold e Ruskey [51] deram indicações de implementações mais eficientes, porém afirmaram não conhecer algoritmos que executam em tempo linear. Ademais, eles também afirmaram que construções que não levam em conta a ordem lexicográfica não apresentam nenhuma vantagem sobre os que levam. Por essa razão, eles utilizaram uma abordagem diferente daquela considerada como tradicional para construir as funções f e f^{-1}, apresentando algoritmos para computá-las em tempo linear no tamanho das permutações.

Mais recentemente, Mares e Straka [49] construíram funções f e f^{-1} considerando a ordem lexicográfica e mostraram como computá-las em tempo linear. Dado que a ordem induzida pela função de indexação não é relevante para o nosso caso, nós optamos pelos algoritmos desenvolvidos por Myrvold e Ruskey [51] para computar as funções de indexação e de desindexação por acharmos o algoritmo destes mais simples.

Abaixo, os algoritmos *Indexa* e *Desindexa* computam as funções f e f^{-1} respectivamente, enquanto os algoritmos *IndexaComSinal* e *DesindexaComSinal* computam as funções g e g^{-1} respectivamente. A rotina `troca`(γ_i, γ_j) permuta os elementos γ_i e γ_j da permutação γ.

O Algoritmo *IndexaRecursivo* possui complexidade de tempo expressa pela relação de recorrência $T(n) = T(n-1) + O(1)$, com $T(1) = O(1)$, portanto ele roda em tempo $O(n)$. Como computar a permutação inversa de uma permutação em S_n pode ser feito em tempo $O(n)$, o Algoritmo *Indexa* também roda em tempo $O(n)$. O Algoritmo *DesindexaRecursivo* possui complexidade de tempo expressa pela relação de recorrência $T(n) = T(n-1) + O(1)$, com $T(0) = O(1)$, portanto ele roda em tempo $O(n)$. Como instanciar a permutação identidade

Algorithm 1: IndexaRecursivo

Entrada: Um número inteiro não-negativo x e duas permutações π e $\sigma \in S_n$.
Saída: Um número inteiro pertencente a $\{0, 1, \ldots, n! - 1\}$.

1 **se** $x = 1$ **então**
2 | **retorna** 0;
3 **fim**
4 $s \leftarrow \pi_{x-1}$;
5 `troca`$(\pi_{x-1}, \pi_{\sigma_{x-1}})$;
6 `troca`(σ_s, σ_{x-1});
7 **retorna** $s + x\cdot$`IndexaRecursivo`$(x-1, \pi, \sigma)$;

Algorithm 2: Indexa

Entrada: Uma permutação $\pi \in S_n$.
Saída: Um número inteiro pertencente a $\{0, 1, \ldots, n! - 1\}$.

1 Compute a permutação inversa de π, isto é, π^{-1};
2 **retorna** `IndexaRecursivo`(n, π, π^{-1});

Algorithm 3: DesindexaRecursivo

Entrada: Dois números inteiros não-negativos x e y e uma permutação $\pi \in S_n$.
Saída: Nenhuma.

1 **se** $x > 0$ **então**
2 | $r \leftarrow y \bmod x$;
3 | `troca`(π_{x-1}, π_r);
4 | `DesindexaRecursivo`$(x-1, \lfloor \frac{y}{x} \rfloor, \pi)$;
5 **fim**

Algorithm 4: Desindexa

Entrada: Um número inteiro $r \in \{0, 1, \ldots, n! - 1\}$.
Saída: Uma permutação $\pi \in S_n$.

1 $\pi \leftarrow \iota$;
2 `DesindexaRecursivo`(n, r, π);
3 **retorna** π;

de tamanho n pode ser feito em tempo $O(n)$, o algoritmo *Desindexa* também roda em tempo $O(n)$.

Quanto ao Algoritmo *IndexaComSinal*, nós temos que a linha 1 é executada em tempo $O(1)$, o laço **para** das linhas 2-6 é executado em tempo $O(n)$ e as linhas 7 e 8 são executadas em tempo

Algorithm 5: IndexaComSinal

Entrada: Uma permutação $\pi \in S_n^{\pm}$ e um vetor de números inteiros $v = [v_1, v_2, \ldots, v_n, v_{n+1}]$ tal que $v_i = 2^{i-1}$ para $1 \leq i \leq n+1$.
Saída: Um número inteiro pertencente a $\{0, 1, \ldots, 2^n n! - 1\}$.

```
   // Calcula em s o valor de h(vs(π))
1  s ← 0;
2  para i = 1 até n faça
3  |  se π_i < 0 então
4  |  |  s ← s + v_i;
5  |  fim
6  fim
7  Compute a permutação modular m(π) de π;
8  retorna v_{n+1}·Indexa(m(π)) + s;
```

Algorithm 6: DesindexaComSinal

Entrada: O número inteiro $r \in \{0, 1, \ldots, 2^n n! - 1\}$ e o número inteiro $p = 2^n$.
Saída: Uma permutação $\pi \in S_n^{\pm}$.

```
1  a ← r mod p;
2  r ← (r − a) div p;
3  π ← Desindexa(r);
4  para i = n até 1 faça
5  |  se r mod 2 = 1 então
6  |  |  π_i ← −π_i;
7  |  fim
8  |  r ← r div 2;
9  fim
10 retorna π;
```

$O(n)$. Quanto ao Algoritmo *DesindexaComSinal*, nós temos que as linhas 1 e 2 são executadas em tempo $O(1)$, a linha 3 é executada em tempo $O(n)$ e o laço **para** das linhas 4-9 é executado em tempo $O(n)$. Portanto, os algoritmos *IndexaComSinal* e *DesindexaComSinal* também rodam em tempo $O(n)$.

2.1.2 O Algoritmo

Dado que apresentamos um algoritmo que transforma uma permutação em um número natural e também apresentamos um algoritmo que realiza a transformação inversa, apresentamos abaixo o Algoritmo *TodasDistancias*, o qual calcula as distâncias de rearranjo de todas as permutações de

S_n com respeito a um modelo M. Omitimos a descrição do algoritmo que calcula as distâncias de rearranjo de todas as permutações de $S_n^{\pm}$ com respeito a um modelo M pelo fato dele ser trivialmente derivável a partir do Algoritmo *TodasDistancias*.

O Algoritmo *TodasDistancias* recebe dois parâmetros de entrada: o número inteiro n relativo ao tamanho das permutações e um modelo de rearranjo M. Ele então cria dois vetores, F e D, de tamanho $|S_n|$, inicializando D com -1. O vetor F mantém uma fila das permutações que vão sendo geradas, isto é, nós temos que $F_i = f(\pi)$ tal que π foi a i-ésima permutação gerada durante a execução do algoritmo. O vetor D armazena as distâncias das permutações que já foram geradas, isto é, nós temos que $D_i = d_M(f^{-1}(i-1))$ caso a permutação $f^{-1}(i-1)$ já tenha sido gerada ou $D_i = -1$ caso contrário. Assim, além de armazenar as distâncias, o vetor D também indica se uma permutação já foi gerada ou não. As variáveis $proximo$ e $ultimo$ gerenciam a fila F: a variável $proximo$ indica a posição da fila contendo a próxima permutação a ser processada e a variável $ultimo$ indica o fim da fila, isto é, indica a posição em que uma nova permutação será inserida em F. Ao término da execução, o algoritmo retorna o vetor D.

Inicialmente, a permutação identidade é inserida em F (linha 4), sua distância (zero) é armazenada em D (linha 5) e a variável $ultimo$ é inicializada com valor 2 (linha 7), ou seja, $ultimo = |F| + 1$, sendo $|F|$ o número de permutações armazenadas em F. Cada vez que uma permutação π é inserida em F, o vetor D é atualizado com o valor de $d_M(\pi)$ e a variável $ultimo$ é incrementada de uma unidade (linhas $16-18$). Logo, $ultimo = |F| + 1$ é uma invariante do laço **enquanto** assim como também é o fato de que D armazena as distâncias das permutações em F. Pela definição, M é formado por geradores de S_n, então todas as permutações de S_n serão necessariamente geradas e inseridas em F. Ademais, cada permutação é inserida uma única vez em F. Esses fatos implicam que teremos $ultimo = |S_n| + 1$ quando $|F| = |S_n|$, fazendo com que o laço **enquanto** termine. Nesse momento, todas as permutações de S_n terão sido inseridas em F e, portanto, suas distâncias terão sido armazenadas em D.

Quanto à complexidade de tempo do Algoritmo *TodasDistancias*, temos que as linhas 1 e 2 são executas em tempo $O(|S_n|)$, as linhas 4-7, 9, 10, 12 e 15-17 são executadas em tempo $O(1)$ e as linhas 3, 11 e 14 são executadas em tempo $O(n)$. Desse modo, nós concluímos que o laço **para** das linhas 13-20 é executado em tempo $O(n|M|)$ e, portanto, o laço **enquanto** das linhas 8-21 é executado em tempo $O(n|M||S_n|)$, sendo esta a complexidade de tempo do algoritmo. Sabendo que $|M|$ é polinomial, nós temos que o Algoritmo *TodasDistancias* executa em tempo exponencial no tamanho das permutações, mas em tempo polinomial no número de permutações.

Para tentar compensar o fato do algoritmo possuir complexidade de tempo exponencial, nós desenvolvemos uma maneira de paralelizá-lo. A ideia que tivemos consiste-se basicamente em encapsular o laço **enquanto** das linhas 8-21 em uma *thread* e então criar múltiplas *threads*. A existência de múltiplas *threads* executando um mesmo trecho do algoritmo que possui dados compartilhados cria uma condição de corrida, isto é, o resultado final do algoritmo pode variar

Algorithm 7: TodasDistancias

Entrada: O número inteiro n relativo ao tamanho das permutações e o modelo M.
Saída: Vetor D contendo as distâncias de rearranjo de todas as permutações de S_n com respeito ao modelo M.

```
1  Crie dois vetores F e D de tamanho |S_n|;
2  Inicialize D de tal maneira que D_i = −1 para i ∈ {1, 2, . . ., |S_n|};
3  i ← Indexa(ι);
4  F_1 ← i;
5  D_{i+1} ← 0;
6  proximo ← 1;
7  ultimo ← 2;
8  enquanto ultimo ≤ |S_n| faça
9  |   i ← F_proximo;
10 |   d ← D_{i+1};
11 |   π ← Desindexa(i);
12 |   proximo ← proximo + 1;
13 |   para cada ρ ∈ M faça
14 |   |   i ← Indexa(π ∘ ρ);
15 |   |   se D_{i+1} = −1 então
16 |   |   |   F_ultimo ← i;
17 |   |   |   ultimo ← ultimo + 1;
18 |   |   |   D_{i+1} ← d + 1;
19 |   |   fim
20 |   fim
21 fim
22 retorna D;
```

ou erros podem ocorrer dependendo da ordem em que as *threads* forem executadas. No caso do laço **enquanto** das linhas 8-21, os dados compartilhados são os vetores D e F e as variáveis $proximo$ e $ultimo$.

Para garantir que as *threads* executem de uma maneira que o algoritmo produza o resultado esperado, torna-se necessário sincronizá-las. Isso pode ser feito por meio da utilização de semáforos binários. Um semáforo binário controla o acesso das threads a uma determinada região do algoritmo, garantindo que nenhuma *thread* possa executar aquela região enquanto uma outra a esteja executando. Desse modo, podemos contornar o problema da condição de corrida utilizando semáforos binários para isolar as regiões em que ocorrem acessos aos dados compartilhados.

A desvantagem da utilização de semáforos binários é que ela pode induzir um cenário em que apenas uma thread executa de fato, não fazendo valer na prática a ideia do paralelismo. Ob-

servando o Algoritmo *TodasDistancias*, concluímos que há uma boa chance de isso acontecer, pois o laço **enquanto** das linhas 8-21 contém praticamente somente acessos aos dados compartilhados. Por essa razão, nós desenvolvemos uma versão um pouco diferente do Algoritmo *TodasDistancias*, a qual chamamos de Algoritmo *TodasDistanciasParalelo*.

Algorithm 8: TodasDistanciasParalelo

Entrada: O número inteiro n relativo ao tamanho das permutações e o modelo M.
Saída: Vetor D contendo as distâncias de rearranjo de todas as permutações de S_n com respeito ao modelo M.

```
1  Crie dois vetores F e D de tamanho |S_n| e crie um vetor P de tamanho |M|;
2  Inicialize D de tal maneira que D_i = −1 para i ∈ {1, 2, ..., |S_n|};
3  i ← Indexa(ι);
4  F_1 ← i;
5  D_{i+1} ← 0;
6  proximo ← 1;
7  ultimo ← 2;
8  enquanto ultimo ≤ |S_n| faça
9      i ← F_proximo;
10     d ← D_{i+1};
11     π ← Desindexa(i);
12     proximo ← proximo + 1;
13     j ← 1;
14     para cada ρ ∈ M faça
15         P_j ← Indexa(π ∘ ρ);
16         j ← j + 1;
17     fim
18     para j = 1 até |M| faça
19         i ← P_j;
20         se D_{i+1} = −1 então
21             F_ultimo ← i;
22             ultimo ← ultimo + 1;
23             D_{i+1} ← d + 1;
24         fim
25     fim
26 fim
27 retorna D;
```

A diferença entre os algoritmos *TodasDistancias* e *TodasDistanciasParalelo* é que o segundo gera as permutações $\pi \circ \rho$ para todo $\rho \in M$ e as armazena em vetor P (laço **para** das linhas 14-17). Se cada *thread* possuir uma instância do vetor P, ou seja, se P não for compar-

tilhado, a região relativa ao laço **para** das linhas 14-17 não precisa ser isolada por um semáforo binário. Isso possibilita um nível de paralelismo maior, pois enquanto uma ou mais *threads* estão preenchendo suas instâncias do vetor P, outra *thread* pode estar acessando os dados compartilhados.

Infelizmente, apenas semáforos binários não resolvem completamente o problema da condição de corrida. Suponha que duas *threads* t_1 e t_2 estejam executando o Algoritmo *TodasDistanciasParalelo* e que os seguintes fatos tenham ocorrido:

1. Ao executar a linha 10, t_1 obteve o valor d_1 e t_2 obteve o valor d_2, $d_1 \neq d_2$;

2. Tanto t_1 quanto t_2 terminaram de executar o laço **para** das linhas 14-17 tal que t_1 preencheu o vetor P_1 e t_2 preencheu o vetor P_2;

3. Existe uma permutação γ tal que $f(\gamma) \in P_1$ e $f(\gamma) \in P_2$, mas $f(\gamma) \notin F$.

Supondo que isolamos o laço **para** das linhas 18-25 com um semáforo binário, apenas uma *thread* pode executá-lo por vez. Independentemente de qual seja a *thread* a executá-lo primeiro, note que a permutação γ será inserida na fila F. A questão é que se a primeira *thread* a executá-lo for t_1, então a distância atribuída a γ será $d_1 + 1$. Caso contrário, a distância atribuída a γ será $d_2 + 1$. Então, dependendo da ordem em que as *threads* executarem do laço **para** das linhas 18-25, a distância atribuída a γ será diferente. Logo, é preciso garantir que t_1 execute o laço primeiro caso $d_1 < d_2$ ou que t_2 execute o laço primeiro caso $d_2 < d_1$.

Seja a *distância momentânea* de uma *thread* o valor d que ela obteve ao executar a linha 10 do Algoritmo *TodasDistanciasParalelo*. No cenário hipotético descrito anteriormente, a distância momentânea de t_1 era d_1 e a de t_2 era d_2. Para garantir que as distâncias de rearranjo sejam calculadas corretamente, devemos evitar que uma *thread* com distância momentânea x execute o laço **para** das linhas 18-25 caso exista uma *thread* com distância momentânea y tal que $y < x$. Para isso, nós criamos uma variável global chamada $distMin$ que armazena o valor da menor distância momentânea dentre todas as *threads* em execução. Desse modo, antes de executar o laço **para** das linhas 18-25, uma *thread* com distância momentânea d sempre verifica se $d > distMin$. Em caso negativo, ela executa o laço (obviamente, respeitando o semáforo binário). Em caso positivo, ela fica esperando até que a verificação seja negativa.

2.1.3 Implementação e Discussão

A implementação do algoritmo descrito na seção anterior foi feita na linguagem C, utilizando a biblioteca *pthread* [4] para gerenciar as threads. O código fonte pode ser obtido acessando o seguinte endereço

http://mirza.ic.unicamp.br:8080/bioinfo/download/allPermutations.zip.

Nós implementamos duas versões do algoritmo descrito na seção anterior: em uma versão, nós representamos as permutações como número inteiros não-negativos de 32 bits; na outra, como número inteiros não-negativos de 64 bits. Portanto, a primeira consegue representar todas as permutações sem sinal de até 12 elementos e todas as permutações com sinal de até 10 elementos, enquanto a segunda consegue representar todas as permutações sem sinal de até 20 elementos e todas as permutações com sinal de até 16 elementos. Ambas as versões foram implementadas considerando diversos modelos de rearranjo, que serão detalhados na Seção 2.2.1. No total, cada uma delas possui aproximadamente 1.500 linhas de código.

Vergara [60] desenvolveu um algoritmo similar para calcular as distâncias de rearranjo de todas as permutações de S_n com respeito a um modelo M. Apesar desse algoritmo também utilizar a ideia de representar permutações como números inteiros, não podemos dizer que ele otimiza o uso de memória. A razão disso é que Vergara [60] diz ter indexado as permutações da seguinte maneira: primeiro ele ordena as permutações em S_n lexicograficamente e depois ele atribui um número inteiro a cada permutação considerando essa ordem. Para ordenar as permutações lexicograficamente, é necessário representá-las como vetores de números inteiros, portanto não há de fato uma economia de memória ao indexar as permutações desse modo.

Dias e Meidanis [16] computaram as distâncias de transposição de prefixo de todas as permutações sem sinal com até 11 elementos e afirmaram que o método utilizado por eles precisaria de aproximadamente 30GB de RAM para conseguir computar as distâncias de transposição de prefixo de todas as permutações sem sinal de tamanho 12. De forma semelhante, Walter, Dias e Meidanis [63] computaram as distâncias de transposição de todas as permutações sem sinal com até 11 elementos e afirmaram que o método utilizado por eles precisaria de aproxidamente 18GB de RAM para conseguir computar as distâncias de transposição de todas as permutações sem sinal de tamanho 12.

A versão de 32 bits implementada por nós precisa de aproximadamente 2.4GB de RAM para computar as distâncias de rearranjo de todas as permutações sem sinal com até 12 elementos com respeito a qualquer modelo de rearranjo. Portanto, nosso método utiliza aproxidamente 12 vezes menos memória do que o método utilizado por Dias e Meidanis [16] e aproximadamente 7 vezes menos memória do que o método utilizado por Walter, Dias e Meidanis [63].

Existem outras publicações [5, 15, 44, 64] onde as distâncias de rearranjo de todas as permutações sem sinal com respeito a um ou mais modelos de rearranjo foram calculadas, mas elas não fornecem informações sobre o uso de memória e, por isso, não podemos realizar comparações justas. Em todo caso, ressaltamos que nenhuma delas considerou permutações sem sinal com mais do que 11 elementos, o que nos leva a inferir que os métodos empregados por esses autores enfrentaram limitações de memória similares às enfrentadas pelos métodos de Dias e Meidanis [16] e Walter, Dias e Meidanis [63].

Em termos de desempenho, gostaríamos de discutir a questão da paralelização. O modo natural de medir o ganho de desempenho de um programa que pode ser executado em paralelo é

observar a razão $\frac{T(n)}{T}$, sendo $T(n)$ o tempo gasto para o programa terminar quando o executamos com n *threads* em paralelo (i.e. cada *thread* roda em núcleo ou em um processador diferente) e T o tempo gasto para o programa terminar quando o executamos apenas com 1 *thread*.

Para ilustrarmos o ganho de desempenho, nós executamos a implementação de 32 bits 10 vezes para cada par (n, t), sendo n o tamanho das permutações e t o número de *threads*, sempre considerando um modelo de rearranjo composto apenas por reversões. O código fonte foi compilado com a versão 4.5.0 do gcc e o programa resultante foi executado em um computador com 16 processadores Intel Xeon® E5520 rodando a 2.27GHz e com 64GB de RAM. A Figura 2.1 mostra o resultado obtido.

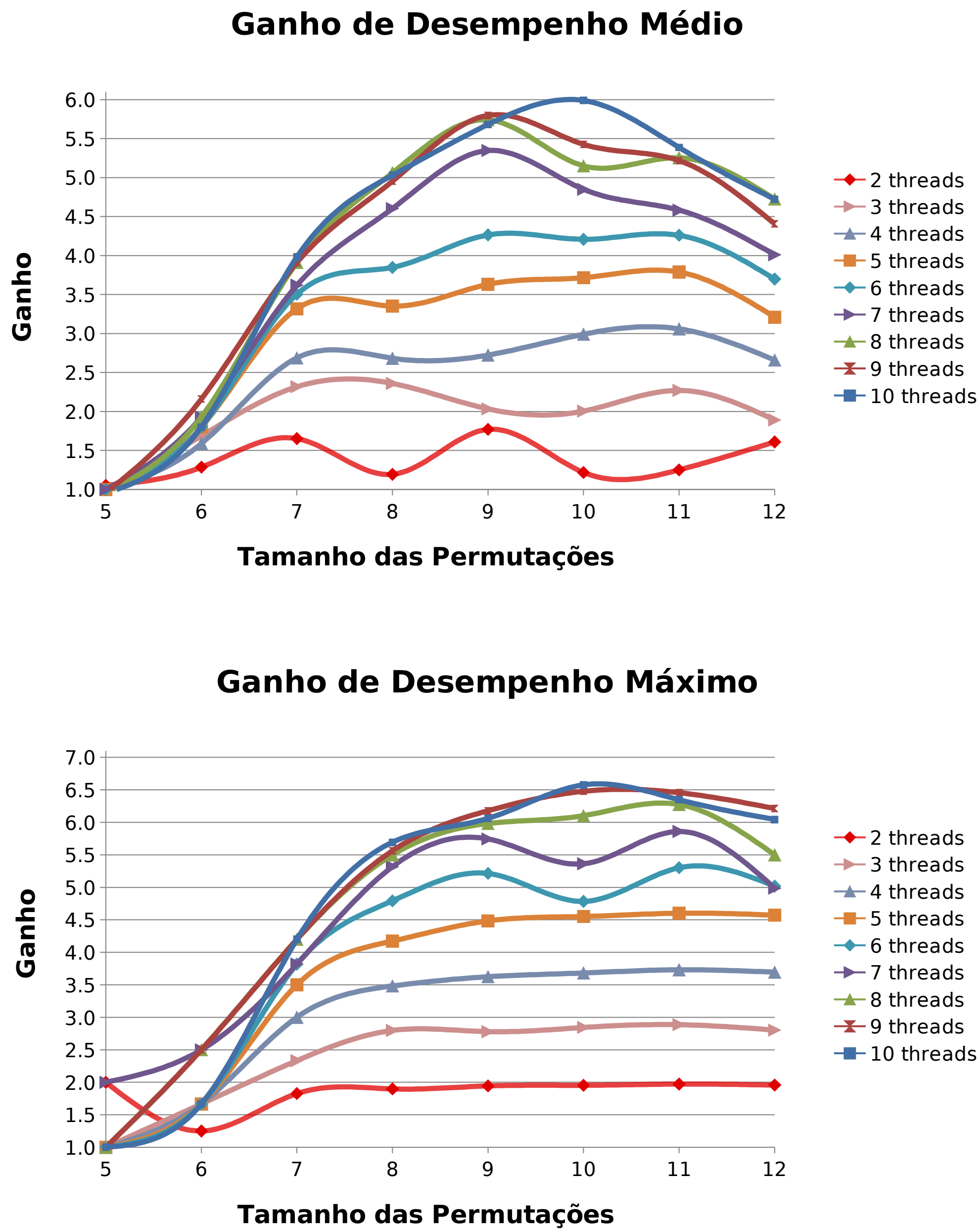


Figura 2.1: Ganho de desempenho médio e máximo observado ao executarmos a implementação de 32 bits 10 vezes para cada par (n, t), sendo n o tamanho das permutações e t o número de *threads*, sempre considerando um modelo de rearranjo composto apenas por reversões.

2.2 Cálculo das Distâncias de Rearranjo

Utilizando as implementações, nós calculamos as distâncias de rearranjo de todas as permutações em S_n e $S_n^{\pm}$ com respeito a 11 modelos de rearranjo, tal como resumido na Tabela 2.1. Esses modelos de rearranjo dão origem à variações do Problema da Ordenação por Rearranjo revisados na Seção 1.2 que são NP-Difíceis ou que não possuem complexidade conhecida e, portanto, são variações para as quais conhecemos apenas algoritmos aproximados, que são o alvo da nossa ferramenta.

Tabela 2.1: Modelos de rearranjo e valores de n considerados para o cálculo das distâncias de rearranjo.

Modelo de Rearranjo	**n** $\leq$
Reversões	13
Reversões de Prefixo	13
Reversões Curtas	12
Transposições	13
Transposições de Prefixo	13
Reversões e Transposições	13
Reversões de Prefixo e Transposições de Prefixo	13
Reversões de Prefixo com Sinal	10
Reversões com Sinal e Transposições	10
Reversões com Sinal, Transposições e Transreversões do Tipo A	10
Transposições e Transreversões do Tipo A e do Tipo B	10

Note que calculamos as distâncias de rearranjo de todas as permutações sem sinal de tamanho[1] $1 \leq n \leq 13$ e de todas as permutações com sinal de tamanho $1 \leq n \leq 10$. Não foi possível realizar cálculos para n maiores devido à restrições de memória. Em todo caso, no melhor do nosso conhecimento, é a primeira vez que cálculos para esses valores de n foram realizados.

2.2.1 Distribuição das Distâncias de Rearranjo

Embora nossa primeira intenção fosse utilizar as distâncias de rearranjo calculadas para construir a ferramenta de auditoria, nós analisamos as distribuições dessas distâncias e observamos alguns padrões que acreditamos ser interessantes. Antes de apresentarmos as distribuições e discutirmos os padrões, precisamos definir alguns conceitos.

[1] No caso do modelo de rearranjo composto apenas por reversões curtas, as distâncias para $n = 13$ ainda não foram calculadas. Em todo caso, o maior valor de n para o qual elas foram calculadas na literatura é 11.

A maior distância de rearranjo de uma permutação em S_n com respeito a um modelo de rearranjo M é chamada de *diâmetro* do grupo simétrico e é denotada por $D_M(n)$. Uma fatia de S_n é o conjunto $S_n^i = \{\pi : d_M(\pi) = i \text{ e } \pi \in S_n\}$, $0 \leq i \leq D_M(n)$. Nós escrevemos que $S_n^i \leq S_n^j$ se $|S_n^i| < |S_n^j|$ ou se $|S_n^i| = |S_n^j|$ e $i \leq j$. Seja S_n^t a fatia de S_n tal $S_n^i \leq S_n^t$ para todo $0 \leq i \leq D(n)$. Nós definimos t como o *diâmetro transversal* de S_n com respeito ao modelo de rearranjo M e o denotamos por $T_M(n)$. Estabelecemos uma relação entre $D_M(n)$ e $T_M(n)$ por meio da definição de uma função $L_M : \mathbb{N} \to \mathbb{N}$ tal que $L_M(n) = D_M(n) - T_M(n)$. Chamamos $L_M(n)$ de *longevidade* de S_n com respeito ao modelo de rearranjo M. Note que as definições de $D_M(n)$, $T_M(n)$ e $L_M(n)$ podem ser naturalmente estendidas a $S_n^\pm$.

O problema de determinar o diâmetro de S_n e de $S_n^\pm$ também é estudado em Rearranjo de Genomas. A Tabela 2.2 apresenta resultados conhecidos para os diâmetros de S_n e $S_n^\pm$ com respeito aos modelos de rearranjo da Tabela 2.1. Os modelos de rearranjo para os quais não encontramos resultados foram omitidos. Como podemos notar, o valor exato de $D_M(M)$ não é conhecido para a maior parte dos modelos de rearranjo M que estamos considerando. Tendo isso em vista, alguns autores realizaram cálculos similares aos que realizamos e conjecturaram alguns desses valores. Sendo assim, analisar as distribuições das distâncias de rearranjo é uma forma de validar essas conjecturas.

Tabela 2.2: Valores exatos e limitantes conhecidos para os diâmetros de S_n e $S_n^\pm$.

Modelo de Rearranjo (M)	$D_M(n) =$	$D_M(n) \geq$	$D_M(n) \leq$
Reversões	$n-1$ [2]	–	–
Reversões de Prefixo	–	$\frac{15n}{14}$ [41]	$\frac{11n}{8} + O(1)$ [10]
Reversões Curtas	–	$\lceil \binom{n}{2}/3 \rceil$ [40]	$\frac{3}{16}n^2 + O(n \log n)$ [22]
Transposições	–	$\frac{17n}{33} + \frac{1}{33}$ [48]	$\lfloor \frac{2n-2}{3} \rfloor$ [20]
Transposições de Prefixo	–	$\lfloor \frac{3n+1}{4} \rfloor$ [45]	$n - \log_{\frac{9}{2}} n$ [11]
Reversões de Prefixo com Sinal	–	$\frac{3n}{2}$ [13]	$2n-2$ [13]
Reversões com Sinal e Transposições	–	$\lfloor \frac{n}{2} \rfloor + 2$ [50]	–

Nós resolvemos propor as medidas $T_M(n)$ e $L_M(n)$ numa tentativa de melhor capturar a maneira como as distâncias se distribuem em S_n e em $S_n^\pm$. Quando olhamos para as distribuições das distâncias de rearranjo, notamos que o tamanho das fatias de S_n e $S_n^\pm$ cresce monotonicamente até alcançar um pico e depois decresce monotonicamente. Assim, $T_M(n)$ representa o número de fatias existentes até se chegar nesse pico e $L_M(n)$ representa o número de fatias existentes após ter se chegado nesse pico.

As distribuições das distâncias de rearranjo são dadas nas tabelas 2.3 a 2.13. Para facilitar a observação dos valores de $D_M(n)$, $T_M(n)$ e $L_M(n)$ relativos a cada tabela, nós os compilamos nas tabelas 2.14, 2.15 e 2.16.

Tabela 2.3: Distribuição da distância de reversão de prefixo com sinal em $S_n^{\pm}$.

	n									
d	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2		2	6	12	20	30	42	56	72	90
3		2	12	36	80	150	252	392	576	810
4		1	18	90	280	675	1386	2548	4320	6885
5			6	124	680	2340	6230	14056	28224	51960
6			2	96	1214	6604	24024	68656	166740	359928
7				18	1127	12795	71568	276136	843822	2193534
8				3	389	15519	159326	901970	3636954	11738418
9					40	6957	222995	2195663	12675375	53257425
10					4	959	136301	3531887	33773653	198586153
11						43	21951	2743477	60758618	570362563
12						1	1021	562095	57953163	1138201788
13							15	24627	15244962	1282857296
14							1	347	701298	435390455
15								1	6721	22703532
16									51	179828
17									1	523
18										1

Tabela 2.4: Distribuição da distância de reversão com sinal e transposição em $S_n^{\pm}$.

	n									
d	1	2	3	4	5	6	7	8	9	10
1	1	4	10	20	35	56	84	120	165	220
2		3	33	157	518	1379	3178	6594	12624	22671
3			4	201	2334	14079	59420	199539	570642	1447413
4				5	952	28668	351053	2503693	12745196	51513506
5						1897	231248	6941233	93986201	773679172
6							136	670740	78428542	2611596863
7									51189	277631156
8										198

Tabela 2.5: Distribuição da distância de reversão com sinal, transposição e transreversão do tipo A em $S_n^{\pm}$.

	n									
d	1	2	3	4	5	6	7	8	9	10
1	1	5	14	30	55	91	140	204	285	385
2		2	31	208	875	2772	7266	16632	34386	65670
3			2	144	2636	23749	135173	569861	1948013	5704127
4				1	273	19458	470796	5610677	41281942	220275867
5						9	31744	4124493	139340551	2237360500
6								52	3189382	1252484109
7										541

Tabela 2.6: Distribuição da distância de transposição e transreversão do tipo A e do tipo B em $S_n^{\pm}$.

	n									
d	1	2	3	4	5	6	7	8	9	10
1		3	12	30	60	105	168	252	360	495
2		3	33	217	969	3282	9127	21959	47330	93609
3		1	2	136	2646	26051	160851	727762	2643721	8157766
4					164	16641	468245	6295028	50811764	292482730
5							6728	3276918	131988777	2436838771
6									302607	978317822
7										6

Tabela 2.7: Distribuição da distância de reversão em S_n.

	n											
d	2	3	4	5	6	7	8	9	10	11	12	13
1	1	3	6	10	15	21	28	36	45	55	66	78
2		2	15	52	129	266	487	820	1297	1954	2831	3972
3			2	55	389	1563	4642	11407	24600	48204	87758	150707
4				2	184	2539	16445	69863	228613	626677	1510973	3304457
5					2	648	16604	169034	1016341	4398136	15240603	44997104
6						2	2111	105365	1686534	14313789	81531167	354920337
7							2	6352	654030	16584988	198802757	1489090761
8								2	17337	3900116	159650162	2717751441
9									2	42878	22073230	1499706234
10										2	102050	116855950
11											2	239756
12												2

Tabela 2.8: Distribuição da distância de transposição em S_n.

	n											
d	2	3	4	5	6	7	8	9	10	11	12	13
1	1	4	10	20	35	56	84	120	165	220	286	364
2		1	12	68	259	770	1932	4284	8646	16203	28600	48048
3			1	31	380	2700	13467	52512	170907	484440	1231230	2864719
4					45	1513	22000	191636	1183457	5706464	22822293	78829491
5							2836	114327	2010571	21171518	157499810	910047453
6									255053	12537954	265819779	3341572727
7											31599601	1893657570
8												427

Tabela 2.9: Distribuição da distância de reversão e transposição em S_n.

	n											
d	2	3	4	5	6	7	8	9	10	11	12	13
1	1	5	13	26	45	71	105	148	201	265	341	430
2			10	89	408	1301	3331	7367	14672	27002	46716	76897
3				4	266	3467	24057	111767	396691	1167102	2993970	6919519
4						200	12826	233587	2321700	15036792	71584145	272688548
5								10010	895535	23229430	325121379	2887887456
6										456208	79255048	3046408308
7												13039641

Tabela 2.10: Distribuição da distância de reversão de prefixo em S_n.

	n											
d	2	3	4	5	6	7	8	9	10	11	12	13
1	1	2	3	4	5	6	7	8	9	10	11	12
2		2	6	12	20	30	42	56	72	90	110	132
3		1	11	35	79	149	251	391	575	809	1099	1451
4			3	48	199	543	1191	2278	3963	6429	9883	14556
5				20	281	1357	4281	10666	22825	43891	77937	130096
6					133	1903	10561	38015	106461	252737	533397	1030505
7					2	1016	15011	93585	377863	1174766	3064788	7046318
8						35	8520	132697	919365	4126515	14141929	40309555
9							455	79379	1309756	9981073	49337252	184992275
10								5804	814678	14250471	118420043	639783475
11									73232	9123648	169332213	1525125357
12										956354	111050066	2183056566
13										6	13032704	1458653648
14											167	186874852
15												2001

Tabela 2.11: Distribuição da distância de transposição de prefixo em S_n.

	n											
d	2	3	4	5	6	7	8	9	10	11	12	13
1	1	3	6	10	15	21	28	36	45	55	66	78
2		2	14	50	130	280	532	924	1500	2310	3410	4862
3			3	55	375	1575	4970	12978	29610	61050	116325	208065
4				4	194	2598	18096	85128	308988	933108	2456256	5812092
5					5	562	15532	188386	1364710	7030210	28488724	96641974
6						3	1161	74183	1679189	19713542	148968371	827628815
7								1244	244430	11759676	242448896	2832043750
8									327	416845	56288493	2323157040
9										3	231058	141492748
10												31375

Tabela 2.12: Distribuição da distância de reversão de prefixo e transposição de prefixo em S_n.

	n											
d	2	3	4	5	6	7	8	9	10	11	12	13
1	1	4	8	13	19	26	34	43	53	64	76	89
2		1	15	70	190	407	759	1290	2050	3095	4487	6294
3				36	469	2478	8282	21691	48678	98049	182203	317982
4					41	2123	25565	149273	581935	1780614	4632302	10717571
5						5	5679	185801	2167742	13780795	60634266	210376864
6								4781	828184	22649298	264782966	1814609062
7									157	1604884	148475998	3711594670
8											289301	479397553
9												714

Tabela 2.13: Distribuição da distância de reversão curta em S_n.

	n										
d	2	3	4	5	6	7	8	9	10	11	12
1	1	3	5	7	9	11	13	15	17	19	21
2		2	10	23	40	61	86	115	148	185	226
3			7	41	110	220	378	592	870	1220	1650
4			1	36	194	559	1217	2273	3850	6088	9143
5				12	214	1024	2997	6861	13577	24387	40840
6					125	1337	5737	16735	39438	81276	152625
7					27	1160	8510	33416	96232	230614	489174
8						558	9460	54615	199275	565336	1367625
9						105	7304	71933	350789	1207352	3373564
10						4	3563	73934	521357	2252339	7393483
11							945	57129	645418	3662528	14442582
12							95	31544	654235	5161980	25151546
13							13	11293	534071	6260394	38977570
14							1	2222	341596	6490650	53608203
15								201	163324	5701092	65272074
16								1	53746	4173131	70165968
17									10063	2481204	66236594
18									754	1146408	54437900
19									39	379775	38444437
20										80477	22834854
21										9796	11065557
22										529	4180662
23										19	1137946
24											197727
25											18770
26											836
27											21
28											1

Tabela 2.14: Diâmetro, diâmetro transversal e longevidade de $S_n^{\pm}$.

Tabela 2.3				**Tabela 2.4**			
n	$D_M(n)$	$T_M(n)$	$L_M(n)$	n	$D_M(n)$	$T_M(n)$	$L_M(n)$
2	4	3	1	2	2	1	1
3	6	4	2	3	3	2	1
4	8	5	3	4	4	3	1
5	10	6	4	5	4	3	1
6	12	8	4	6	5	4	1
7	14	9	5	7	6	4	2
8	15	10	5	8	6	5	1
9	17	11	6	9	7	5	2
10	18	13	5	10	8	6	2
Tabela 2.5				**Tabela 2.6**			
n	$D_M(n)$	$T_M(n)$	$L_M(n)$	n	$D_M(n)$	$T_M(n)$	$L_M(n)$
2	2	1	1	2	3	2	1
3	3	2	1	3	3	2	1
4	4	2	2	4	3	2	1
5	4	3	1	5	4	3	1
6	5	3	2	6	4	3	1
7	5	4	1	7	5	4	1
8	6	4	2	8	5	4	1
9	6	5	1	9	6	5	1
10	7	5	2	10	7	5	2

Tabela 2.15: Diâmetro, diâmetro transversal e longevidade de S_n.

Tabela 2.7				**Tabela 2.8**			
n	$D_M(n)$	$T_M(n)$	$L_M(n)$	n	$D_M(n)$	$T_M(n)$	$L_M(n)$
2	1	1	0	2	1	1	0
3	2	1	1	3	2	1	1
4	3	2	1	4	3	2	1
5	4	3	1	5	3	2	1
6	5	3	2	6	4	3	1
7	6	4	2	7	4	3	1
8	7	5	2	8	5	4	1
9	8	5	3	9	5	4	1
10	9	6	3	10	6	5	1
11	10	7	3	11	6	5	1
12	11	7	4	12	7	6	1
13	12	8	4	13	8	6	2

Tabela 2.9				**Tabela 2.10**			
n	$D_M(n)$	$T_M(n)$	$L_M(n)$	n	$D_M(n)$	$T_M(n)$	$L_M(n)$
2	1	1	0	2	1	1	0
3	1	1	0	3	3	2	1
4	2	1	1	4	4	3	1
5	3	2	1	5	5	4	1
6	3	2	1	6	7	5	2
7	4	3	1	7	8	6	2
8	4	3	1	8	9	7	2
9	5	4	1	9	10	8	2
10	5	4	1	10	11	9	2
11	6	5	1	11	13	10	3
12	6	5	1	12	14	11	3
13	7	6	1	13	15	12	3

Tabela 2.16: Diâmetro, diâmetro transversal e longevidade de S_n.

Tabela 2.11

n	$D_M(n)$	$T_M(n)$	$L_M(n)$
2	1	1	0
3	2	1	1
4	3	2	1
5	4	3	1
6	5	3	2
7	6	4	2
8	6	4	2
9	7	5	2
10	8	6	2
11	9	6	3
12	9	7	2
13	10	7	3

Tabela 2.12

n	$D_M(n)$	$T_M(n)$	$L_M(n)$
2	1	1	0
3	2	1	1
4	2	2	0
5	3	2	1
6	4	3	1
7	5	3	2
8	5	4	1
9	6	5	1
10	7	5	2
11	7	6	1
12	8	6	2
13	9	7	2

Tabela 2.13

n	$D_M(n)$	$T_M(n)$	$L_M(n)$
2	1	1	0
3	2	1	1
4	4	2	2
5	5	3	2
6	7	5	2
7	10	6	4
8	14	8	6
9	16	10	6
10	19	12	7
11	23	14	9
12	28	16	12

Kececioglu e Sankoff [43] apresentaram a distribuição da distância de reversão em S_n para $n \leq 8$, porém tal distribuição é diferente da nossa. Por exemplo, podemos observar na Tabela 2.7 que $|S_5^2| = 52$, $|S_6^2| = 129$, $|S_7^2| = 266$ e $|S_8^2| = 487$, mas eles computaram que $|S_5^2| = 51$, $|S_6^2| = 127$, $|S_7^2| = 263$ e $|S_8^2| = 483$. Nós confirmamos que o nosso resultado está correto computando as distâncias de reversão das permutações de cada uma dessas fatias utilizando o GRIMM [58].

Baseados na distribuição da distância de transposição de prefixo em S_n para $1 \leq n \leq 11$, Dias e Meidanis [16] conjecturaram que o diâmetro de transposição de prefixo de S_n é igual a $n - \lfloor \frac{n}{4} \rfloor$ para $n > 3$. Como podemos observar na Tabela 2.11, tal conjectura é válida para $n = 12$ e $n = 13$. Walter, Dias and Meidanis [62] demonstraram que $\lfloor \frac{n}{2} \rfloor + 2$ é um limitante inferior para o diâmetro de reversão com sinal e transposição (Tabela 2.2) e conjecturaram que este também era um limitante superior. Como podemos observar na Tabela 2.4, esta conjectura não é válida para $n = 7$ e $n = 9$.

Com base nas distribuições apresentadas anteriormente, nós também conjecturamos valores para $D_M(n)$, $T_M(n)$ e $L_M(n)$. Tais conjecturas estão listadas na tabela 2.17.

Tabela 2.17: Conjecturas para os valores de $D_M(n)$, $T_M(n)$ e $L_M(n)$.

Modelo de Rearranjo (M)	$D_M(n)$	$T_M(n)$	$L_M(n)$	$n \geq$
Reversões	–	$\lceil \frac{2n}{3} \rceil - 1$	$\lfloor \frac{n}{3} \rfloor$	3
Reversões de Prefixo	–	$n - 1$	–	1
Reversões Curtas	–	$\lfloor \frac{(n+2)(n+1)}{11} \rfloor$	–	1
Transposições	–	$\lfloor \frac{n}{2} \rfloor$	–	1
Transposições de Prefixo	–	$n - \lceil \frac{2n}{5} \rceil$	$\lceil \frac{2n}{5} \rceil - \lfloor \frac{n}{4} \rfloor$	4
Reversões de Prefixo com Sinal	–	$\lfloor \frac{5n+2}{4} \rfloor$	–	1
Reversões com Sinal e Transposições	$n - \lfloor \frac{n-2}{3} \rfloor$	$n - \lceil \frac{n-2}{2} \rceil$	$\lceil \frac{n-2}{2} \rceil - \lfloor \frac{n-2}{3} \rfloor$	3

2.2.2 Base de Dados de Distâncias de Rearranjo

As distâncias de rearranjo calculadas foram armazenadas em arquivos indexados pelas permutações, isto é, para cada par (n, M), nós criamos um arquivo contendo todas as distâncias de rearranjo das permutações de S_n ($S_n^{\pm}$) com respeito a M tal que o registro da posição i desse arquivo contém a distância de rearranjo da permutação $f^{-1}(i)$ ($g^{-1}(i)$). Como o valor das distâncias de rearranjo calculadas não ultrapassou 255, cada registro possui um *byte* de tamanho. Dessa forma, os cálculos deram origem a 131 arquivos, totalizando aproximadamente 60GB de informação.

Nós criamos uma interface web onde é possível acessar as informações contidas nesses arquivos. O endereço de acesso é

http://mirza.ic.unicamp.br:8080.

Por meio dessa interface, um usuário pode:

- pesquisar a distância de rearranjo de uma permutação com respeito a um modelo de rearranjo;
- pesquisar as permutações pertencentes a uma fatia de S_n ($S_n^{\pm}$) com respeito a um modelo de rearranjo. Por razões de eficiência, essa pesquisa retorna 200 permutações no máximo;
- verificar a distribuição das distâncias de rearranjo em S_n ($S_n^{\pm}$) com respeito a um modelo de rearranjo.

Além de pesquisar a distância de rearranjo de uma permutação com respeito a um modelo de rearranjo, também é possível conferir a sequência de permutações que ilustra a transformação da permutação fornecida na permutação identidade. Nós chamamos essa sequência de *solução*. Note que o Algoritmo *TodasDistanciasParalelo* não computa a solução. Isso significa que implementamos uma versão diferente do Algoritmo *TodasDistanciasParalelo* a fim de computá-la.

O que nós fizemos foi criar um vetor V de tamanho $|S_n|$ tal que $V_i = f(\pi)$, sendo π a permutação pai da permutação $\sigma = f^{-1}(i-1)$, isto é, π é uma permutação tal que $\sigma = \pi \circ \rho$, $\rho \in M$, e $d_M(\sigma) = d_M(\pi) + 1 = D_i$. Desse modo, o vetor V é atualizado no momento em que σ é inserido em D e sua distância de rearranjo é armazenada no vetor D, o que corresponde ao laço **para** das linhas 18-25 do Algoritmo *TodasDistanciasParalelo*. Quando o algoritmo termina, além do vetor D, ele também retorna o vetor V.

Tendo em mãos o vetor V, é fácil recuperar a sequência de permutações que ilustra a transformação de uma permutação π na permutação identidade. O Algoritmo *RecuperaSolucao* apresentado abaixo descreve como isso pode ser feito.

Algorithm 9: RecuperaSolucao

Entrada: Uma permutação $\pi \in S_n$.
Saída: Lista L contendo uma sequência de permutações que ilustra a transformação de uma permutação π na permutação identidade.

```
1 L ← {π};
2 enquanto π ≠ ι faça
3     i ← Indexa(π);
4     π ← Desindexa(V_{i+1});
5     L ← L ∪ {π};
6 fim
7 retorna L;
```

Assim como as distâncias de rearranjo, as permutações pai foram armazenadas em arquivos indexados por permutações, isto é, para cada par $(n,\ M)$, nós criamos um arquivo contendo todas as permutações pai das permutações de S_n ($S_n^{\pm}$) com respeito a M tal que o registro da posição i desse arquivo contém a permutação pai da permutação $f^{-1}(i)$ ($g^{-1}(i)$). Diferentemente das distâncias de rearranjo, que ocupam apenas um *byte* por registro, uma permutação ocupa quatro *bytes* no caso da implementação de 32 bits e ocupa oito *bytes* no caso da implementação de 64 bits. Isso faz com que os arquivos gerados sejam muito grandes e, por esse motivo, não computamos as permutações pai das permutações sem sinal de tamanho 13. Dessa forma, foram gerados 124 arquivos, totalizando aproximadamente 72GB de informação.

Um das razões que nos levaram a construir essa interface foi permitir um "escrutínio público" das distâncias de rearranjo armazenadas na base de dados. Por exemplo, uma maneira de detectar incorreções nas distâncias de rearranjo que calculamos é analisar a distribuição dessas distâncias, a exemplo do que fizemos com a distribuição da distância de reversão apresentada por Kececioglu e Sankoff [43]. Caso alguma anomalia seja detectada, é possível recuperar permutações de uma fatia de S_n ($S_n^{\pm}$) e então verificar essas permutações. Um modo de verificá-las é avaliar se as soluções apresentadas são de fato válidas.

Outra razão foi dar ao usuários a possibilidade de extrair informações que lhes interessem a respeito das distâncias de rearranjo. Por exemplo, Grusea e Labarre [35] utilizaram as informações a respeito da distribuição das distâncias de rearranjo em um trabalho recente. Além disso, saber quais são as permutações pertencentes à fatia S_n^i tal que $i = D_M(n)$ pode ser uma informação interessante para aqueles que estão interessados em demonstrar um valor exato para o diâmetro do grupo simétrico (com ou sem sinal) com respeito ao modelo M.

2.3 GRAAu: *Genome Rearrangement Algorithm Auditor*

A auditoria de um algoritmo de rearranjo de genomas consiste-se em calcular a distância de rearranjo de todas as permutações que possuem até um certo tamanho, compará-las com as distâncias retornadas por esse algoritmo para essas permutações e produzir estatísticas, agrupadas pelo tamanho das permutações, que meçam a qualidade das respostas. GRAAu realiza exatamente esse procedimento, produzindo as seguintes estatísticas:

- **Diâmetro**: valor da maior distância retornada pelo algoritmo de rearranjo de genomas.
- **Distância Média**: média aritmética das distâncias retornada pelo algoritmo de rearranjo de genomas.
- **Razão Média**: média aritmética das razões entre a distância retornada pelo algoritmo de rearranjo de genomas e a respectiva distância de rearranjo.

- **Razão Máxima**: maior razão dentre todas as razões entre a distância retornada pelo algoritmo de rearranjo de genomas e a respectiva distância de rearranjo.

- **Igualdade**: porcentagem das distâncias retornadas pelo algoritmo de rearranjo de genomas que são iguais às respectivas distâncias de rearranjo.

Além das estatísticas, são exibidas até 50 permutações que apresentaram a razão máxima.

GRAAu foi implementado como uma aplicação cliente-servidor, sendo composta por um servidor remoto, que armazena os distâncias de rearranjo e os resultados da auditoria, e por um cliente local, que executa a implementação do algoritmo de rearranjo, compara as distâncias retornadas pelo algoritmo com as distâncias de rearranjo (obtidas do servidor) e reporta os resultados para o servidor. Note que, em vez dos resultados da auditoria serem armazenados localmente, eles são armazenados no servidor. Por essa razão, eles são disponibilizados por meio de uma interface web implementada no servidor. Nós optamos por disponibilizar os resultados da auditoria desse modo com intuito de criar um repositório centralizado e, com isso, facilitar o acesso a esses resultados por parte dos usuários.

Tanto a aplicação cliente quanto a aplicação servidora foram implementadas em Java. A comunicação entre elas é feita por meio de *web services*, que foram implementados e implantados utilizando o Apache Axis2 1.5.2. A aplicação servidora executa no Apache Tomcat 6.0.26 instalado em uma máquina com um processador Intel® Core™ i7-2600K, o qual possui quatro núcleos executando a 3.40GHz, e com 16GB de RAM. A interface web do servidor foi implementada utilizando JSF 1.2 e também executa no Apache Tomcat 6.0.26 instalado nesta máquina. O código compilado em Java e um manual (em inglês) de como utilizar o GRAAu está disponível em

http://mirza.ic.unicamp.br:8080/bioinfo/graau.jsf.

As distâncias de rearranjo ficam armazenadas no servidor tal como descrito na Seção 2.2.2. Todos os outros dados manipulados pela aplicação servidora são armazenados utilizando um gerenciador de banco de dados, mais especificamente o PostgreSQL 8.4.

A aplicação cliente é basicamente composta por uma classe, chamada *Auditor*, que possui um único método público, chamado *audit*, responsável por realizar a auditoria. Esse método recebe como parâmetro um objeto que implementa uma interface, chamada *RearrangementAlgorithm*, a qual define como um algoritmo de rearranjo de genomas deve ser implementado. Essa interface contém três métodos a serem implementados: *getName*, *getRearrangementModelCode* e *getDistance*. O método *getName* deve retornar o nome do algoritmo de rearranjo; o método *getRearrangementModelCode* deve retornar o código do modelo de rearranjo considerado pelo algoritmo; e o método *getDistance* recebe como entrada uma permutação e deve retornar a distância calculada pelo algoritmo de rearranjo de genomas para essa permutação.

A maneira como o método *audit* da classe *Auditor* realiza a auditoria é descrita em alto nível pelo Algoritmo *Auditoria*. Para facilitar a exposição do algoritmo, foram consideradas apenas permutações sem sinal, mas é trivial estendê-lo para permutações com sinal.

Algorithm 10: Auditoria

Entrada: Um objeto *obj* que implementa a interface *RearrangementAlgorithm*.
Saída: Nenhuma.

1 Seja m um número inteiro maior do que zero. Este é o número de permutações que serão processadas a cada iteração do algoritmo;
2 Crie um vetor A de tamanho m;
3 Seja M o modelo de rearranjo relativo ao código *obj.getRearrangementModelCode()*;
4 $n \leftarrow 1$;
5 **enquanto** *servidor possuir arquivo de distâncias relativo ao par* (n, M) **faça**
6     $i \leftarrow 0$;
7   **enquanto** $i < |S_n|$ **faça**
8         $j \leftarrow 0$;
9         $k \leftarrow \min(|S_n|, i + m)$;
10   **enquanto** $i + j < k$ **faça**
11             $\pi \leftarrow$ `Desindexa(`$i + j$`);`
12             $A_{j+1} \leftarrow$ *obj.getDistance(*π*)*;
13             $j \leftarrow j + 1$;
14   **fim**
15   Solicita que a aplicação servidora preencha um vetor D tal que D_t, $1 \leq t \leq k - i$, contém o valor do registro localizado na posição $i + t - 1$ do arquivo de distâncias relativo ao par (n, M);
16   Computa estatísticas parciais a partir de A e D;
17         $i \leftarrow i + j$;
18   **fim**
19   Computa as estatísticas finais a partir das estatísticas parciais e reporta o resultado para a aplicação servidora;
20 **fim**

Inicialmente são criados dois vetores A e D de tamanho m. O vetor D é utilizado para armazenar o valor das distâncias de rearranjo de permutações de S_n com respeito ao modelo de rearranjo M, enquanto o vetor A é utilizado para armazenar as distâncias retornadas pelo algoritmo de rearranjo genomas sendo auditado. Para cada par (n, M), é verificado se existe o arquivo correspondente contendo as distâncias de rearranjo no servidor. Se existir, o laço **enquanto** das linhas 7-18 é iterado $\lceil \frac{|S_n|}{m} \rceil$ vezes, sendo que na iteração i os vetores A e D são preenchidos de tal forma que se $\pi = f^{-1}(m(i-1) + j)$, então A_j contém o valor da distância de π calculada pelo algoritmo sendo auditado e D_j contém o valor $d_M(\pi)$ para todo $1 \leq j \leq$

$k - i$. Após preenchimento dos vetores, as seguintes estatísticas parciais são calculadas:

- **Diâmetro parcial**: maior valor A_j observado até a iteração i, mantido em uma variável chamada DP;
- **Soma das distâncias em A**: valor acumulado da soma $\sum A_j$ até a iteração i, mantido em uma variável chamada SA;
- **Soma das razões**: valor acumulado da soma $\sum \frac{A_j}{D_j}$ até a iteração i, mantido em uma variável chamada SR;
- **Razão máxima parcial**: maior valor $\frac{A_j}{D_j}$ observado até a iteração i, mantido em uma variável chamada RMP. Ademais, para os valores $\frac{A_j}{D_j} = RMP$, nós adicionamos o número $m(i-1)+j$ a uma lista L se L possuir menos do que 50 elementos. Caso o valor de RMP seja atualizado, a lista L é zerada;
- **Igualdade parcial**: número de vezes que foi observado $A_j = D_j$ até a iteração i, mantido em uma variável chamada IP.

Observamos que, ao término do laço **enquanto** das linhas 7-18, o Diâmetro é igual a DP, a Distância Média é igual a $\frac{SA}{|S_n|}$, a Razão Média é igual a $\frac{SR}{|S_n|}$, a Razão Máxima é igual a RMP e a Igualdade é igual a $\frac{IP}{|S_n|}$. Além disso, a lista L contém até 50 números inteiros, cada um representando uma permutação que levou a observação da Razão Máxima.

Como as permutações são processadas na ordem em que suas distâncias de rearranjo foram armazenadas no arquivo relativo ao par (n, M), a aplicação servidora preenche o vetor D (linha 15) realizando um acesso aleatório a este arquivo seguido de uma leitura sequencial de, no máximo, m *bytes*. Então, podemos considerar que o arquivo de distâncias relativo ao par (n, M) é lido em blocos de m *bytes* pela aplicação servidora. Isso implica que quanto menor o valor de m, maior o número de acessos a disco necessários para que o arquivo de distâncias relativo ao par (n, M) seja lido completamente.

Em contrapartida, um valor de m muito grande também possui desvantagens. Uma delas diz respeito a quantidade de memória necessária para alocar o vetor D tanto por parte da aplicação servidora quanto por parte da aplicação cliente. Outra desvantagem tem a ver com o mecanismo que implementamos para salvar o estado da auditoria, de maneira a podermos recomeçá-la a partir do último estado salvo. Esse mecanismo é útil nos casos em que a auditoria é interrompida (por exemplo, por um erro de comunicação entre a aplicação cliente e a aplicação servidora), pois evita que a auditoria tenha que começar do zero, economizando tempo.

O mecanismo é muito simples: logo após computar as estatísticas parciais, a aplicação cliente as reporta para o servidor juntamente com a quantidade de permutações processadas até aquele ponto (que é igual ao valor de $i+j$ no caso do Algoritmo Auditoria). Caso a auditoria seja

interrompida e reiniciada posteriormente, as estatísticas parciais são recuperadas do servidor e as permutações são processadas a partir daquela quantidade reportada ao servidor (isto é, a variável i da linha 6 seria inicializada com esse valor ao invés do valor 0). Como o estado da auditoria é salvo $\lceil \frac{|S_n|}{m} \rceil$ vezes, um valor de m muito grande implica em poucos estados salvos e, portanto, o mecanismo torna-se potencialmente menos eficaz.

Apesar da dificuldade em escolher um valor para m, nós fixamos arbitrariamente $m = 3000000$ de modo que obtivemos resultados satisfatórios na prática. Além da escolha de m, outras decisões de implementação foram tomadas no sentido de melhorar o desempenho ou poupar recursos.

Uma das decisões foi compactar o vetor D na aplicação servidora antes de enviá-lo para a aplicação cliente. Com isso, diminui-se a quantidade de dados trafegados pela rede. A compactação foi realizada utilizando uma implementação em Java [42] do algoritmo bzip2 [54].

Outra decisão foi paralelizar a implementação do Algoritmo *Auditoria*, mais especificamente do laço **enquanto** das linhas 7-18. Basicamente, a paralelização é realizada criando-se t *threads* e dividindo-se igualitariamente o cálculo do vetor A e das estatísticas parciais entre as *threads* criadas. Uma vez que as estatística parciais de cada *thread* tenham sido calculadas, elas são compiladas em uma única estatística parcial tal qual a que teríamos obtido caso tivéssemos executado o algoritmo sequencialmente. A implementação foi feita de tal forma que as *threads* não compartilham dados entre si, portanto o ganho de desempenho obtido na execução do laço **enquanto** das linhas 7-18 tende a ser quase linear no número de *threads*. O número de *threads* t deve ser passado como parâmetro do método *audit*, ou seja, é definido pelo usuário.

Devemos citar também as decisões de implementação que foram tomadas visando questões de segurança. Uma delas diz respeito à autenticação das chamadas realizadas entre a aplicação cliente e a aplicação servidora. Quando a aplicação cliente inicia a auditoria de um novo algoritmo de rearranjo de genomas, ela requisita que a aplicação servidora atribua um *id* a esse algoritmo. A aplicação servidora então cria um *id* para o algoritmo juntamente com uma chave AES aleatória de 256 bits e um contador, armazena essas informações no banco de dados e, em seguida, as envia para a aplicação cliente de modo seguro utilizando criptografia de chave pública (RSA).

A cada nova chamada ocorrida da aplicação cliente para aplicação servidora, aquela envia para esta o *id* do algoritmo e o valor encriptado (com a chave AES) do contador juntamente com os demais parâmetros da chamada. Desse modo, a aplicação servidora pode autenticar tal chamada verificando se o *id* do algoritmo e o valor encriptado do contador correspondem às informações armazenadas no banco de dados. Após chamadas como essa ocorridas com sucesso, ambas aplicações incrementam o valor do contador. A aplicação cliente armazena o *id* do algoritmo e a chave AES aleatória de 256 bits em um arquivo localizado na máquina do usuário, de tal modo que essas informações podem ser recuperadas caso a auditoria seja interrompida.

A razão de termos implementado a autenticação tal como descrito anteriormente é que não queríamos obrigar os usuários a cadastrarem credenciais (por exemplo, nome de usuário e senha) para utilizar o GRAAu. Por outro lado, isso implica que não temos nenhum controle sobre sua utilização. Por isso, resolvemos implementar um mecanismo para tentar evitar que programas (maliciosos) automatizados utilizem o GRAAu indiscriminadamente, consumindo recursos do servidor à exaustão.

Cada vez que um usuário deseja auditar um novo algoritmo de rearranjo de genomas, ele precisa gerar um tíquete, isto é, uma *string* aleatória de 32 bytes. O ponto chave está no fato de que, para gerá-lo, é necessário responder corretamente um CAPTCHA[2]. Uma vez gerado, o tíquete deve ser passado como parâmetro para o método *audit* da classe *Auditor* e ele será validado pela aplicação servidora no momento em que for lhe for requisitada a atribuição de um *id* para o algoritmo. Uma vez validado pela aplicação servidora, o tíquete torna-se inválido. Por segurança, ele também torna-se inválido caso não seja utilizado em até 15 minutos após sua criação. Note que o usuário só precisa gerar um tíquete quando for iniciar uma nova auditoria, não sendo necessário gerar um tíquete para reiniciar uma auditoria que tenha sido interrompida.

Com relação ao tempo gasto para auditar um algoritmo de rearranjo genomas com o GRAAu, infelizmente não podemos fornecer estimativas precisas, uma vez que esse tempo depende de diversos fatores, como a complexidade do algoritmo de rearranjo genomas, o número de *threads* escolhidas, a velocidade de conexão com a internet, a velocidade do processador e etc. Por exemplo, nós implementamos o algoritmo de Watterson e colegas [65] para o Problema da Ordenação por Reversões e o auditamos utilizando um computador que possuía um processador Intel Core 2 Duo rodando a 2.20 GHz e acessava a internet com uma conexão de 4 Mbps. A auditoria terminou em menos de 4 horas. Por outro lado, foram gastas aproximadamente 72 horas para auditar uma implementação ingênua do algoritmo de Kececioglu e Sankoff [43] para o Problema da Ordenação por Reversões utilizando o mesmo computador.

[2]Acrônimo para a expressão em inglês *Completely Automated Public Turing test to tell Computers and Humans Apart*. Um CAPTCHA nada mais é do que um desafio supostamente impossível de ser resolvido por um computador, mas que pode ser facilmente resolvido por um ser humano. Sendo assim, tal desafio pode ser utilizado para diferenciar seres humanos de computadores.

Capítulo 3

Algoritmos de Rearranjo de Genomas

Neste capítulo, iremos apresentar os algoritmos de rearranjo de genomas que auditamos com o GRAAu. A apresentação está dividida de tal forma que cada seção deste capítulo é destinada aos algoritmos que resolvem uma variação específica do Problema da Ordenação por Rearranjo. Antes de prosseguirmos, devemos fazer algumas observações.

Com exceção dos conceitos introduzidos na Seção 1.1, as seções não compartilham conceitos ou definições entre si. Nós optamos por construir as seções de maneira totalmente independente umas das outras para podermos redefinir alguns conceitos que, apesar de semelhantes e de até possuírem o mesmo nome, são específicos de cada problema. Os algoritmos de rearranjo de genomas são descritos tendo em vista a implementação do método *getDistance* da interface *RearrangementAlgorithm*, isto é, eles sempre recebem uma permutação como entrada e retornam o número de operações aplicadas pelo algoritmo para ordenar tal permutação. Por fim, as demonstrações dos lemas e teoremas apresentados tanto neste quanto no próximo capítulo foram desenvolvidas por nós. Resultados demonstrados na literatura são descritos ao longo dos capítulos na forma de texto corrido. Adotamos esse estilo a fim de ajudar o leitor a distinguir quais foram os resultados oriundos da nossa pesquisa e também propiciar uma leitura mais fluida do texto.

3.1 Problema da Ordenação por Reversões

O Problema da Ordenação por Reversões consiste-se em determinar a menor a sequência de reversões que ordena uma determina permutação sem sinal π. O tamanho dessa sequência é denominado como a distância de reversão de π, denotada por $d_r(\pi)$. Nesta seção, iremos apresentar dois algoritmos aproximados para o Problema da Ordenação por Reversões: o algoritmo $\frac{n-1}{2}$-aproximado proposto por Watterson e colegas [65] e o algoritmo 2-aproximado proposto Kececioglu e Sankoff [43].

Um elemento de uma permutação sem sinal é considerado estar na posição correta caso ele

não precise ser movido no processo de ordenação. Caso contrário, ele é considerado fora de posição. Formalmente, o elemento π_i de uma permutação $\pi \in S_n$ está na posição correta se $\pi_k = k$ para $1 \leq k \leq i$ ou $i \leq k \leq n$.

Exemplo 1. *Seja* $\pi = (1\ 2\ 4\ 3\ 5)$. *Nós temos que os elementos* 1, 2 *e* 5 *estão na posição correta e os elementos* 4 *e* 3 *estão fora de posição.*

O algoritmo proposto por Watterson e colegas [65] é bem simples: a cada iteração, ele aplica uma reversão que coloca na posição correta o menor elemento fora posição (Algoritmo 11). Pela definição, se $\pi_j = i$ é o menor fora de posição, então $j > i$ e $\pi_k = k$ para $1 \leq k \leq i - 1$. Logo, a reversão $r(i, j)$ coloca o elemento π_j na posição correta. Ademais, quando todos elementos estiverem sido colocados na posição correta, a permutação estará ordenada.

Algorithm 11: Algoritmo $\frac{n-1}{2}$-aproximado proposto por Watterson e colegas [65]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de reversões aplicadas para ordenar π.

1 $d \leftarrow 0$;
2 **enquanto** $\pi \neq \iota$ **faça**
3 | Seja $\pi_j = i$ o menor elemento fora de posição;
4 | $\pi \leftarrow \pi \circ r(i, j)$;
5 | $d \leftarrow d + 1$;
6 **fim**
7 **retorna** d;

O Teorema 1 mostra que esse algoritmo é uma $\frac{n-1}{2}$-aproximação. Quanto à complexidade de tempo, nós temos que as linhas 1 e 5 executam em tempo $O(1)$, as linhas 3 e 4 executam em tempo $O(n)$ e o laço **enquanto** executa $O(n)$ vezes, portanto o Algoritmo 11 roda em tempo $O(n^2)$.

Lema 1. *Seja* π *uma permutação sem sinal e* $A_{11}(\pi)$ *o número de reversões aplicadas pelo Algoritmo 11 para ordenar* π. *Se* $d_r(\pi) = 1$, *então* $A_{11}(\pi) = 1$.

Demonstração. Se $d_r(\pi) = 1$, nós temos que $\pi = r(i, j) = (1\ 2\ \ldots\ i-1\ j\ j-1\ \ldots\ i+1\ i\ j+1\ \ldots\ n)$. Isso significa que o elemento i na posição j é o menor elemento fora de posição em π. Assim, se π fosse fornecida como entrada para o Algoritmo 11, ele iria a aplicar reversão $r(i, j)$ em π, obtendo $\pi \circ r(i, j) = \iota$ e, portanto, $A_{11}(\pi) = 1$. □

Teorema 1. *O Algoritmo 11 é uma* $\frac{n-1}{2}$*-aproximação.*

Demonstração. Seja π uma permutação sem sinal de tamanho $n > 2$. No pior caso, nós temos que o Algoritmo 11 ordena π aplicando $n - 1$ reversões: primeiro ele coloca o elemento 1 na

posição correta, depois o elemento 2 na posição correta, e assim sucessivamente até que ele coloque tanto o elemento $n-1$ quando o elemento n na posição correta aplicando uma única reversão. Pelo Lema 1, $d_r(\pi) > 1$ para que $A_{11}(\pi) = n-1$. Logo, o maior valor possível para a razão $\frac{A_{11}(\pi)}{d_r(\pi)}$ é $\frac{n-1}{2}$. □

Kececioglu e Sankoff [43] atacaram o problema de uma forma diferente, baseando-se na ideia de *breakpoints*. Apesar deles terem sido os primeiros a utilizar o conceito de *breakpoint* para produzir um algoritmo com fator de aproximação constante, observamos que Watterson e colegas [65] já haviam introduzido tal conceito.

Dada uma permutação π em S_n, nós a estendemos com dois elementos $\pi_0 = 0$ e $\pi_{n+1} = n+1$. A permutação estendida continua sendo denotada por π. Um *breakpoint* de π é um par de elementos adjacentes (π_i, π_{i+1}) de π tal que $|\pi_i - \pi_{i+1}| \neq 1$, $0 \leq i \leq n$. O número de *breakpoints* de π é denotado por $b_r(\pi)$.

Exemplo 2. *Seja* $\pi = (0\ 2\ 1\ 4\ 3\ 5\ 6)$ *uma permutação estendida. Nós temos que os pares de elementos adjacentes* $(0, 2)$, $(1, 4)$ *e* $(3, 5)$ *são breakpoints, portanto* $b_r(\pi) = 3$.

Note que $b_r(\pi) = 0$ se, e somente se, $\pi = \iota$. Pelo fato de uma reversão remover no máximo dois *breakpoints* de uma permutação, o seguinte lema pode ser derivado trivialmente.

Lema 2. *Para toda permutação* π *sem sinal,* $d_r(\pi) \geq \frac{b_r(\pi)}{2}$.

Uma *strip* de π é uma sequência de elementos $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$, $0 \leq i \leq j \leq n+1$, tal que (π_{i-1}, π_i) e (π_j, π_{j+1}) são *breakpoints* e nenhum par (π_k, π_{k+1}), $i \leq k \leq j-1$, é um *breakpoint*. Uma *strip* é dita ser decrescente se $\pi_i > \pi_{i+1} > \cdots > \pi_j$, caso contrário ela é dita ser crescente. Uma *strip* formada por um único elemento é considerada decrescente, exceto quando é formada apenas por π_0 ou π_{n+1}, casos em que ela é considerada crescente.

Exemplo 3. *Seja* π *a permutação do Exemplo 2. Nós temos que as strips* 0 *e* $5\ 6$ *são crescentes e as strips* $2\ 1$ *e* $4\ 3$ *são decrescentes.*

Kececioglu e Sankoff [43] desenvolveram um algoritmo guloso (Algoritmo 13) que tenta eliminar o maior número de *breakpoints* a cada iteração. Tanto a corretude quanto o fator de aproximação desse algoritmo derivam basicamente dos dois fatos listados a seguir, provados por eles [43]:

1. Para toda permutação sem sinal que possui uma *strip* decrescente, existe uma reversão que remove um *breakpoint*;

2. Se uma permutação sem sinal π possui uma *strip* decrescente e todas as reversões que removem um *breakpoint* de π produzem uma permutação sem *strips* decrescentes, então existe uma reversão que remove dois *breakpoints* de π.

O fato 1 garante que sempre podemos remover pelo menos um *breakpoint* enquanto a permutação possuir *strips* decrescentes. Caso a permutação não possua *strips* decrescentes, podemos aplicar, no pior caso, uma reversão "neutra", que não remove e nem cria *breakpoints*, mas produz uma permutação com pelo menos uma *strip* decrescente. Se essa reversão neutra não tiver sido a primeira reversão aplicada pelo algoritmo, o fato 2 garante que a reversão aplicada anteriormente a ela removeu dois *breakpoints* e, portanto, as duas combinadas removeram um *breakpoint* na média. Do contrário, se a reversão neutra tiver sido a primeira reversão aplicada, podemos combiná-la com a última reversão aplicada pelo algoritmo, que sempre remove dois *breakpoints*, obtendo também a média de um *breakpoint* removido por reversão.

Algorithm 12: ReversaoGulosa

Entrada: Uma permutação sem sinal π.
Saída: Uma reversão gulosa $r(i, j)$.

```
1  se existir r(i, j) que remove dois breakpoints então
2  |  retorna r(i, j);
3  senão se existir r(i, j) que remove um breakpoint e deixa uma strip decrescente então
4  |  retorna r(i, j);
5  senão se existir r(i, j) que remove um breakpoint então
6  |  retorna r(i, j);
7  senão
8  |  Seja π_j = i o menor elemento fora de posição;
9  |  retorna r(i, j);
10 fim
```

Algorithm 13: Algoritmo 2-aproximado proposto por Kececioglu e Sankoff [43]

Entrada: Uma permutação sem sinal π.
Saída: Número de reversões aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto b_r(π) > 0 faça
3  |  r(i, j) ← ReversaoGulosa(π);
4  |  π ← π ∘ r(i, j);
5  |  d ← d + 1;
6  fim
7  retorna d;
```

Em resumo, podemos concluir que o Algoritmo 13 aplica no máximo $b_r(\pi)$ para ordenar uma permutação sem sinal π. Como $d_r(\pi) \geq \frac{b_r(\pi)}{2}$ (Lema 2), temos que o Algoritmo 13 é uma 2-aproximação. Com relação à complexidade de tempo, consideremos primeiramente o Algoritmo 12. Kececioglu e Sankoff [43] apontaram duas maneiras de implementá-lo: a ingênua e

a esperta. A maneira ingênua consiste-se em verificar todas as $\binom{n}{2}$ reversões para decidir qual é a reversão gulosa. Logo, essa abordagem encontra uma reversão gulosa em tempo $O(n^2)$. A maneira esperta consegue encontrar a reversão gulosa em tempo $O(n)$ considerando a forma das reversões que removem *breakpoints*.

Quanto ao Algoritmo 13, temos que o laço **enquanto** é executado $O(n)$ vezes, portanto ele executa em tempo $O(n^2)$ caso encontremos uma reversão gulosa da maneira esperta ou em tempo $O(n^3)$ caso encontremos uma reversão gulosa da maneira ingênua.

3.2 Problema da Ordenação por Reversões de Prefixo

O Problema da Ordenação por Reversões de Prefixo consiste-se em determinar a menor sequência de reversões de prefixo que ordena uma determinada permutação sem sinal π. O tamanho dessa sequência é denominado como a distância de reversão de prefixo de π, denotada por $d_{rp}(\pi)$. Nesta seção, iremos apresentar dois algoritmos aproximados propostos por Fischer e Ginzinger [24] para o Problema da Ordenação por Reversões de Prefixo.

Dada uma permutação π em S_n, nós a estendemos com dois elementos $\pi_0 = 0$ e $\pi_{n+1} = n + 1$. A permutação estendida continua sendo denotada por π. Um *breakpoint* de π em S_n é um par de elementos adjacentes (π_i, π_{i+1}) de π tal que $|\pi_i - \pi_{i+1}| \neq 1$, $1 \leq i \leq n$. O par (π_0, π_1) nunca é considerado como um *breakpoint*. O número de *breakpoints* de π é denotado por $b_{rp}(\pi)$.

Exemplo 4. *Seja $\pi = (0\ 1\ 3\ 2\ 4\ 5\ 6)$ uma permutação estendida. Nós temos que os pares de elementos adjacentes $(1, 3)$ e $(2, 4)$ são breakpoints, portanto $b_{rp}(\pi) = 2$.*

Note que $b_{rp}(\pi) = 0$ se, e somente se, $\pi = \iota$. Pelo fato de uma reversão de prefixo remover no máximo um *breakpoint* de uma permutação sem sinal, o seguinte lema pode ser derivado trivialmente.

Lema 3. *Para toda permutação sem sinal π, $d_{rp}(\pi) \geq b_{rp}(\pi)$.*

Uma *strip* de π é uma sequência de elementos $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$, $1 \leq i \leq j \leq n$, tal que (π_{i-1}, π_i) e (π_j, π_{j+1}) são *breakpoints* e nenhum par (π_k, π_{k+1}), $i \leq k \leq j-1$, é um *breakpoint*. Uma *strip* com mais de um elemento é dita ser decrescente se $\pi_i > \pi_{i+1} > \cdots > \pi_j$, caso contrário ela é dita ser crescente. Uma *strip* formada por um único elemento é chamada de *singleton*.

Exemplo 5. *Seja π a permutação do Exemplo 4. Nós temos que a strip $3\ 2$ é decrescente, a strip $4\ 5$ é crescente e a strip 1 é um singleton. Note que os elementos adicionados para estender π não são considerados.*

Um elemento de uma permutação sem sinal é considerado estar na posição correta caso ele não precise ser movido no processo de ordenação. Caso contrário, ele é considerado fora

de posição. Ademais, uma *strip* é considerada estar na posição correta se todos elementos pertencentes a ela estão na posição correta. Formalmente, o elemento π_i de uma permutação $\pi \in S_n$ está na posição correta se $\pi_k = k$ para $i \leq k \leq n$.

Exemplo 6. *Seja* $\pi = (1\ 3\ 2\ 4\ 5)$. *Nós temos que os elementos* 4 *e* 5 *estão na posição correta e os elementos* 1, 2 *e* 3 *estão fora de posição. Ademais, a strip* $4\ 5$ *está na posição correta e as strips* 1 *e* $3\ 2$ *estão fora de posição.*

Fischer e Ginzinger [24] descreveram informalmente um algoritmo 3-aproximado para ordenar uma permutação sem sinal π. A cada iteração, o algoritmo coloca a *strip* com o maior elemento fora de posição no início da permutação, corrige a direção dos elementos da *strip* (se for o caso) e então coloca a *strip* na posição correta (no final da permutação). O Algoritmo 14 descreve mais detalhadamente esse procedimento, enquanto o Teorema 2 mostra que ele é uma 3-aproximação.

Algorithm 14: Algoritmo 3-aproximado proposto por Fischer e Ginzinger [24]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de reversões de prefixo aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto π ≠ ι faça
3      Seja π_i = k maior elemento fora de posição;
4      se i > 1 então
5          j ← i;
6          enquanto j < n e π_{j+1} = π_j − 1 faça
7              j ← j + 1;
8          fim
9          π ← π ∘ rp(j);
10         d ← d + 1;
11         se j > i então
12             π ← π ∘ rp(j − i + 1);
13             d ← d + 1;
14         fim
15     fim
16     π ← π ∘ rp(k);
17     d ← d + 1;
18 fim
19 retorna d;
```

Para entender como o Algoritmo 14 funciona, primeiramente vamos assumir que o maior elemento π_j fora de posição de uma permutação π pertence à *strip* crescente $\pi_i\ \pi_{i+1} \ldots \pi_j$, $i <$

j. Então, devemos aplicar a reversão de prefixo $rp(j)$, obtendo a permutação $\sigma = \pi \circ rp(j)$. Na permutação σ, temos que $\sigma_1\ \sigma_2\ \ldots\ \sigma_{j-i+1}$ é uma *strip* decrescente tal que $\sigma_k = \pi_{i+1-k}$ para $1 \leq k \leq j - i + 1$. Logo, aplicando a reversão de prefixo $rp(\pi_j)$ em σ, obtemos a permutação $\gamma = \sigma \circ rp(\pi_j)$ tal que $\gamma_{\pi_k} = \pi_k$ para $i \leq k \leq j$.

Se tivéssemos assumido que o maior elemento π_i, $i > 1$, fora de posição da permutação π pertencesse à *strip* decrescente $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$, então a *strip* $\sigma_1\ \sigma_2\ \ldots\ \sigma_{j-i+1}$ seria crescente e, portanto, precisaríamos aplicar a reversão de prefixo $rp(j - i + 1)$ para torná-la decrescente. Caso $i = 1$, então a *strip* $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$ já estaria no início da permutação, por isso bastaria aplicar a reversão de prefixo $rp(\pi_j)$.

Por fim, se tivéssemos assumido que o maior elemento π_j fora de posição da permutação π fosse um *singleton*, teríamos que aplicar as reversões de prefixo $rp(j)$ e $rp(\pi_j)$ caso $j > 1$. Caso contrário, teríamos que aplicar apenas a reversão de prefixo $rp(\pi_j)$.

Teorema 2. *O Algoritmo 14 é uma* 3*-aproximação.*

Demonstração. As reversões de prefixo aplicadas pelo Algoritmo 14 nunca quebram *strips*, isto é, se ele aplica uma reversão de prefixo $rp(i)$ na permutação π, então (π_i, π_{i+1}) é necessariamente um *breakpoint*, ou seja, as reversões de prefixo aplicadas por ele nunca aumentam o número de *breakpoints*. Ademais, sempre que ele aplica a reversão de prefixo que coloca na posição correta a *strip* contendo o maior elemento fora de posição, ele diminui o número de *breakpoints* em pelo menos uma unidade.

No pior caso, apenas uma em cada três reversões de prefixo aplicadas pelo Algoritmo 14 diminui o número de *breakpoints* em uma unidade. Logo, se $A_{14}(\pi)$ representa o número de reversões de prefixo aplicadas pelo Algoritmo 14 para ordenar π, então $A_{14}(\pi) \leq 3b_{rp}(\pi)$. Assim, de acordo com o Lema 3, nós temos que $A_{14}(\pi) \leq 3d_{rp}(\pi)$. □

Quanto à complexidade de tempo, nós temos que as linhas 1, 4, 5, 6, 7, 10, 11, 13 e 17 executam em tempo $O(1)$, as linhas 2, 3, 9, 12 e 16 executam em tempo $O(n)$ e o laço **enquanto** executa $O(n)$ vezes, portanto o Algoritmo 14 roda em tempo $O(n^2)$.

Para desenvolver um algoritmo com um fator de aproximação melhor, Fischer e Ginzinger [24] utilizaram um estrutura chamada grafo de *breakpoints*. O grafo de *breakpoints* $G_{rp}(\pi) = (V, E)$ de uma permutação $\pi \in S_n$ é um grafo cujo conjunto de vértices é composto pelos elementos de π, isto é, $V = \{\pi_0, \ldots, \pi_{n+1}\}$, e o conjunto de arestas E é composto por arestas azuis e vermelhas, definidas da seguinte forma: uma aresta $e = (\pi_i, \pi_{i+1})$ é vermelha se (π_i, π_{i+1}) é um *breakpoint* ou se $e = (\pi_0, \pi_1)$ e $|\pi_0 - \pi_1| \neq 1$; uma aresta $e = (\pi_i, \pi_j)$ é azul se $\pi_j = \pi_i \pm 1$ e $i < j - 1$. A Figura 3.1 ilustra o grafo de *breakpoints* $G_{rp}(\pi)$ da permutação $\pi = (4\ 2\ 1\ 3)$.

Como existe pelo menos uma aresta vermelha adjacente a cada lado de uma aresta azul $e = (\pi_i, \pi_j)$ pertencente ao grafo $G_{rp}(\pi)$ de uma permutação sem sinal π, nós podemos classificá-la em pelo menos um dos quatro tipos a seguir:

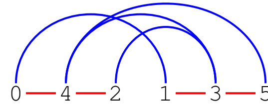


Figura 3.1: Grafo de *breakpoints* $G_{rp}(\pi)$ da permutação $\pi = (4\ 2\ 1\ 3)$.

- tipo 1, se (π_{i-1}, π_i) e (π_{j-1}, π_j) são arestas vermelhas;
- tipo 2, se (π_i, π_{i+1}) e (π_j, π_{j+1}) são arestas vermelhas;
- tipo 3, se (π_i, π_{i+1}) e (π_{j-1}, π_j) são arestas vermelhas;
- ou tipo 4, se (π_{i-1}, π_i) e (π_j, π_{j+1}) são arestas vermelhas.

Ademais, a aresta azul $e = (\pi_i, \pi_j)$ é dita ser boa se ela satisfaz uma das seguintes condições

1. e é do tipo 1, sendo $i = 1$;
2. e é do tipo 2, sendo $i \neq 0$;
3. e é do tipo 3.

Fischer e Ginzinger [24] mostraram que se o grafo $G_{rp}(\pi)$ de uma permutação $\pi \in S_n$ possui uma aresta azul boa, então é possível remover um *breakpoint* aplicando no máximo duas reversões de prefixo, tal como descrito abaixo:

- caso a aresta azul $e = (\pi_i, \pi_j)$ satisfaça a condição 1, nós temos que (π_{j-1}, π_j) é um *breakpoint* pois e é do tipo 1. Assim, a reversão de prefixo $rp(j-1)$ remove esse *breakpoint* uma vez que $\sigma_{j-1} = \pi_i$ e $\sigma_j = \pi_j$ na permutação $\sigma = \pi \circ rp(j-1)$;
- caso a aresta azul $e = (\pi_i, \pi_j)$ satisfaça a condição 2, aplicamos a reversão de prefixo $rp(j)$ em π, obtendo a permutação

  $$\sigma = \pi \circ rp(j) = (\pi_j \ldots \pi_{i+1}\ \pi_i \ldots \pi_1\ \pi_{j+1} \ldots \pi_n).$$

  Note que $j < n+1$ uma vez que e é do tipo 2, por isso a reversão de prefixo $rp(j)$ é válida. O grafo $G_{rp}(\sigma)$ contém a aresta azul $e' = (\sigma_1, \sigma_{j-i+1})$, pois $\sigma_1 = \pi_j$ e $\sigma_{j-i+1} = \pi_i$, e $(\sigma_{j-i}, \sigma_{j-i+1})$ é um *breakpoint*, pois $\sigma_{j-i} = \pi_i$ e $\sigma_{j-i+1} = \pi_{i+1}$. Assim, a reversão de prefixo $rp(j-i)$ remove esse *breakpoint* uma vez que $\gamma_{j-i} = \sigma_1$ e $\gamma_{j-i+1} = \sigma_{j-i+1}$ na permutação $\gamma = \sigma \circ rp(j-i)$;
- caso a aresta azul $e = (\pi_i, \pi_j)$ satisfaça a condição 3, aplicamos a reversão de prefixo $rp(i)$ em π, obtendo a permutação

$$\sigma = \pi \circ rp(i) = (\pi_i \ldots \pi_1 \; \pi_{i+1} \ldots \pi_{j-1} \; \pi_j \ldots \pi_n).$$

O grafo $G_{rp}(\sigma)$ contém a aresta azul $e' = (\sigma_1, \sigma_j)$, pois $\sigma_1 = \pi_i$ e $\sigma_j = \pi_j$, e (σ_{j-1}, σ_j) é um *breakpoint*, pois $\sigma_{j-1} = \pi_{j-1}$ e $\sigma_j = \pi_j$. Assim, a reversão de prefixo $rp(j-1)$ remove esse *breakpoint* uma vez que $\gamma_{j-1} = \sigma_1$ e $\gamma_j = \sigma_j$ na permutação $\gamma = \sigma \circ rp(j-1)$.

Note que se $i = 1$ e $\pi_1 \neq 1$, então a aresta azul e também satisfaz a condição 1, portanto aplicamos apenas a reversão de prefixo $rp(j-1)$ para remover um *breakpoint*. Caso $\pi_1 = 1$, então a aresta azul e não satisfaz a condição 1, porém se aplicarmos a reversão de prefixo $rp(j-1)$, removeremos o *breakpoint* (π_{j-1}, π_j) uma vez que $\sigma_{j-1} = \pi_i$ e $\sigma_j = \pi_j$ na permutação $\sigma = \pi \circ rp(j-1)$.

Se o grafo $G_{rp}(\pi)$ de uma permutação $\pi \in S_n$, $\pi \neq \iota$, não possui uma aresta azul boa, então seja $\sigma = (\pi_1 \; \pi_2 \ldots \pi_l)$, $l \leq n$, a permutação formada pelos elementos fora de posição de π. Claramente, $b_{rp}(\sigma) = b_{rp}(\pi)$ e a mesma sequência de reversões de prefixo que ordena σ irá ordenar π. Fischer e Ginzinger [24] mostraram que σ possui a seguinte forma

$$\sigma = (\underbrace{p_1 \ldots 1}_{l_1} \; \underbrace{p_2 \ldots p_1 + 1}_{l_2} \ldots \underbrace{l \ldots p_{b_{rp}(\sigma)-1} + 1}_{l_{b_{rp}(\sigma)}}).$$

Isso significa que σ é formada por $b_{rp}(\sigma) \geq 2$ *strips* decrescentes de tamanho l_i, $1 \leq i \leq b_{rp}(\sigma)$. Nesse caso, eles [24] demonstraram que a sequência de $2b_{rp}(\sigma)$ reversões de prefixo $rp(n)$, $rp(n - l_1)$, $rp(n)$, $rp(n - l_2)$, ..., $rp(n)$, $rp(n - l_{b_{rp}}(\sigma))$ transforma σ na permutação identidade.

Logo, é possível ordenar uma permutação sem sinal π aplicando não mais do que $2b_{rp}(\pi)$ reversões de prefixo utilizando a seguinte estratégia: enquanto $G_{rp}(\pi)$ possui arestas azuis boas, selecionamos uma delas e removemos um *breakpoint* aplicando no máximo duas reversões de prefixo. Caso $G_{rp}(\pi)$ não possua arestas azuis boas, a permutação σ derivada de π possui a forma descrita anteriormente, portanto podemos ordená-la aplicando exatamente $2b_{rp}(\sigma)$ reversões de prefixo. Como $d_{rp}(\pi) \geq b_{rp}(\pi)$ (Lema 3), isso significa que qualquer algoritmo que utilize tal estratégia é uma 2-aproximação.

Fischer e Ginzinger [24] não especificaram um algoritmo que implementa tal estratégia. Eles apenas disseram que arestas azuis boas que satisfazem a condição 1 devem ser privilegiadas em relação às arestas azuis boas que satisfazem as condições 2 e 3 porque as primeiras permitem que um *breakpoint* seja removido aplicando apenas uma reversão de prefixo. Por essa razão, apresentamos dois algoritmos: um que privilegia arestas azuis boas que satisfazem a condição 2 em relação às que satisfazem as condição 3 (Algoritmo 15) e outro que privilegia o inverso (Algoritmo 16).

Quanto à complexidade de tempo desses algoritmos, nós podemos dividir a análise em duas partes independentes: uma parte considera que π sempre possui uma aresta azul boa; a outra

parte considera que π não possui uma aresta azul boa. Em relação à primeira parte, nós temos que tanto verificar qual das três condições é satisfeita quanto executar as ações prevista pela condição satisfeita toma tempo $O(n)$. Como o laço **enquanto** roda $O(n)$ vezes, podemos concluir que a primeira parte roda em tempo $O(n^2)$. Em relação à segunda parte, temos que a aplicação da sequência de reversões de prefixo que transforma π na permutação identidade pode ser feita em tempo $O(n^2)$. Logo, podemos concluir que os algoritmos 15 e 16 rodam em tempo $O(n^2)$.

Algorithm 15: Algoritmo 2-aproximado baseado na estratégia proposta por Fischer e Ginzinger [24] e que privilegia arestas azuis boas que satisfazem a condição 2

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de reversões de prefixo aplicadas para ordenar π.

1 $d \leftarrow 0$;
2 **enquanto** $\pi \neq \iota$ **faça**
3 **se** $G_{rp}(\pi)$ *contém uma aresta azul* (π_i, π_j) *do tipo* 1*, sendo* $i = 1$ **então**
4 $\pi \leftarrow \pi \circ rp(j-1)$;
5 $d \leftarrow d + 1$;
6 **senão se** $G_{rp}(\pi)$ *contém uma aresta azul* (π_i, π_j) *do tipo* 2*, sendo* $i \neq 0$ **então**
7 $\pi \leftarrow \pi \circ rp(j)$;
8 $\pi \leftarrow \pi \circ rp(j-i)$;
9 $d \leftarrow d + 2$;
10 **senão se** $G_{rp}(\pi)$ *contém uma aresta azul* (π_i, π_j) *do tipo* 3 **então**
11 **se** $i > 1$ **então**
12 $\pi \leftarrow \pi \circ rp(i)$;
13 $d \leftarrow d + 1$;
14 **fim**
15 $\pi \leftarrow \pi \circ rp(j-1)$;
16 $d \leftarrow d + 1$;
17 **senão**
18 Seja l o número de elementos fora de posição de π e seja l_i o tamanho da i-ésima *strip* de π;
19 $\pi \leftarrow \pi \circ rp(l) \circ rp(l - l_1) \circ rp(l) \circ rp(l - l_2) \circ \cdots \ rp(l) \circ rp(l - l_{b_{rp}}(\pi))$;
20 $d \leftarrow d + 2b_{rp}(\pi)$;
21 **fim**
22 **fim**
23 **retorna** d;

Algorithm 16: Algoritmo 2-aproximado baseado na estratégia proposta por Fischer e Ginzinger [24] e que privilegia arestas azuis boas que satisfazem a condição 3

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de reversões de prefixo aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto π ≠ ι faça
3     se G_rp(π) contém uma aresta azul (π_i, π_j) do tipo 1, sendo i = 1 então
4        π ← π ∘ rp(j − 1);
5        d ← d + 1;
6     senão se G_rp(π) contém uma aresta azul (π_i, π_j) do tipo 3 então
7        se i > 1 então
8           π ← π ∘ rp(i);
9           d ← d + 1;
10       fim
11       π ← π ∘ rp(j − 1);
12       d ← d + 1;
13    senão se G_rp(π) contém uma aresta azul (π_i, π_j) do tipo 2, sendo i ≠ 0 então
14       π ← π ∘ rp(j);
15       π ← π ∘ rp(j − i);
16       d ← d + 2;
17    senão
18       Seja l o número de elementos fora de posição de π e seja l_i o tamanho da i-ésima
         strip de π;
19       π ← π ∘ rp(l) ∘ rp(l − l_1) ∘ rp(l) ∘ rp(l − l_2) ∘ ··· rp(l) ∘ rp(l − l_{b_rp}(π));
20       d ← d + 2b_rp(π);
21    fim
22 fim
23 retorna d;
```

3.3 Problema da Ordenação por Reversões de Prefixo com Sinal

O Problema da Ordenação por Reversões de Prefixo com Sinal consiste-se em determinar a menor a sequência de reversões de prefixo com sinal que ordena uma determinada permutação com sinal π. O tamanho dessa sequência é denominado como a distância de reversão de prefixo com sinal de π, denotada por $d_{rps}(\pi)$. Nesta seção, iremos apresentar três algoritmos aproximados para o Problema da Ordenação por Reversões de Prefixo com Sinal: um algoritmo 3-aproximado adaptado da seção anterior, um algoritmo 2-aproximado proposto por Cohen e Blum [13] e uma versão gulosa deste algoritmo proposta por nós.

Antes de prosseguirmos, devemos salientar que Cohen e Blum [13] trataram do Problema de Ordenação de Panquecas Queimadas. Apesar desse problema ser equivalente ao Problema da Ordenação por Reversões de Prefixo com Sinal, a abordagem utilizada por eles [13] não contempla os conceitos de permutação, *breakpoint* ou *strip*. Em razão disso, iremos fazer uma releitura dos resultados apresentados por eles [13] a fim de que possamos utilizar tais conceitos, mantendo assim uma linha de raciocínio a mais homogênea possível.

Dada uma permutação π em $S_n^{\pm}$, nós a estendemos com dois elementos $\pi_0 = 0$ e $\pi_{n+1} = n + 1$. A permutação estendida continua sendo denotada por π. Um *breakpoint* de uma permutação π em $S_n^{\pm}$ é um par de elementos adjacentes (π_i, π_{i+1}) tal que $\pi_{i+1} - \pi_i \neq 1$, $1 \leq i \leq n$. O par (π_0, π_1) nunca é considerado um *breakpoint*. O número de *breakpoints* de π é denotado por $b_{rps}(\pi)$.

Exemplo 7. *Seja* $\pi = (0\ {-1}\ 2\ 3\ {-5}\ {-4}\ 6)$ *uma permutação estendida. Nós temos que os pares de elementos adjacentes* $(-1, 2)$, $(3, -5)$ *e* $(-4, 6)$ *são breakpoints, portanto* $b_{rps}(\pi) = 3$.

Lema 4. *Para toda permutação com sinal* π, $d_{rps}(\pi) \geq b_{rps}(\pi)$.

Demonstração. Note que $b_{rps}(\pi) = 0$ se, e somente se, $\pi = \iota$. O lema procede pelo fato de que uma reversão de prefixo com sinal pode remover, no máximo, um *breakpoint* de uma permutação com sinal. □

Uma *strip* de π é uma sequência de elementos $\pi_i\ \pi_{i+1} \ldots \pi_j$, $1 \leq i \leq j \leq n$, tal que (π_{i-1}, π_i) e (π_j, π_{j+1}) são *breakpoints* e nenhum par (π_k, π_{k+1}), $i \leq k \leq j - 1$, é um *breakpoint*. Uma *strip* é dita ser negativa se $\pi_k < 0$ para todo $i \leq k \leq j$ e é dita ser positiva caso contrário.

Exemplo 8. *Seja* π *a permutação do Exemplo 7. Nós temos que a strip* $2\ 3$ *é positiva e as strips* -1 *e* $-5\ {-4}$ *são negativas. Note que os elementos adicionados para estender* π *não são considerados.*

Um elemento de uma permutação sem sinal é considerado estar na posição correta caso ele não precise ser movido no processo de ordenação. Caso contrário, ele é considerado fora de posição. Ademais, uma *strip* é considerada estar na posição correta se todos elementos pertencentes a ela estão na posição correta. Formalmente, o elemento π_i de uma permutação $\pi \in S_n$ está na posição correta se $\pi_k = k$ para $i \leq k \leq n$.

Exemplo 9. *Seja* $\pi = (-3\ {-2}\ {-1}\ 4\ 5)$. *Nós temos que os elementos* 4 *e* 5 *estão na posição correta e os elementos* -1, -2 *e* -3 *estão fora de posição. Ademais, a strip* $4\ 5$ *está na posição correta e a strip* $-3\ {-2}\ {-1}$ *está fora de posição.*

Cohen e Blum [13] descreveram um algoritmo trivial para ordenar uma permutação com sinal π que, a cada iteração, coloca na posição correta o elemento fora de posição com maior valor absoluto. Uma melhoria para esse algoritmo seria considerar a *strip* em que se encontra o

elemento fora de posição com maior valor absoluto. Em outras palavras, seria adaptar o algoritmo 3-aproximado descrito na seção anterior para o Problema da Ordenação por Reversões de Prefixo com Sinal. O Algoritmo 17 é o resultado dessa adaptação. O Teorema 3 mostra que ele é uma 3-aproximação.

Algorithm 17: Algoritmo 3-aproximado adaptado da seção anterior

Entrada: Uma permutação $\pi \in S_n^{\pm}$.
Saída: Número de reversões de prefixo com sinal aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto π ≠ ι faça
3      Seja π_i = k o elemento fora de posição com maior valor absoluto;
4      se i > 1 então
5          j ← i;
6          enquanto j < n e π_{j+1} = π_j + 1 faça
7              j ← j + 1;
8          fim
9          π ← π ∘ rps(j);
10         d ← d + 1;
11         se π_1 > 0 então
12             π ← π ∘ rps(j − i + 1);
13             d ← d + 1;
14         fim
15     senão se π_1 > 0 então
16         π ← π ∘ rps(1);
17         d ← d + 1;
18     fim
19     π ← π ∘ rps(|k|);
20     d ← d + 1;
21 fim
22 retorna d;
```

Para entender como o Algoritmo 17 funciona, primeiramente vamos assumir que o elemento fora de posição com maior valor absoluto π_j de uma permutação π pertence à *strip* positiva π_i $\pi_{i+1} \ldots \pi_j$, $i > 1$. Para colocar essa *strip* no início da permutação, devemos aplicar a reversão de prefixo com sinal $rps(j)$, obtendo a permutação $\sigma = \pi \circ rps(j)$. Na permutação σ, temos que $\sigma_1\ \sigma_2 \ldots \sigma_{j-i+1}$ é uma *strip* negativa pois $\sigma_k = -\pi_{i+1-k}$ para $1 \leq k \leq j - i + 1$. Logo, aplicando a reversão de prefixo com sinal $rps(\pi_j)$ em σ, obtemos a permutação $\gamma = \sigma \circ rps(\pi_j)$ tal que $\gamma_{\pi_k} = \pi_k$ para $i \leq k \leq j$. Caso $i = 1$, a *strip* seria formada por apenas um elemento e não precisaríamos colocá-la no início da permutação. Entretanto, precisaríamos aplicar a reversão de prefixo com sinal $rps(1)$ para torná-la negativa.

Se tivéssemos assumido que o elemento fora de posição com maior valor absoluto π_i, $i > 1$, da permutação π pertence à *strip* negativa $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$, então a *strip* $\sigma_1\ \sigma_2\ \ldots\ \sigma_{j-i+1}$ seria positiva e, portanto, precisaríamos aplicar a reversão de prefixo com sinal $rps(j - i + 1)$ para torná-la negativa. Caso $i = 1$, então a *strip* negativa $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$ já estaria no início da permutação, por isso bastaria aplicar a reversão de prefixo com sinal $rps(|\pi_i|)$.

Teorema 3. *O Algoritmo 17 é uma* 3*-aproximação.*

Demonstração. Análoga à demonstração do Teorema 2. □

Quanto à complexidade de tempo, nós temos que as linhas 1, 4, 5, 7, 10, 11, 13, 15, 17 e 20 executam em tempo $O(1)$, as linhas 2, 3, 9, 12, 16 e 19 executam em tempo $O(n)$ e o laço **enquanto** executa $O(n)$ vezes, portanto o Algoritmo 17 roda em tempo $O(n^2)$.

Na verdade, Cohen e Blum [13] estavam interessados em determinar um limitante superior para o diâmetro da distância de reversão de prefixo com sinal. Isso quer dizer que, ao desenvolver um algoritmo, eles não estavam interessados no fator de aproximação desse algoritmo, mas sim no número de reversões de prefixo que o algoritmo aplica no pior caso. Dessa forma, para desenvolver um algoritmo que aplica, no pior caso, menos reversões de prefixo com sinal do que o algoritmo trivial, Cohen e Blum [13] mostraram que é sempre possível eliminar um *breakpoint* de uma permutação com sinal $\pi \neq \iota$ aplicando, em média, não mais do que duas reversões de prefixo com sinal.

Eles analisaram dois casos principais:

- Caso 1: a permutação π possui pelo menos um elemento positivo fora de posição. Nesse caso, seja $\pi_i = k$ o maior elemento positivo de π fora de posição. Temos três subcasos para considerar:

  (a) Existe $\pi_j = -(k + 1)$ tal que $i < j$. Nesse subcaso, nós temos que tanto (π_{i-1}, π_i) quanto (π_j, π_{j+1}) são *breakpoints*. Assim, o *breakpoint* (π_{i-1}, π_i) pode ser removido, sem criar novos *breakpoints*, aplicando $rps(j)$ seguida de $rps(j - i)$.

  (b) Existe $\pi_j = -(k + 1)$ tal que $j < i$. Nesse subcaso, nós temos que tanto (π_i, π_{i+1}) quanto (π_{j-1}, π_j) são *breakpoints*. Assim, o *breakpoint* (π_{j-1}, π_j) pode ser removido, sem criar novos *breakpoints*, aplicando $rps(i)$ seguida de $rps(i - j)$.

  (c) Não existe $\pi_j = -(k + 1)$, ou seja, $\pi_i = k$ é elemento fora de posição com maior valor absoluto. Nesse subcaso, nós temos que tanto (π_i, π_{i+1}) quanto (π_k, π_{k+1}) são *breakpoints*. Assim, o *breakpoint* (π_k, π_{k+1}) pode ser removido, sem criar novos *breakpoints*, aplicando $rps(i)$ seguida de $rps(k)$.

- Caso 2: a permutação π não possui elementos positivos fora de posição. Temos dois subcasos para considerar:

(a) Existe $\pi_i = -(k+1)$ e $\pi_j = -k$ tal que $i < j-1$. Nesse subcaso, nós temos que tanto (π_i, π_{i+1}) quanto (π_{j-1}, π_j) são *breakpoints*. Assim, o *breakpoint* (π_{j-1}, π_j) pode ser removido, sem criar novos *breakpoints*, aplicando $rps(i)$ seguida de $rps(j - 1)$.

(b) Não existe $\pi_i = -(k+1)$ e $\pi_j = -k$ tal que $i < j-1$. Nesse subcaso, seja a permutação $\sigma = (\pi_1\ \pi_2\ \ldots\ \pi_l)$, $l \leq n$, formada apenas pelos elementos fora de posição de π. Claramente, a mesma sequência de reversões de prefixo com sinal que ordena σ irá ordenar π. O Lema 5 mostra a forma da permutação σ e o Lema 6 mostra como ordená-la aplicando não mais do $2b_{rps}(\sigma)$ reversões de prefixo com sinal.

Lema 5. *Seja $\pi \in S_n^{\pm}$ uma permutação que não possui elementos positivos fora de posição e seja $\sigma = (\pi_1\ \pi_2\ \ldots\ \pi_l)$, $l \leq n$, a permutação formada apenas pelos elementos fora de posição de π. Se o caso 2(a) não se aplica a π, então σ possui a forma*

$$\sigma = (\underbrace{p_1 \ldots -1}_{l_1}\ \underbrace{p_2 \ldots (p_1 - 1)}_{l_2} \ldots \underbrace{-l \ldots (p_{b_{rps}(\sigma)-1} - 1)}_{l_{b_{rps}(\sigma)}}),$$

tal que $p_i < 0$ para todo $1 \leq i \leq b_{rps}(\sigma) - 1$. Isto é, σ é formada por $b_{rps}(\sigma)$ strips *negativas de tamanho l_i, $1 \leq i \leq b_{rps}(\pi)$.*

Demonstração. A prova segue por indução no número de *strips* negativas de π. É trivial notar que o lema procede caso π possua uma única *strip* negativa. Para o passo da indução, vamos assumir que o lema é válido para qualquer permutação com sinal que possua até n strips negativas.

Seja π uma permutação com sinal para a qual o caso 2(a) não se aplica e que possui $n+1$ *strips* negativas. Claramente, σ é formada por essas *strips*. Se $\sigma_1 \ldots \sigma_i$, $1 \leq i < l$, é a primeira *strip* de σ, então nós temos que $\sigma_i = -1$. Note que se, ao contrário, $\sigma_i \neq -1$, então existiria um elemento $\sigma_j = \sigma_i + 1$ tal que $j > i + 1$ e, portanto, o caso 2(a) seria aplicável.

Sendo assim, seja $\sigma' \in S_{l-i}^{\pm}$ a permutação construída a partir de σ de tal forma que $\sigma'_k = \sigma_{k+i} - \pi_1$ para todo $1 \leq k \leq l - i$. Ou seja, σ' foi construída removendo-se a primeira *strip* de σ e incrementado os elementos das *strips* remanescentes de $|\sigma_1|$ unidades. É fácil notar que σ' possui n *strips*, portanto, por hipótese, ela possui a forma descrita pelo lema. Pelo modo como construímos σ', torna-se fácil concluir que a permutação σ também possui a forma descrita pelo lema e a demonstração está completa. □

Lema 6. *Se a permutação σ possui a forma descrita no Lema 5, então σ pode ser ordenada ou aplicando-se a reversão de prefixo com sinal $rps(l)$ ou aplicando-se a sequência de $2b_{rps}(\sigma)$ reversões de prefixo com sinal $rps(l)$, $rps(l-l_1)$, $rps(l)$, $rps(l-l_2)$, …, $rps(l)$, $rps(l-l_{b_{rps}}(\pi))$.*

Demonstração. É fácil notar que se σ é formada por única *strip* negativa, então a reversão de prefixo com sinal $rps(l)$ ordena σ.

Consideremos o caso em que σ é formada por $b_{rps}(\sigma) > 1$ *strips* negativas e seja σ^i a permutação obtida após aplicarmos a sequência de reversões de prefixo com sinal $rps(l)$, $rps(l - l_1)$, $rps(l)$, $rps(l - l_2)$, $\ldots$, $rps(l)$, $rps(l - l_i)$. Podemos provar por indução que σ^i é igual a

$$(\underbrace{p_{i+1} \ldots -(p_i + 1)}_{l_{i+1}}\ \underbrace{p_{i+2} \ldots p_{i+1} - 1}_{l_{i+2}} \ldots \underbrace{-l \ldots (p_{b_{rps}(\sigma)-1} - 1)}_{l_{b_{rps}(\sigma)}}\ \underbrace{1 \ldots p_1 \ldots p_i}_{l_1+l_2+\cdots+l_i})$$

para $1 \leq i \leq b_{rps}(\sigma) - 1$, tal que $p_j > 0$ para todo $1 \leq j \leq i$ e $p_j < 0$ para $i + 1 \leq j \leq b_{rps}(\sigma) - 1$.

Para $i = 1$, nós temos que $\sigma^1 = \sigma \circ rps(l) \circ rps(l - l_1)$ é igual a

$$(\underbrace{p_2 \ldots -(p_1 + 1)}_{l_2}\ \underbrace{p_3 \ldots p_3 - 1}_{l_3} \ldots \underbrace{-l \ldots (p_{b_{rps}(\sigma)-1} - 1)}_{l_{b_{rps}(\sigma)}}\ \underbrace{1 \ldots p_1}_{l_1}),$$

tal que $p_1 > 0$ e $p_j < 0$ para $2 \leq j \leq b_{rps}(\sigma) - 1$. Para o passo da indução, vamos assumir que σ^i, $i > 1$, possui a forma descrita anteriormente. Então, nós temos que $\sigma^{i+1} = \sigma^i \circ rps(l) \circ rps(l - l_{i+1})$ é igual a

$$(\underbrace{p_{i+2} \ldots -(p_{i+1} + 1)}_{l_{i+2}} \ldots \underbrace{-l \ldots (p_{b_{rps}(\sigma)-1} - 1)}_{l_{b_{rps}(\sigma)}}\ \underbrace{1 \ldots p_1 \ldots p_i \ldots p_{i+1}}_{l_1+l_2+\cdots+l_i+l_{i+1}}),$$

tal que $p_j > 0$ para todo $1 \leq j \leq i + 1$ e $p_j < 0$ para $i + 2 \leq j \leq b_{rps}(\sigma) - 1$.

É fácil observar que $\sigma^{b_{rps}(\sigma)-1} \circ rps(l) \circ rps(l - l_{b_{rps}}(\pi)) = \iota$, portanto o lema procede. $\square$

Apresentamos abaixo um algoritmo (Algoritmo 18) para resolver o Problema da Ordenação por Reversões de Prefixo com Sinal derivado da análise realizada anteriormente. O Teorema 4 mostra que esse algoritmo é uma 2-aproximação. Quanto à complexidade de tempo, nós podemos dividir a análise em duas partes independentes: uma parte considera que π sempre satisfaz as condições discutidas nos casos 1(a), 1(b), 1(c) e 2(a); outra parte considera que π satisfaz a condição discutida no caso 2(b).

Teorema 4. *O Algoritmo 18 é uma* 2*-aproximação.*

Demonstração. A partir da análise caso a caso realizada, podemos concluir que o Algoritmo 18 ordena uma permutação com sinal π aplicando não mais do que $2b_{rps}(\pi)$ reversões de prefixo. Logo, se $A_{18}(\pi)$ representa o número de reversões de prefixo com sinal aplicadas pelo Algoritmo 18 para ordenar π, então $A_{18}(\pi) \leq 2b_{rps}(\pi)$. Assim, de acordo com o Lema 4, nós temos que $A_{18}(\pi) \leq 2d_{rps}(\pi)$. $\square$

Em relação à primeira parte, nós temos que a verificação e a execução das ações propostas pelos casos 1(a), 1(b) e 1(c) (linhas 3-17) podem ser feitas em tempo $O(n)$. A verificação e a execução das ações propostas pelo caso 2(a) (linhas 19-22) também podem ser feitas em tempo $O(n)$ graças ao fato de que, como mostrado pelos lemas 5 e 7, o caso 2(a) não aplicável a uma permutação σ se, e somente se, σ possui a forma descrita pelo Lema 5. Com isso, ao invés de tentarmos encontrar dois elementos $\pi_i = -(k+1)$ e $\pi_j = -k$ tal que $i < j - 1$, nós podemos tentar descobrir se σ possui a forma descrita pelo Lema 5.

Para descobrirmos se σ possui a forma descrita pelo Lema 5, realizamos o procedimento induzido pela sua demonstração: percorremos σ da esquerda para direita verificando se o menor elemento da i-ésima *strip* é igual ao menor elemento da $(i-1)$-ésima *strip* decrescido de uma unidade (no caso da primeira *strip*, verificamos se o menor elemento é igual a -1). Se a verificação for positiva, então σ possui a forma descrita pelo Lema 5 e, portanto, o caso 2(a) não é aplicável a π (Lema 7). Caso contrário, seja σ_i o menor elemento da *strip* para qual a verificação foi negativa. Então existe um elemento $\sigma_j = \sigma_i + 1$ tal que $j > i + 1$.

Note que, em ambos os casos, realizamos uma busca linear na permutação, o que toma tempo $O(n)$. Como o laço **enquanto** roda $O(n)$ vezes, podemos concluir que a primeira parte roda em tempo $O(n^2)$. Em relação à segunda parte, temos que a aplicação da sequência de reversões de prefixo com sinal tal qual descrita pelo Lema 6 (linhas 23-31) pode ser feita em tempo $O(n^2)$. Logo, podemos concluir que Algoritmo 18 roda em tempo $O(n^2)$.

Lema 7. *Seja $\pi \in S_n^{\pm}$ uma permutação que não possui elementos positivos fora de posição e seja $\sigma = (\pi_1\ \pi_2\ \dots\ \pi_l)$, $l \leq n$, a permutação formada apenas pelos elementos fora de posição de π. Se σ possui a forma descrita no Lema 5, então o caso 2(a) não é aplicável a π.*

Demonstração. Seja $\sigma_j = -k$ um elemento pertencente à uma *strip* negativa de σ. Se σ_j não é o menor elemento da *strip*, então $\sigma_i = -(k+1)$ é tal que $i = j - 1$. Do contrário, se σ_j é o menor elemento da *strip*, então $\sigma_i = -(k+1)$ é tal que $i > j$. Isso significa que não existem dois elementos σ_i e σ_j em σ tal que $\pi_i = -(k+1)$, $\pi_j = -k$ e $i < j - 1$, portanto o lema procede. □

Apesar da estratégia de eliminação de *breakpoints* desenvolvida por Cohen e Blum [13] ser bastante semelhante à estratégia proposta por Fischer e Ginzinger [24], ela não é tão gulosa quanto, isto é, a estratégia de Cohen e Blum [13] não prevê (e, consequentemente, não prioriza) a eliminação de um *breakpoint* por meio da aplicação de uma única reversão de prefixo com sinal.

É fácil notar que a reversão de prefixo com sinal $rps(i-1)$, $i > 1$, remove um *breakpoint* de uma permutação com sinal π se, e somente se, existe $\pi_i = -\pi_1 + 1$. Assim, criamos uma versão gulosa do Algoritmo 18 (Algoritmo 19), que tenta aplicar tal reversão de prefixo com sinal antes de aplicar aquelas propostas pelos casos 1 e 2. Note que a aplicação dessa reversão de prefixo com sinal não altera a análise feita para os casos 1 e 2, com exceção de um pequeno

Algorithm 18: Algoritmo 2-aproximado proposto por Cohen e Blum [13]

Entrada: Uma permutação $\pi \in S_n^{\pm}$.
Saída: Número de reversões de prefixo com sinal aplicadas para ordenar π.

1 $d \leftarrow 0$;
2 **enquanto** $\pi \neq \iota$ **faça**
3   **se** *π possui pelo menos um elemento positivo fora de posição* **então**
4   Seja $\pi_i = k$ o maior elemento positivo de π fora de posição;
5   **se** *existe* $\pi_j = -(k+1)$ *tal que* $i < j$ **então**
6       $\pi \leftarrow \pi \circ rps(j)$;
7       $\pi \leftarrow \pi \circ rps(j-i)$;
8       $d \leftarrow d + 2$;
9   **senão se** *existe* $\pi_j = -(k+1)$ *tal que* $j < i$ **então**
10       $\pi \leftarrow \pi \circ rps(i)$;
11       $\pi \leftarrow \pi \circ rps(i-j)$;
12       $d \leftarrow d + 2$;
13   **senão**
14       $\pi \leftarrow \pi \circ rps(i)$;
15       $\pi \leftarrow \pi \circ rps(k)$;
16       $d \leftarrow d + 2$;
17   **fim**
18   **senão**
19   **se** *existe* $\pi_i = -(k+1)$ *e* $\pi_j = -k$ *tal que* $i < j - 1$ **então**
20       $\pi \leftarrow \pi \circ rps(i)$;
21       $\pi \leftarrow \pi \circ rps(j-1)$;
22       $d \leftarrow d + 2$;
23   **senão**
24   Seja l o número de elementos fora de posição de π e seja l_i o tamanho da i-ésima *strip* de π;
25       $\pi \leftarrow \pi \circ rps(l)$;
26       $d \leftarrow d + 1$;
27   **se** $\pi_1 \neq 1$ **então**
28         $\pi \leftarrow \pi \circ rps(l - l_1) \circ rps(l) \circ rps(l - l_2) \circ \cdots \circ rps(l) \circ rps(l - l_{b_{rps}}(\pi))$;
29         $d \leftarrow d + 2b_{rps}(\pi) - 1$;
30   **fim**
31   **fim**
32   **fim**
33 **fim**
34 **retorna** d;

detalhe: se π satisfaz o caso 2(b) e σ é formada por uma única *strip* negativa, então π possui uma reversão de prefixo com sinal que não só remove um *breakpoint* como também ordena π, portanto ela irá satisfazer a condição gulosa antes de satisfazer a condição descrita pelo caso 2(b).

Como, no pior caso, o Algoritmo 19 se comporta como Algoritmo 18, podemos concluir que ele também é uma 2-aproximação. Ademais, dado que as linhas 3-6 podem ser executadas em tempo $O(n)$, temos que ele também roda em tempo $O(n^2)$.

Algorithm 19: Versão gulosa do Algoritmo 18

Entrada: Uma permutação $\pi \in S_n^{\pm}$.
Saída: Número de reversões de prefixo com sinal aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto π ≠ ι faça
3      se existe π_i = −π_1 + 1 com i > 1 então
4          π ← π ∘ rps(i − 1);
5          d ← d + 1;
6      senão
7          Execute as linhas 3-32 do Algoritmo 18;
8      fim
9  fim
10 retorna d;
```

3.4 Problema da Ordenação por Reversões Curtas

O Problema da Ordenação por Reversões Curtas consiste-se em determinar a menor a sequência de reversões curtas que ordena uma determinada permutação sem sinal π. O tamanho dessa sequência é denominado como a distância de reversão curta de π, denotada por $d_{rc}(\pi)$. Nesta seção, iremos apresentar três algoritmos aproximados para o Problema da Ordenação por Reversões Curtas: um algoritmo 3-aproximado e um algoritmo 2-aproximado proposto por Vergara [60] e um algoritmo 2-aproximado propostos por Heath e Vergara [40, 60].

Uma inversão em uma permutação sem sinal π é um par de elementos (π_i, π_j) que não estão na ordem relativa correta, isto é, $i < j$ e $\pi_i > \pi_j$. O número de inversões de π é denotado por $inv(\pi)$. Note que a permutação identidade é a única permutação que não possui inversões.

Exemplo 10. *Seja* $\pi = (3\ 2\ 4\ 1)$. *Os pares de elementos* $(3,\ 2)$, $(3,\ 1)$, $(2,\ 1)$ *e* $(4,\ 1)$ *são inversões em* π.

Uma reversão curta $rc(i,\ j)$ é dita ser corretiva se ela inverte a ordem de dois elementos π_i e π_j que são uma inversão em π. Heath e Vergara [40, 60] demonstraram que, para qualquer

permutação sem sinal π, existe uma sequência de reversões curtas que ordena π otimamente tal que todas as reversões curtas são corretivas. Isso garante que podemos restringir nossa análise apenas às reversões curtas corretivas. Como uma reversão curta corretiva pode remover 1 ou 3 inversões de uma permutação sem sinal (Lema 8), o Lema 9 segue trivialmente.

Lema 8. *Uma reversão curta corretiva pode remover 1 ou 3 inversões de uma permutação sem sinal.*

Demonstração. Uma reversão curta corretiva $rc(i, i+1)$ remove somente 1 inversão de uma permutação π, aquela caracterizada pelos par de elementos π_i e π_{i+1}. Já uma reversão curta corretiva $rc(i, i+2)$ remove 1 inversão caso $\pi_{i+1} > \pi_i > \pi_{i+2}$ ou $\pi_i > \pi_{i+2} > \pi_{i+1}$ e remove 3 inversões caso $\pi_i > \pi_{i+1} > \pi_{i+2}$. □

Lema 9. *Para toda permutação sem sinal π, $d_{rc}(\pi) \geq \frac{inv(\pi)}{3}$.*

Três elementos contíguos $(\pi_i, \pi_{i+1}, \pi_{i+2})$, $0 \leq i \leq n-2$, de uma permutação $\pi \in S_n$ formam uma tripla se $\pi_i > \pi_{i+1} > \pi_{i+2}$. Como visto na demonstração do Lema 8, a reversão curta corretiva $rc(i, i+2)$ remove 3 inversões de π. O conjunto formado por todas as triplas de π é denotado por $triplas(\pi)$.

Exemplo 11. *Seja $\pi = (4\ 3\ 2\ 1\ 5)$. Nós temos que $triplas(\pi) = \{(4, 3, 2), (3, 2, 1)\}$.*

Vergara [60] desenvolveu um algoritmo guloso (Algoritmo 20) baseado em inversões. A cada iteração, esse algoritmo seleciona a reversão curta corretiva que remove a maior quantidade de inversões possível. Caso exista mais do que uma reversão curta corretiva que remove a maior quantidade de inversões, o algoritmo seleciona aquela que produz uma permutação com o maior número de triplas.

Como o Algoritmo 20 remove pelo menos uma inversão a cada reversão curta aplicada e $d_{rc}(\pi) \geq \lceil \frac{inv(\pi)}{3} \rceil$ (Lema 9), nós concluímos que ele é uma 3-aproximação. Quanto à complexidade de tempo, $triplas(\pi)$ pode ser computada em tempo $O(n)$. Como existem no máximo $n-2$ triplas e aplicar uma reversão curta leva tempo constante, podemos computar tanto a linha 4 quanto a linha 8 também em tempo $O(n)$. Ou seja, as linhas 3-11 são executadas em tempo $O(n)$. Dado que podem existir no máximo $\binom{n}{2}$ inversões, o laço **enquanto** executa $O(n^2)$ vezes, portanto o Algoritmo 20 roda em tempo $O(n^3)$.

Para desenvolver um algoritmo com um fator de aproximação melhor, Heath e Vergara [40, 60] utilizaram um estrutura chamada diagrama vetorial. Para cada elemento π_i de uma permutação sem sinal π, definimos um vetor $v(\pi_i)$ cujo tamanho é dado por $|v(\pi_i)| = |\pi_i - i|$. Se $|v(\pi_i)| > 0$, a direção de $v(\pi_i)$ é dada pelo sinal de $\pi_i - i$. O diagrama vetorial $V_{rc}(\pi)$ de π é o conjunto formado pelos vetores dos elementos de π.

Exemplo 12. *Seja π a permutação do Exemplo 11. Nós temos que $v(\pi_1) = 3$, $v(\pi_2) = 1$, $v(\pi_3) = -1$, $v(\pi_4) = -3$ e $v(\pi_5) = 0$. Logo, $V_{rc}(\pi) = \{3, 1, -1, -3, 0\}$.*

Algorithm 20: Algoritmo 3-aproximado proposto por Vergara [60]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de reversões curtas aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto π ≠ ι faça
3      se |triplas(π)| > 0 então
4          Seja (π_i, π_{i+1}, π_{i+2}) uma tripla tal que |triplas(π ∘ rc(i, i + 2))| possui valor
           máximo;
5          π ← π ∘ rc(i, i + 2);
6          d ← d + 1;
7      senão
8          Seja rc(i, j) uma reversão curta corretiva tal que |triplas(π ∘ rc(i, j))| possui
           valor máximo;
9          π ← π ∘ rc(i, j);
10         d ← d + 1;
11     fim
12 fim
13 retorna d;
```

A soma dos tamanhos dos vetores pertencentes a $V_{rc}(\pi)$ é denotada por $|V_{rc}(\pi)|$. Note que $|V_{rc}(\pi)| = 0$ se, e somente se, $\pi = \iota$. Como uma reversão curta corretiva pode diminuir $|V_{rc}(\pi)|$ de 4 unidades no máximo (Lema 10), o Lema 11 segue trivialmente.

Lema 10. *Seja π uma permutação sem sinal. Uma reversão curta corretiva pode diminuir $|V_{rc}(\pi)|$ de 4 unidades no máximo.*

Demonstração. Uma reversão curta corretiva $rc(i, j)$ só pode diminuir o tamanho dos vetores $v(\pi_i)$ e $v(\pi_j)$. Ademais, o tamanho de cada um desses vetores pode ser diminuído de duas unidades no máximo, caso $\pi_i - i \geq 2$ e $\pi_j - j \leq -2$. Logo, $|V_{rc}(\pi)| - |V_{rc}(\pi \circ rc(i, j))| \leq 4$. □

Lema 11. *Para toda permutação sem sinal π, $d_{rc}(\pi) \geq \frac{|V_{rc}(\pi)|}{4}$.*

Dois elementos π_i e π_j, $i < j$, de uma permutação sem sinal π são ditos serem vetorialmente opostos se os vetores $v(\pi_i)$ e $v(\pi_j)$ possuem direções opostas e tanto $|v(\pi_i)| \geq j - i$ quanto $|v(\pi_j)| \geq j - i$. Ademais, eles são dito serem m-vetorialmente opostos se $j - i = m$. Note que m especifica a distância entre elementos vetorialmente opostos.

Exemplo 13. *Seja π a permutação do Exemplo 11. Nós temos que os elementos π_2 e π_3 são 1-vetorialmente opostos e os elementos π_1 e π_4 são 3-vetorialmente opostos.*

Heath e Vergara [40, 60] provaram que toda permutação sem sinal π, $\pi \neq \iota$, possui pelo menos dois elementos vetorialmente opostos. Sejam π_i e π_j elementos m-vetorialmente opostos de $\pi \in S_n$ e seja $\pi' \in S_n$ a permutação que possui tais elementos em posições invertidas, isto é, $\pi'_k = \pi_k$ para $k \in \{1, 2, \ldots, n\} \setminus \{i, j\}$, $\pi'_i = \pi_j$ e $\pi'_j = \pi_i$. Eles também provaram que $|V_{rc}(\pi)| - |V_{rc}(\pi')| = 2m$, ou seja, ao trocarmos dois elementos m-vetorialmente opostos de uma permutação sem sinal π, diminuímos $|V_{rc}(\pi)|$ de $2m$ unidades.

Sejam π_i e π_j elementos m-vetorialmente opostos em $\pi \in S_n$. Vergara [60] mostrou que é possível trocar π_i e π_j de posição por meio da aplicação de reversões curtas, tal como descrito pelo Algoritmo *TrocaComReversoesCurtas*. Além disso, ele também mostrou que o Algoritmo *TrocaComReversoesCurtas* aplica $m - 1$ reversões curtas se m é par e aplica m reversões curtas se m é ímpar. Isso implica que cada reversão curta aplicada pelo Algoritmo *TrocaComReversoesCurtas* diminui $|V_{rc}(\pi)|$ de $\frac{2m}{m-1}$ unidades em média se m é par. Por outro lado, se m é ímpar, então cada reversão curta aplicada diminui $|V_{rc}(\pi)|$ de 2 unidades em média.

Algorithm 21: TrocaComReversoesCurtas

Entrada: Uma permutação $\pi \in S_n$ e par de posições (i, j) tal que $i < j$.
Saída: O número de reversões curtas aplicadas para trocar os elementos π_i e π_j de posição.

```
1  k ← i;
2  d ← 0;
3  enquanto k < j − 2 faça
4  |   π ← π ∘ rc(k, k + 2);
5  |   k ← k + 2;
6  |   d ← d + 1;
7  fim
8  se j − i é ímpar então
9  |   π ← π ∘ rc(k, k + 1);
10 |   k ← k − 2;
11 |   d ← d + 1;
12 fim
13 enquanto k ≥ i faça
14 |   π ← π ∘ rc(k, k + 2);
15 |   k ← k − 2;
16 |   d ← d + 1;
17 fim
18 retorna d;
```

Heath e Vergara [40, 60] apresentam um algoritmo para uma ordenar uma permutação sem sinal por reversões curtas baseado em elementos vetorialmente opostos (Algoritmo 23). A cada iteração, o algoritmo seleciona dois elementos vetorialmente opostos e chama o Algoritmo

Algorithm 22: EncontraOpostos

Entrada: Uma permutação $\pi \in S_n$.
Saída: Par de posições (i, j) tal que π_i e π_j são vetorialmente opostos.

1 $i \leftarrow n$;
2 **enquanto** $\pi_i \leq i$ **faça**
3 | $i \leftarrow i - 1$;
4 **fim**
5 $j \leftarrow i + 1$;
6 **enquanto** $\pi_j = j$ **faça**
7 | $j \leftarrow j + 1$;
8 **fim**
9 **retorna** (i, j);

TrocaComReversoesCurtas para realizar a troca desses elementos. Como as reversões curtas aplicadas pelo Algoritmo *TrocaComReversoesCurtas* diminui a soma dos tamanhos dos vetores, em algum momento esta soma irá ser nula, precisamente quando a permutação estiver ordenada. No pior caso, cada reversão curta aplicada pelo algoritmo diminui a soma dos tamanhos dos vetores de 2 unidades em média. Dado que $d_{rc}(\pi) \geq \frac{|V_{rc}(\pi)|}{4}$ (Lema 11), concluímos que esse algoritmo é uma 2-aproximação.

Algorithm 23: Algoritmo 2-aproximado proposto por Heath e Vergara [40, 60]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de reversões curtas aplicadas para ordenar π.

1 $d \leftarrow 0$;
2 **enquanto** $\pi \neq \iota$ **faça**
3 | $(i, j) \leftarrow$ `EncontraOpostos(`π`);`
4 | $d \leftarrow d +$ `TrocaComReversoesCurtas(`π, i, j`);`
5 **fim**
6 **retorna** d;

Quanto à complexidade de tempo, nós temos que podem existir no máximo $\binom{n}{2}$ elementos vetorialmente opostos, portanto o laço **enquanto** executa $O(n^2)$ vezes. Cada chamada ao Algoritmo *EncontraOpostos* toma tempo $O(n)$, então as chamadas a esse algoritmo tomam tempo $O(n^3)$ no total. O tempo total gasto pelas chamadas ao Algoritmo *TrocaComReversoesCurtas* não depende do número de vezes que o laço *enquanto* itera, mas sim da soma dos tamanhos dos vetores. Cada chamada do Algoritmo *TrocaComReversoesCurtas* toma tempo $O(m)$ para diminuir a soma dos tamanhos dos vetores de $2m$. Como cada vetor pode ter tamanho igual a n no máximo, a soma dos tamanhos dos vetores é igual a n^2 no máximo. Logo, no total, as

chamadas ao Algoritmo *TrocaComReversoesCurtas* tomam tempo $O(n^2)$. Sendo assim, podemos concluir que o Algoritmo 23 executa em tempo $O(n^3)$.

Vergara [60] desenvolveu uma variante gulosa do Algoritmo 23 (Algoritmo 25) baseada na distância entre elementos vetorialmente opostos. A cada iteração, o algoritmo tenta encontrar dois elementos m-vetorialmente opostos tal que m seja par e possua o menor valor possível. Caso ele encontre, o algoritmo seleciona tais elementos. Caso ele não encontre, o algoritmo seleciona dois elementos m-vetorialmente opostos tal que m possua o menor valor possível. Após ter selecionado dois elementos m-vetorialmente opostos, o algoritmo chama o Algoritmo *TrocaComReversoesCurtas* para realizar a troca desses elementos. No pior caso, o algoritmo seleciona apenas elementos m-vetorialmente opostos tal que m é ímpar e aplica, portanto, $\frac{|V_{rc}(\pi)|}{2}$ reversões curtas. Desse modo, esse algoritmo também é uma 2-aproximação.

Algorithm 24: EncontraOpostosGuloso

Entrada: Uma permutação $\pi \in S_n$.
Saída: Par de posições (i, j) tal que π_i e π_j são vetorialmente opostos.

1 **se** *existem dois elementos m-vetorialmente opostos tal que m é par* **então**
2     $(i, j) \leftarrow$ posições de dois elementos m-vetorialmente opostos tal que m é par e m possui o menor valor possível;
3   **retorna** (i, j);
4 **senão**
5     $(i, j) \leftarrow$ posições de dois elementos m-vetorialmente opostos tal que m possui o menor valor possível;
6   **retorna** (i, j);
7 **fim**

Algorithm 25: Variante gulosa do Algoritmo 23 proposta por Vergara [60]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de reversões curtas aplicadas para ordenar π.

1 $d \leftarrow 0$;
2 **enquanto** $\pi \neq \iota$ **faça**
3     $(i, j) \leftarrow$ `EncontraOpostosGuloso(`π`)`;
4     $d \leftarrow d +$ `TrocaComReversoesCurtas(`π, i, j`)`;
5 **fim**
6 **retorna** d;

Quanto à complexidade de tempo, a única diferença em relação ao Algoritmo 23 é o tempo gasto pelas chamadas ao Algoritmo *EncontraOpostosGuloso*. A cada iteração, ele tem verificar

todos os elementos vetorialmente oposto, o que toma tempo $O(n^2)$. Sendo assim, podemos concluir que o Algoritmo 25 executa em tempo $O(n^4)$.

3.5 Problema da Ordenação por Transposições

O Problema da Ordenação por Transposições consiste-se em determinar a menor a sequência de transposições que ordena uma determinada permutação sem sinal π. O tamanho dessa sequência é denominado como a distância de transposição de π, denotada por $d_t(\pi)$. Nesta seção, iremos apresentar três algoritmos aproximados para o Problema da Ordenação por Transposições: um algoritmo 2.25-aproximado proposto por Walter, Dias e Meidanis [63]; um algoritmo 3-aproximado proposto por Benoît-Gagné e Hamel [5]; e uma versão restrita de uma heurística proposta por Guyer, Heath e Vergara [37], que demonstramos ser uma 3-aproximação.

Dada uma permutação π em S_n, nós a estendemos com dois elementos $\pi_0 = 0$ e $\pi_{n+1} = n+1$. A permutação estendida continua sendo denotada por π. Um *breakpoint* de uma permutação π em S_n é um par de elementos adjacentes (π_i, π_{i+1}) tal que $\pi_{i+1} - \pi_i \neq 1$, $0 \leq i \leq n$. O número de *breakpoints* de π é denotado por $b_t(\pi)$.

Exemplo 14. *Seja* $\pi = (0\ 4\ 5\ 2\ 3\ 1\ 6)$ *uma permutação estendida. Nós temos que os pares* $(0, 4)$, $(5, 2)$, $(3, 1)$ *e* $(1, 6)$ *são* breakpoints*, portanto* $b_t(\pi) = 4$.

Note que $b_t(\pi) = 0$ se, e somente se, $\pi = \iota$. Pelo fato de uma transposição remover no máximo três *breakpoints* de uma permutação, o seguinte lema pode ser derivado trivialmente.

Lema 12. *Para toda permutação sem sinal* π, $d_t(\pi) \geq \frac{b_t(\pi)}{3}$.

Walter, Dias e Meidanis [63] desenvolveram um algoritmo aproximado baseado em *breakpoints*. Não iremos discutir detalhadamente como este algoritmo funciona pois ele depende de uma análise caso a caso extensa e essa análise não é relevante para a discussão que enveredaremos nesta dissertação. Em todo caso, apresentamos abaixo um esboço desse algoritmo (Algoritmo 26) para que fique claro qual o seu fator de aproximação.

Note que, no pior caso, o Algoritmo 26 remove 4 *breakpoints* aplicando 3 transposições. Assim, se o número de transpoições aplicadas por esse algoritmo para ordenar uma permutação sem sinal π é denotado por $A_{26}(\pi)$, nós temos que $A_{26}(\pi) \leq \frac{3}{4} b_t(\pi)$. Como $d_t(\pi) \geq \frac{b_t(\pi)}{3}$ (Lema 12), nós podemos concluir que o Algoritmo 26 é uma 2.25-aproximação. Quanto à complexidade de tempo, Walter, Dias e Meidanis [63] mostraram que o algoritmo executa em tempo $O(n^2)$.

Uma *strip* de π é uma sequência de elementos $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$, $1 \leq i \leq j \leq n$, tal que (π_{i-1}, π_i) e (π_j, π_{j+1}) são *breakpoints* e nenhum par (π_k, π_{k+1}), $i \leq k \leq j-1$, é um *breakpoint*. O número de *strips* de π é denotado por $s_t(\pi)$.

Algorithm 26: Esboço do algoritmo 2.25-aproximado proposto por Walter, Dias e Meidanis [63]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de transposições aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto π ≠ ι faça
3      se existe uma transposição t(i, j, k) que remove 3 breakpoints de π então
4          π ← π ∘ t(i, j, k);
5          d ← d + 1;
6      senão se existe uma transposição t(i, j, k) que remove 2 breakpoints de π então
7          π ← π ∘ t(i, j, k);
8          d ← d + 1;
9      fim
10     Encontre até 3 transposições que removem pelo menos 4 breakpoints quando
       aplicadas em π;
11     Aplique em π as transposições encontradas, atualizando d de acordo;
12 fim
13 retorna d;
```

Exemplo 15. *Seja $\pi =$ a permutação do Exemplo 14. Nós temos que $4\ 5$, $2\ 3$ e 1 são strips de π, portanto $s_t(\pi) = 3$. Note que os elementos adicionados para estender π não são considerados.*

Seja $\pi \in S_n$, $\pi \neq \iota$, s_1 a primeira *strip* de π e s_m a última *strip* de π, $m \leq n$. Se assumirmos que $\pi_1 = 1$, nós podemos reduzir π para uma permutação $\sigma \in S_{n-|s_1|}$ tal que $\sigma_i = \pi_i - |s_1|$, $i > |s_1|$. Não é difícil notar que $d_t(\pi) = d_t(\sigma)$. Um argumento análogo pode ser utilizado para mostrar que podemos reduzir π para uma permutação $\gamma \in S_{n-|s_m|}$ tal que $d_t(\pi) = d_t(\gamma)$ caso $\pi_n = n$. Nós chamados de irredutível qualquer permutação para a qual tais reduções não se aplicam e denotamos por S_n^* o conjunto formado por todas as permutações irredutíveis de S_n.

Lema 13. *Para toda permutação $\pi \in S_n^*$, $s_t(\pi) = b_t(\pi) - 1$.*

Demonstração. Sejam $s_1, s_2, \ldots, s_{s_t(\pi)}$ as *strips* de uma permutação $\pi \in S_n^*$. O último elemento da *strip* s_i e o primeiro elemento da *strip* s_{i+1}, $1 \leq i \leq s_t(\pi) - 1$, formam um *breakpoint*. Além disso, os pares (π_0, π_1) e (π_n, π_{n+1}) também são *breakpoints* já que π é irredutível. Portanto, nós temos que $b_t(\pi) = s_t(\pi) + 1$ e o lema segue. □

Dada uma permutação $\pi \in S_n$, o código esquerdo de um elemento π_i, denotado por $ce(\pi_i)$, é definido como

$$ce(\pi_i) = |\{\pi_j : \pi_j > \pi_i \text{ e } 1 \leq j \leq i - 1\}|$$

para todo $1 \leq i \leq n$. O código esquerdo da permutação π, denotado por $ce(\pi)$, é então definido como $ce(\pi) = ce(\pi_1)\ ce(\pi_2)\ \ldots\ ce(\pi_n)$. Similarmente, o código direito de um elemento π_i, denotado por $cd(\pi_i)$, é definido como

$$cd(\pi_i) = |\{\pi_j : \pi_j < \pi_i \text{ e } i+1 \leq j \leq n\}|$$

para todo $1 \leq i \leq n$. O código direito da permutação π, denotado por $cd(\pi)$, é então definido como $cd(\pi) = cd(\pi_1)\ cd(\pi_2)\ \ldots\ cd(\pi_n)$. Note que $ce(\pi) = cd(\pi) = 0\ 0\ \ldots\ 0$ se, e somente se, $\pi = \iota$.

Exemplo 16. *Seja* $\pi = (5\ 3\ 2\ 4\ 1)$. *Nós temos que* $ce(\pi) = ce(\pi_1)\ ce(\pi_2)\ ce(\pi_3)\ ce(\pi_4)\ ce(\pi_5) = 0\ 1\ 2\ 1\ 4$ *e que* $cd(\pi) = cd(\pi_1)\ cd(\pi_2)\ cd(\pi_3)\ cd(\pi_4)\ cd(\pi_5) = 4\ 2\ 1\ 1\ 0$.

Em um código, uma sequência de elementos contíguos de tamanho maximal que possuem o mesmo valor diferente de zero é chamada de platô. Dada uma permutação sem sinal π, o número de platôs em $ce(\pi)$ é denotado por $p(ce(\pi))$ e o número de platôs em $cd(\pi)$ é denotado por $p(cd(\pi))$. Nós denotamos por $p(\pi)$ o mínimo entre $p(ce(\pi))$ e $p(cd(\pi))$. Note que $p(\pi) = 0$ se, se somente se, $\pi = \iota$.

Exemplo 17. *Seja* π *a permutação do exemplo anterior. Nós temos que* 1, 2, 1 *e* 4 *são platôs em* $ce(\pi)$ *e que* 4, 2 *e* $1\ 1$ *são platôs em* $cd(\pi)$. *Então,* $p(\pi) = \min\{p(ce(\pi)), p(cd(\pi))\} = \min\{4, 3\} = 3$.

Benoît-Gagné e Hamel [5] mostraram que é sempre possível diminuir o número de platôs de um código (esquerdo ou direito) em uma unidade por meio da aplicação de uma transposição. Seja π uma permutação sem sinal, $\pi \neq \iota$, e seja $ce(\pi_i)\ ce(\pi_{i+1})\ \ldots\ ce(\pi_j)$ o platô mais a esquerda de $ce(\pi)$. Claramente, $ce(\pi_k) = 0$ para todo $1 \leq k \leq i-1$ e, portanto, $\pi_1 < \pi_2 < \cdots < \pi_{i-1}$. Desse modo, se $ce(\pi_i) = v$, a transposição $t(i-v, i, j+1)$ elimina o platô mais a esquerda de π. De modo análogo, seja $cd(\pi_i)\ cd(\pi_{i+1})\ \ldots\ cd(\pi_j)$ o platô mais a direita de $cd(\pi)$. Claramente, $cd(\pi_k) = 0$ para todo $j+1 \leq k \leq n$ e, portanto, $\pi_{j+1} < \pi_{j+2} < \cdots < \pi_n$. Desse modo, se $cd(\pi_i) = v$, a transposição $t(i, j+1, j+1+v)$ elimina o platô mais a direita de π.

Dado que $p(\pi)$ representa o valor mínimo entre $p(ce(\pi))$ e $p(cd(\pi))$, é possível ordenar π aplicando no máximo $p(\pi)$ transposições. Assim, Benoît-Gagné e Hamel [5] propuseram um algoritmo simples para aproximar a distância de transposição que apenas computa o valor de $p(\pi)$. Tal algoritmo está descrito abaixo.

Embora Benoît-Gagné e Hamel [5] tenham afirmado que o Algoritmo 27 é uma 3-aproximação, nós acreditamos que eles não apresentaram uma prova completa para tal afirmação. Eles demonstraram que o Algoritmo 27 possui fator de aproximação

$$\frac{c \cdot p(\pi)}{b_t(\pi)}, \text{ sendo } c = \frac{3\lfloor \frac{b_t(\pi)}{3} \rfloor + b_t(\pi) \mod 3}{\lceil \frac{b_t(\pi)}{3} \rceil},$$

Algorithm 27: Algoritmo 3-aproximado proposto por Benoît-Gagné e Hamel [5]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de transposições aplicadas para ordenar π.

1 Compute $ce(\pi)$;
2 Compute $cd(\pi)$;
3 $i \leftarrow p(ce(\pi))$;
4 $j \leftarrow p(cd(\pi))$;
5 $d \leftarrow \min\{i, j\}$;
6 **retorna** d;

e que $c \leq 3$ (basta tomar o limite de $b_t(\pi)$), mas ficou faltando demonstrar que $p(\pi) \leq b_t(\pi)$. Tal demonstração é dada pelo Lema 14.

Lema 14. *Para toda permutação $\pi \in S_n$, nós temos que $p(\pi) < b_t(\pi)$.*

Demonstração. Seja $\pi \in S_n$, $\pi \neq \iota$, s_1 a primeira *strip* de π e s_m a última *strip* de π, $m \leq n$. Se $\pi_1 = 1$, então $ce(\pi_i) = cd(\pi_i) = 0$ para qualquer elemento $\pi_i \in s_1$. Logo, os elementos pertencentes a s_1 não afetam o valor de $p(\pi)$. O mesmo pode ser observado para o elementos pertencentes a s_m caso $\pi_n = n$. Isso significa que se reduzirmos π para uma permutação irredutível σ, então $p(\pi) = p(\sigma)$. É fácil notar que $b_t(\pi) = b_t(\sigma)$, portanto podemos restringir nossa análise para permutações irredutíveis.

Seja $\gamma \in S_n^*$ e seja $\gamma_i\ \gamma_{i+1} \ldots \gamma_j$ uma *strip* de γ. Nós temos que $ce(\gamma_{k+1}) = ce(\gamma_k)$ e $cd(\gamma_{k+1}) = cd(\gamma_k)$ para todo $i \leq k < j$. Isso implica que, com respeito a $ce(\gamma)$ e $cd(\gamma)$, os elementos pertencentes a uma *strip* de γ ou possuem valor zero ou pertencem a um mesmo platô. Assim, $s_t(\gamma) \geq p(ce(\gamma))$ e $s_t(\gamma) \geq p(cd(\gamma))$, portanto $s_t(\gamma) \geq p(\gamma)$. Como $b_t(\gamma) > s_t(\gamma)$ (Lema 13), o lema está provado. □

Quanto à complexidade de tempo, podemos computar $ce(\pi)$ e $cd(\pi)$ em tempo $O(n^2)$. Dado que computar $p(ce(\pi))$ e $p(ce(\pi))$ toma tempo $O(n)$ e computar $\min\{i, j\}$ toma tempo $O(1)$, podemos concluir que o Algoritmo 27 roda em tempo $O(n^2)$.

Uma subsequência crescente de uma permutação sem sinal π é uma subsequência $\pi_{i_1}\ \pi_{i_2} \ldots \pi_{i_j}$ de elementos não necessariamente contíguos de π tal que $i_k < i_{k+1}$ e $\pi_{i_k} < \pi_{i_{k+1}}$ para todo $0 < k < j$. Uma subsequência crescente máxima de π é uma subsequência crescente de π de tamanho máximo. O conjunto de elementos pertencentes a uma subsequência crescente máxima de π é denotado por SCM(π). É fácil notar que |SCM(π)| $= n$ se, e somente se, $\pi = \iota$.

Exemplo 18. *Seja $\pi = (2\ 3\ 1\ 5\ 4)$. As subsequências crescentes $2\ 3\ 5$ e $2\ 3\ 4$ são máximas, portanto SCM(π) $= \{2, 3, 5\}$ ou SCM(π) $= \{2, 3, 4\}$.*

Guyer, Heath e Vergara [37] desenvolveram um algoritmo guloso baseado na subsequência crescente máxima de uma permutação $\pi \in S_n$. A cada iteração, o algoritmo seleciona, dentre

todas as $\binom{n}{3}$ transposições possíveis, a transposição $t(i, j, k)$ tal que $|\text{SCM}(\pi \circ t(i, j, k))|$ é máximo. Nós dizemos que a transposição que satisfaz essa escolha gulosa é uma transposição gulosa.

Sabendo que pode existir mais do que uma transposição gulosa para ser selecionada a cada iteração, a qualidade das respostas desse algoritmo pode variar dependendo da regra definida para selecionar uma transposição gulosa. Guyer, Heath e Vergara [37] não especificaram nenhuma regra e tampouco apresentaram um fator de aproximação para o algoritmo. Em razão disso, decidimos definir uma regra que pudesse nos ajudar a demonstrar um fator de aproximação.

Nós dizemos que a transposição $t(i, j, k)$ não corta uma *strip* de uma permutação sem sinal π se os pares de elementos adjacentes (π_{i-1}, π_i), (π_{j-1}, π_j) e (π_{k-1}, π_k) são *breakpoints*. A regra que definimos é: somente transposições gulosas que não cortam *strips* devem ser aplicadas. O Lema 15 mostra que sempre existe uma transposição satisfazendo essa regra.

O algoritmo que desenvolvemos está descrito abaixo (Algoritmo 28). O Teorema 5 prova que ele é uma 3-aproximação.

Algorithm 28: Algoritmo 3-aproximado baseado na estratégia gulosa proposta por Guyer, Heath, e Vergara [37]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de transposições aplicadas para ordenar π.

```
1 d ← 0;
2 enquanto π ≠ ι faça
3     Seja t(i, j, k) uma transposição gulosa que não corta nenhuma strip de π;
4     π ← π ∘ t(i, j, k);
5     d ← d + 1;
6 fim
7 retorna d
```

Lema 15. *Se π é uma permutação sem sinal tal que $\pi \neq \iota$, então existe uma transposição gulosa que não corta nenhuma* strip *de π.*

Demonstração. Seja $t(i, j, k)$ uma transposição gulosa e seja π' a permutação tal que $\pi' = \pi \circ t(i, j, k)$. Se $t(i, j, k)$ não corta nenhum *strip* de π, então o lema está provado. Caso contrário, temos que considerar três casos:

(a) (π_{i-1}, π_i) não é um *breakpoint*. Nesse caso, seja i' o maior número inteiro tal que $i' < i$ e $(\pi_{i'-1}, \pi_{i'})$ é um *breakpoint* e seja π'' a permutação tal que $\pi'' = \pi \circ t(i', j, k)$. Então, temos quatro subcasos para analisar:

(i) $\pi_i \in$ SCM(π') e $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \in$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \in$ SCM(π''), portanto $\rho(i', j, k)$ também é uma transposição gulosa pois $|$SCM(π'')$| = |$SCM(π')$|$.

(ii) $\pi_i \in$ SCM(π') e $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \notin$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \in$ LIS(π''), portanto $|$LIS(π'')$| > |$LIS(π')$|$ e isso contradiz a nossa hipótese de que $t(i, j, k)$ é uma transposição gulosa.

(iii) $\pi_i \notin$ SCM(π') e $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \in$ SCM(π'). Nesse subcaso, nós temos que $|$SCM($\rho(i+1, j, k) \cdot \pi$)$| > |$SCM($\pi'$)$|$ e isso contradiz a nossa hipótese de que $t(i, j, k)$ é uma transposição gulosa.

(iv) $\pi_i \notin$ SCM(π') e $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \notin$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \notin$ SCM(π''), portanto $t(i', j, k)$ também é uma transposição gulosa pois $|$SCM(π'')$| = |$SCM(π')$|$.

(b) (π_{j-1}, π_j) não é um *breakpoint*. Nesse caso, seja j' o menor número inteiro tal que $j < j'$ e $(\pi_{j'-1}, \pi_{j'})$ é um *breakpoint* e seja j'' o maior número inteiro tal que $j'' < j$ e $(\pi_{j''-1}, \pi_{j''})$ é um *breakpoint*. Note que é possível j' = k ou j'' = i, mas é impossível que j' = k e j'' = i, senão $t(i, j, k)$ iria mover apenas elementos pertencentes a uma mesma *strip*; consequentemente, $|$SCM(π')$| \leq |$SCM(π)$|$ e $t(i, j, k)$ não seria uma transposição gulosa. Também note que a situação em que $\pi_{j-1} \in$ SCM(π') e $\pi_j \in$ SCM(π') não pode ocorrer devido à definição de uma subsequência crescente.

Então, se assumirmos que $j' \neq k$ e que π'' é a permutação tal que $\pi'' = \pi \circ t(i, j', k)$, nós temos três subcasos para analisar:

(i) $\pi_{j-1} \in$ SCM(π') e $\{\pi_j, \pi_{j+1}, \ldots, \pi_{j'-1}\} \notin$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_j, \pi_{j+1}, \ldots, \pi_{j'-1}\} \in$ SCM(π''), portanto $|$SCM(π'')$| > |$SCM(π')$|$ e isso contradiz a nossa hipótese de que $t(i, j, k)$ é uma transposição gulosa.

(ii) $\pi_{j-1} \notin$ SCM(π') e $\{\pi_j, \pi_{j+1}, \ldots, \pi_{j'-1}\} \in$ SCM(π'). Nesse subcaso, nós temos que $|$SCM($\rho(i, j-1, k) \cdot \pi$)$| > |$SCM($\pi'$)$|$ e isso contradiz a nossa hipótese de que $t(i, j, k)$ é uma transposição gulosa.

(iii) $\pi_{j-1} \notin$ SCM(π') e $\{\pi_j, \pi_{j+1}, \ldots, \pi_{j'-1}\} \notin$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_j, \pi_{j+1}, \ldots, \pi_{j'-1}\} \notin$ SCM(π''), portanto $t(i, j', k)$ também é uma transposição gulosa pois $|$SCM(π'')$| = |$SCM(π')$|$.

Por outro lado, se assumirmos que $j'' \neq i$ e que π'' é a permutação tal que $\pi'' = \pi \circ t(i, j'', k)$, nós temos três subcasos para analisar:

(i) $\pi_j \in$ SCM(π') e $\{\pi_{j''}, \pi_{j''+1}, \ldots, \pi_{j-1}\} \notin$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_{j''}, \pi_{j''+1}, \ldots, \pi_{j-1}\} \in$ SCM(π''), portanto $|$SCM(π'')$| > |$SCM(π')$|$ e isso contradiz a nossa hipótese de que $t(i, j, k)$ é uma transposição gulosa.

(ii) $\pi_j \notin$ SCM(π') e $\{\pi_{j''}, \pi_{j''+1}, \ldots, \pi_{j-1}\} \in$ SCM(π'). Nesse subcaso, nós temos que $|$SCM($\rho(i, j+1, k) \cdot \pi$)$| > |$SCM($\pi'$)$|$ e isso contradiz a nossa hipótese de que $t(i, j, k)$ é uma transposição gulosa.

(iii) $\pi_j \notin$ SCM(π') e $\{\pi_{j''}, \pi_{j''+1}, \ldots, \pi_{j-1}\} \notin$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_{j''}, \pi_{j''+1}, \ldots, \pi_{j-1}\} \notin$ SCM(π''), portanto $t(i, j'', k)$ também é uma transposição gulosa pois $|$SCM(π'')$| = |$SCM(π')$|$.

(c) (π_{k-1}, π_k) não é um *breakpoint*. Nesse caso, seja k' o menor número inteiro tal que $k < k'$ e $(\pi_{k'-1}, \pi_{k'})$ é um *breakpoint* e seja π'' a permutação tal que $\pi'' = \pi \circ t(i, j, k')$. Então, temos quatro subcasos para analisar:

(i) $\pi_{k-1} \in$ SCM(π') e $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \in$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \in$ SCM(π''), portanto $t(i, j, k')$ também é uma transposição gulosa pois $|$LIS(π'')$| = |$LIS(π')$|$.

(ii) $\pi_{k-1} \in$ SCM(π') e $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \notin$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \in$ SCM(π''), portanto $|$SCM(π'')$| > |$SCM(π')$|$ e isso contradiz a nossa hipótese de que $t(i, j, k)$ é uma transposição gulosa.

(iii) $\pi_{k-1} \notin$ SCM(π') e $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \in$ SCM(π'). Nesse subcaso, nós temos que $|$SCM($\rho(i, j, k-1) \cdot \pi$)$| > |$SCM($\pi'$)$|$ e isso contradiz a nossa hipótese de que $t(i, j, k)$ é uma transposição gulosa.

(iv) $\pi_{k-1} \notin$ SCM(π') e $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \notin$ SCM(π'). Nesse subcaso, nós temos que $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \notin$ SCM(π''), portanto $t(i, j, k')$ também é uma transposição gulosa pois $|$SCM(π'')$| = |$SCM(π')$|$.

Embora os casos (a), (b) e (c) possam ocorrer simultaneamente, eles são independentes uns dos outros, de tal forma que se a transposição $t(i, j, k)$ corta uma *strip* de π, então é possível derivar uma transposição gulosa que não corta nenhuma *strip* de π. □

Teorema 5. *O Algoritmo 28 é uma* 3*-aproximação.*

Demonstração. Para determinar um limite superior para o número de transposições aplicadas pelo Algoritmo 28, nós definimos um algoritmo simples, chamado *SomaStrip*, que recebe como entrada uma permutação sem sinal $\pi \in S_n$ e procede da seguinte maneira. Primeiramente, ele ordena todas as *strips* de π com respeito ao tamanho delas, obtendo uma lista de *strips* s^0, s^1, $\ldots$, $s^{s_t(\pi)-1}$ tal que $|s^i| \geq |s^{i+1}|$ para todo $0 \leq i < s_t(\pi) - 1$. Depois, ele inicializa uma variável chamada SOMA com valor $|s^0|$. Finalmente, começando por s^1, ele itera sobre lista de *strips* de tal modo que, na iteração i, o valor da variável SOMA é acrescido de $|s^i|$.

Seja SOMA$_i$ o valor da variável SOMA na iteração i, sendo SOMA$_0 = |s^0|$. Claramente, SOMA$_i$ = SOMA$_{i-1} + |s^i|$ para todo $1 \leq i \leq s_t(\pi) - 1$. Ademais, quando o algoritmo termina, SOMA = SOMA$_{s_t(\pi)-1} = |s^0| + |s^1| + \cdots + |s^{s_t(\pi)}| = n$.

Agora, assuma que π foi dada como entrada para o Algoritmo 28 e seja π^i a permutação produzida após i iterações, sendo $\pi^0 = \pi$. Podemos provar por indução que $|\text{SCM}(\pi^i)| \geq \text{SOMA}_i$. Para o caso base, temos que $|\text{SCM}(\pi^0)| \geq \text{SOMA}_0$ porque, por definição, $|\text{SCM}(\pi^0)|$ deve ser igual ou maior do que o tamanho de qualquer *strip* de π^0. Para o passo da indução, vamos assumir que $|\text{SCM}(\pi^k)| \geq \text{SOMA}_k$ para todo $0 \leq k \leq i$.

Como o Algoritmo 28 nunca corta uma *strip*, todas as *strips* de π^i são formadas por *strips* de π^0. Seja s' a *strip* de maior tamanho dentre todas as *strips* de π^0 cujos elementos não pertencem a uma determinada SCM(π^i). Nós temos que $|\text{SCM}(\pi^{i+1})| \geq |\text{SCM}(\pi^i)| + |s'|$ porque é possível aplicar uma transposição em π^i e obter uma nova permutação contendo uma subsequência crescente formada pelos elementos de s' e SCM(π^i). Se $|s'| \geq |s^{i+1}|$, então $|\text{SCM}(\pi^{i+1})| \geq |\text{SCM}(\pi^i)| + |s'| \geq \text{SOMA}_i + |s^{i+1}| = \text{SOMA}_{i+1}$. Caso contrário, se $|s'| < |s^{i+1}|$, isso significa que os elementos de todas as *strips* s^t, $0 \leq t \leq i+1$, pertencem a SCM(π^i), portanto $|\text{SCM}(\pi^{i+1})| > |\text{SCM}(\pi^i)| \geq \text{SOMA}_{i+1}$.

A desigualdade $|\text{SCM}(\pi^i)| \geq \text{SOMA}_i$ implica que, no pior caso, o Algoritmo 28 faz o valor $|\text{SCM}(\pi)|$ se igualar a n iterando tantas vezes quanto o Algoritmo *SomaStrip* itera para fazer o valor da variável SOMA se igualar a n. Logo, denotando por $A_{28}(\pi)$ o número de transposições aplicadas pelo Algoritmo 28 para ordenar π, nós temos que $A_{28}(\pi) \leq s_t(\pi) - 1$.

Se $\pi_1 = 1$, é fácil notar que os elementos da primeira *strip* de π irão pertencer a SCM(π^i) para todo $0 \leq i \leq A_{28}(\pi)$. Ou seja, o Algoritmo 28 nunca irá aplicar uma transposição que move os elementos da primeira *strip* de π se $\pi_1 = 1$. O mesmo pode ser observado para os elementos pertencentes à última *strip* de π caso $\pi_n = n$. Por essa razão, se reduzirmos π para uma permutação irredutível σ, teremos que $A_{28}(\pi) = A_{28}(\sigma)$. Esse fato nos permite restringir nossa análise às permutações irredutíveis.

Seja $\gamma \in S_n^*$. Nós temos que $A_{28}(\gamma) \leq s_t(\gamma) - 1$. Como $s_t(\gamma) = b_t(\gamma) - 1$ (Lema 13), nós podemos concluir que $A_{28}(\gamma) \leq b_t(\gamma) - 2$. Isso significa que $A_{28}(\gamma) \leq 3d_t(\gamma)$ pois $d_t(\gamma) \geq \frac{b_t(\gamma)}{3}$ (Lema 12). □

Quanto à complexidade de tempo do Algoritmo 28, nós temos que existem $O(n^3)$ transposições a serem consideradas por iteração, o tamanho da subsequência crescente máxima de uma permutação pode ser computada em tempo $O(n \log n)$ [25], determinar se uma transposição corta uma *strip* toma tempo $O(1)$ e o laço **enquanto** executa $O(n)$ vezes. Logo, o Algoritmo 28 roda em tempo $O(n^5 \log n)$.

3.6 Problema da Ordenação por Transposições de Prefixo

O Problema da Ordenação por Transposições de Prefixo consiste-se em determinar a menor sequência de transposições de prefixo que ordena uma determinada permutação sem sinal π. O tamanho dessa sequência é denominado como a distância de transposição de prefixo de π,

denotada por $d_{tp}(\pi)$. Nesta seção, iremos apresentar dois algoritmos aproximados para o Problema da Ordenação por Transposições de Prefixo: o algoritmo 2-aproximado proposto por Dias e Meidanis [16] e uma versão gulosa desse algoritmo proposta por nós.

Um *breakpoint* de uma permutação π em S_n é um par de elementos adjacentes (π_i, π_{i+1}) tal que $\pi_{i+1} - \pi_i \neq 1$, $1 \leq i \leq n$. O par (π_0, π_1) nunca é considerado como um *breakpoint*. O número de *breakpoints* de π é denotado por $b_{tp}(\pi)$.

Exemplo 19. *Seja* $\pi = (0\ 1\ 3\ 4\ 2\ 5\ 6)$ *uma permutação estendida. Nós temos que os pares* $(1, 3)$, $(4, 2)$ *e* $(2, 5)$ *são* breakpoints, *portanto* $b_{tp}(\pi) = 3$.

Note que $b_{tp}(\pi) = 0$ se, e somente se, $\pi = \iota$. Pelo fato de uma transposição de prefixo remover no máximo dois *breakpoints* de uma permutação sem sinal, o seguinte lema pode ser derivado trivialmente.

Lema 16. *Para toda permutação sem sinal* π, $d_{tp}(\pi) \geq \frac{b_{tp}(\pi)}{2}$.

Uma *strip* de π é uma sequência de elementos $\pi_i\ \pi_{i+1} \ldots \pi_j$, $1 \leq i \leq j \leq n$, tal que (π_{i-1}, π_i) e (π_j, π_{j+1}) são *breakpoints* e nenhum par (π_k, π_{k+1}), $i \leq k \leq j - 1$, é um *breakpoint*.

Exemplo 20. *Seja* π *a permutação do Exemplo 19. Nós temos que* 1, $3\ 4$, 2 *e* 5 *são* strips *de* π. *Note que os elementos adicionados para estender* π *não são considerados.*

Dias e Meidanis [16] mostraram que é sempre possível remover um *breakpoint* de uma permutação sem sinal diferente da identidade aplicando uma transposição de prefixo. Seja $\pi \in S_n$ e seja $\pi_1 \ldots \pi_i$ a primeira *strip* de π. Se $\pi_i < n$, então existe uma *strip* de π que começa com o elemento $\pi_j = \pi_i + 1$ tal que $i < j - 1$ e (π_{j-1}, π_j) é um *breakpoint*. Assim, a transposição de prefixo $tp(i + 1, j)$ remove este *breakpoint*. Caso contrário, se $\pi_i = n$, então a transposição de prefixo $tp(i + 1, n + 1)$ remove o *breakpoint* (π_n, π_{n+1}). O Algoritmo 29 é o algoritmo derivado dessa análise.

Como o Algoritmo 29 remove um *breakpoint* a cada transposição de prefixo aplicada e $d_{tp}(\pi) \geq \frac{b_{tp}(\pi)}{2}$ (Lema 16), nós podemos concluir que ele é uma 2-aproximação. Quanto à complexidade de tempo, temos que as linhas 1, 4, 6 e 10 executam em tempo $O(1)$, as linhas 3, 5, 8 e 9 executam em tempo $O(n)$ e o lanço **enquanto** executa $O(n)$ vezes, portanto o Algoritmo 29 roda em tempo $O(n^2)$.

Dias e Meidanis [16] também mostraram que existe no máximo uma transposição de prefixo que elimina dois *breakpoint* de uma permutação sem sinal. Seja $\pi \in S_n$ e seja $tp(i, j)$ uma transposição que elimina dois *breakpoints* de π. Nós temos que $\pi \circ tp(i, j) = (\pi_i \ldots \pi_{j-1}\ \pi_1 \ldots \pi_{i-1}\ \pi_j \ldots \pi_n)$, sendo $\pi_{i-1} \neq \pi_i - 1$, $\pi_{j-1} \neq \pi_j - 1$, $\pi_{j-1} = \pi_1 - 1$ e $\pi_{i-1} = \pi_j - 1$. Logo, π_1 determina univocamente qual deve ser o valor de j e j determina univocamente qual deve ser o valor de i.

Algorithm 29: Algoritmo 2-aproximado proposto por Dias e Meidanis [16]

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de transposições de prefixo aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto π ≠ ι faça
3      Seja π_i o último elemento da primeira strip de π;
4      se π_i = n então
5          π ← π ∘ tp(i + 1, n + 1);
6          d ← d + 1;
7      senão
8          Seja π_j o elemento de π tal que π_j = π_i + 1;
9          π ← π ∘ tp(i + 1, j);
10         d ← d + 1;
11     fim
12 fim
13 retorna d;
```

Algorithm 30: Versão gulosa do Algoritmo 29

Entrada: Uma permutação $\pi \in S_n$.
Saída: Número de transposições de prefixo aplicadas para ordenar π.

```
1  d ← 0;
2  enquanto π ≠ ι faça
3      Seja j a posição do elemento π_1 − 1;
4      j ← j + 1;
5      Seja i a posição do elemento π_j − 1;
6      i ← i + 1;
7      se i > 1 e i < j então
8          π ← π ∘ tp(i, j);
9          d ← d + 1;
10     senão
11         Execute as linhas 3-11 do Algoritmo 29;
12     fim
13 fim
14 retorna d;
```

O Algoritmo 30 é uma versão gulosa do Algoritmo 29, isto é, antes do algoritmo eliminar um *breakpoint*, ele tenta eliminar dois. No pior caso, o Algoritmo 30 se comporta como o Algoritmo 29, portanto ele também é uma 2-aproximação. Quanto à complexidade de tempo, temos que as linhas 1, 4, 6 e 9 executam em tempo $O(1)$, as linhas 3, 5, 8 e 11 executam em tempo $O(n)$ e o lanço **enquanto** executa $O(n)$ vezes, portanto o Algoritmo 29 também roda em tempo $O(n^2)$.

Capítulo 4

Auditoria dos Algoritmos de Rearranjo de Genomas

Este capítulo apresenta e analisa os resultados que obtivemos ao auditar os algoritmos de rearranjo de genomas apresentados no capítulo anterior. A apresentação está dividida de tal forma que cada seção deste capítulo contém os resultados obtidos para os algoritmos que resolvem uma mesma variação do Problema da Ordenação por Rearranjo.

4.1 Problema da Ordenação por Reversões

Os resultados da auditoria dos algoritmos 11 e 13 são dados pelas tabelas 4.1 e 4.2 respectivamente. Como podemos ver na Figura 4.1, o Algoritmo 13 apresentou resultados muito melhores do que o Algoritmo 11, algo que já era esperado dada a discrepância entre os fatores de aproximação dos algoritmos.

A razão máxima obtida para o Algoritmo 11 se igualou ao fator de aproximação teórico, que é igual a $\frac{n-1}{2}$, para $3 \leq n \leq 13$. De fato, o Teorema 6 mostra que o fator de aproximação do Algoritmo 11 é justo.

Teorema 6. *O fator de aproximação do Algoritmo 11 é justo.*

Demonstração. Seja $\pi \in S_n$ uma permutação tal que $\pi = (n\ 1\ 2\ \ldots\ n-1)$. Note que $b_r(\pi) = 3$, portanto $d_r(\pi) \geq 2$. Como $\pi \circ r(2, n) \circ r(1, n) = \iota$, nós podemos concluir que $d_r(\pi) = 2$. Por outro lado, o Algoritmo 11 irá ordenar π aplicando a sequência de $n-1$ reversões $r(1, 2)$, $r(2, 3)$, ..., $r(n-1, n)$, implicando que $A_{11}(\pi) = n-1$. Logo, nós temos que $\frac{A_{11}(\pi)}{d(\pi)} = \frac{n-1}{2}$ e o teorema está provado. □

Por outro lado, a razão máxima obtida para o Algoritmo 13 não se igualou ao fator de aproximação teórico, que é igual a 2, porém ela parece estar convergindo para isso. Por exemplo, para n ímpar e $n \geq 3$, temos que a razão máxima é dada por $\frac{2n-2}{n+1}$; para n par e $n \geq 4$,

Tabela 4.1: Resultado da auditoria do Algoritmo 11.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,17	1,00	1,00	100,00%
4	3	1,92	1,08	1,50	83,33%
5	4	2,72	1,14	2,00	70,00%
6	5	3,55	1,18	2,50	56,94%
7	6	4,41	1,20	3,00	45,50%
8	7	5,28	1,22	3,50	35,61%
9	8	6,17	1,23	4,00	27,54%
10	9	7,07	1,24	4,50	21,07%
11	10	7,98	1,24	5,00	15,95%
12	11	8,90	1,25	5,50	11,95%
13	12	9,82	1,25	6,00	8,88%

Tabela 4.2: Resultado da auditoria do Algoritmo 13.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,17	1,00	1,00	100,00%
4	3	1,75	1,00	1,00	100,00%
5	4	2,41	1,01	1,33	98,33%
6	5	3,12	1,03	1,33	91,67%
7	6	3,85	1,04	1,50	84,76%
8	8	4,60	1,05	1,50	78,18%
9	9	5,36	1,06	1,60	71,74%
10	10	6,13	1,07	1,60	65,39%
11	11	6,92	1,07	1,67	58,99%
12	12	7,73	1,08	1,67	52,68%
13	13	8,55	1,08	1,71	46,59%

temos que a razão máxima é dada por $\frac{2n-4}{n}$. Apesar de não termos conseguido encontrar uma família de permutações que confirme a convergência da razão máxima, nós conjecturamos que o fator de aproximação do Algoritmo 13 é justo.

Conjectura 1. *O fator de aproximação do Algoritmo 13 é justo.*

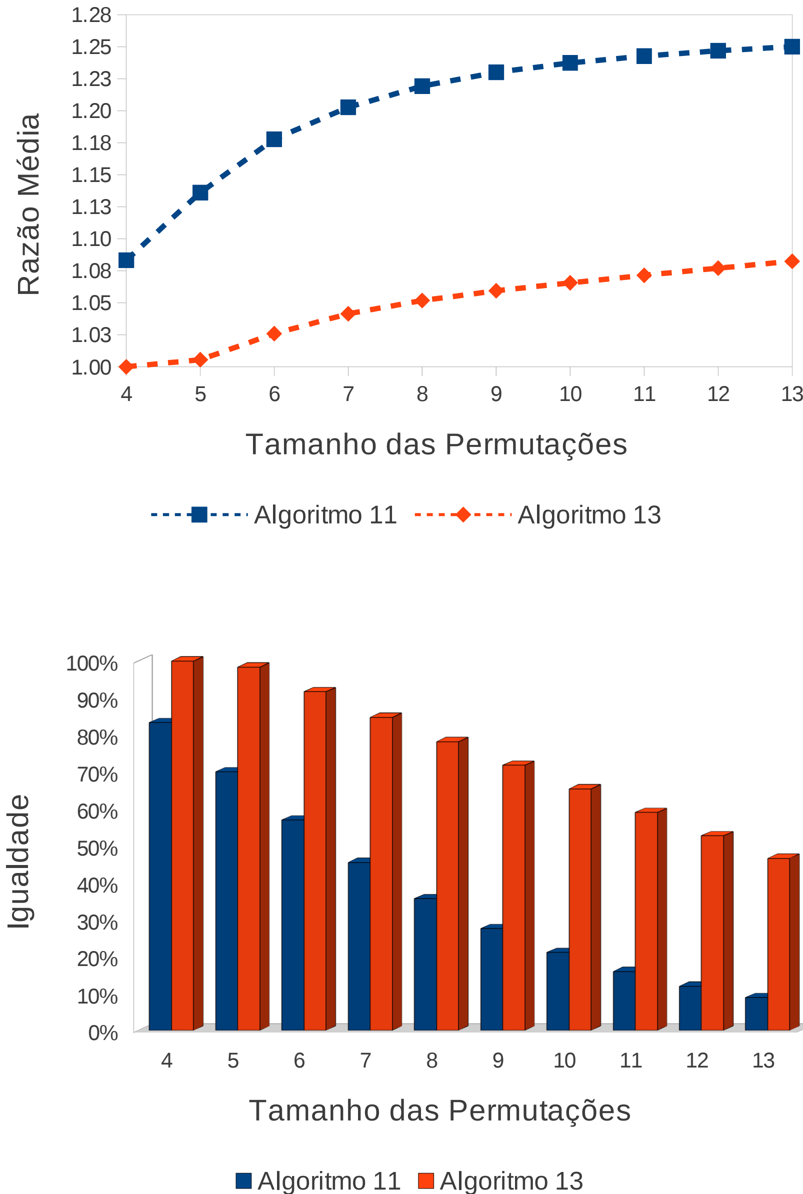


Figura 4.1: Comparação entre os algoritmos 11 e 13 com base nos resultados produzidos pelo GRAAu.

4.2 Problema da Ordenação por Reversões de Prefixo

Os resultados da auditoria dos algoritmos 14, 15 e 16 são dados pelas tabelas 4.3, 4.4 e 4.5 respectivamente. Como podemos ver na Figura 4.2, o Algoritmo 14 apresentou os piores resultados, enquanto que os algoritmos 15 e 16 apresentaram resultados praticamente iguais. Isso

indica que a prioridade dada às arestas azuis boas que satisfazem as condições 2 e 3 não influi de forma expressiva na qualidade das respostas do algoritmo proposto por Fischer e Ginzinger [24].

Tabela 4.3: Resultado da auditoria do Algoritmo 14.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	3	1,50	1,00	1,00	100,00%
4	5	2,63	1,04	1,67	91,67%
5	7	3,89	1,09	1,75	73,33%
6	9	5,27	1,14	2,00	52,36%
7	11	6,72	1,20	2,25	34,25%
8	13	8,24	1,24	2,25	20,62%
9	15	9,82	1,28	2,40	11,51%
10	17	11,43	1,32	2,60	5,98%
11	19	13,08	1,35	2,60	2,91%
12	21	14,75	1,38	2,67	1,33%
13	23	16,45	1,40	2,67	0,58%

Tabela 4.4: Resultado da auditoria do Algoritmo 15.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	3	1,50	1,00	1,00	100,00%
4	5	2,79	1,09	1,33	70,83%
5	8	4,01	1,12	1,75	62,50%
6	9	5,26	1,14	1,80	51,25%
7	12	6,50	1,15	1,83	42,54%
8	14	7,74	1,16	1,83	34,51%
9	15	8,98	1,17	2,00	27,75%
10	17	10,21	1,17	2,00	22,17%
11	19	11,45	1,18	2,00	17,63%
12	21	12,68	1,18	2,00	13,98%
13	23	13,91	1,19	2,00	11,07%

A razão máxima obtida para os algoritmos 14 e 16 não se igualou ao fator de aproximação teórico desses algoritmos, que são iguais a 3 e 2 respectivamente, porém ela parece estar con-

Tabela 4.5: Resultado da auditoria do Algoritmo 16.

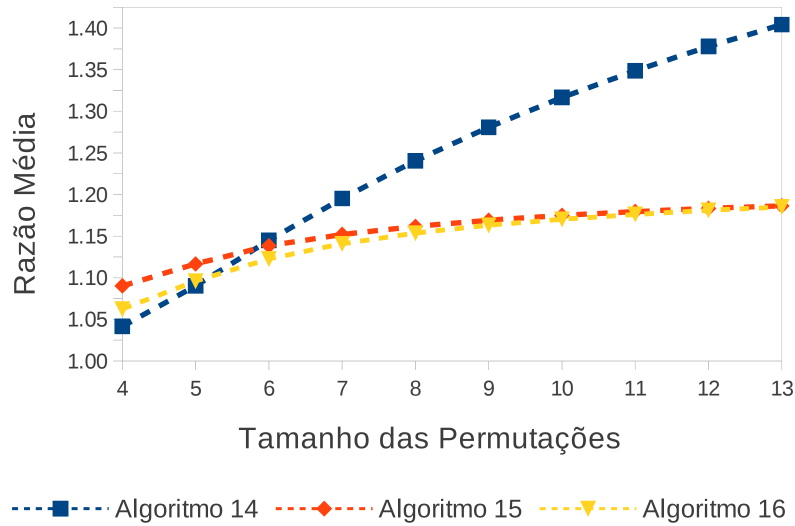
n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	3	1,50	1,00	1,00	100,00%
4	5	2,71	1,06	1,33	79,17%
5	7	3,93	1,10	1,75	69,17%
6	9	5,18	1,12	1,75	57,36%
7	11	6,44	1,14	1,75	47,66%
8	13	7,68	1,15	1,83	38,94%
9	15	8,93	1,16	1,86	31,61%
10	17	10,17	1,17	1,89	25,52%
11	19	11,41	1,18	1,90	20,54%
12	21	12,65	1,18	1,91	16,48%
13	23	13,89	1,19	1,91	13,20%

vergindo para isso. Apesar de não termos conseguidos encontrar uma família de permutações que confirme a convergência da razão máxima para nenhum deles, nós conjecturamos que o fator de aproximação de ambos algoritmos é justo.

Conjectura 2. *O fator de aproximação do Algoritmo 14 é justo.*

Conjectura 3. *O fator de aproximação do Algoritmo 16 é justo.*

Por outro lado, a razão máxima obtida para o Algoritmo 15 igualou-se ao fator de aproximação teórico, que é igual a 2, para $9 \leq n \leq 13$. De fato, o Teorema 7 mostra que o fator de aproximação do Algoritmo 15 é justo.



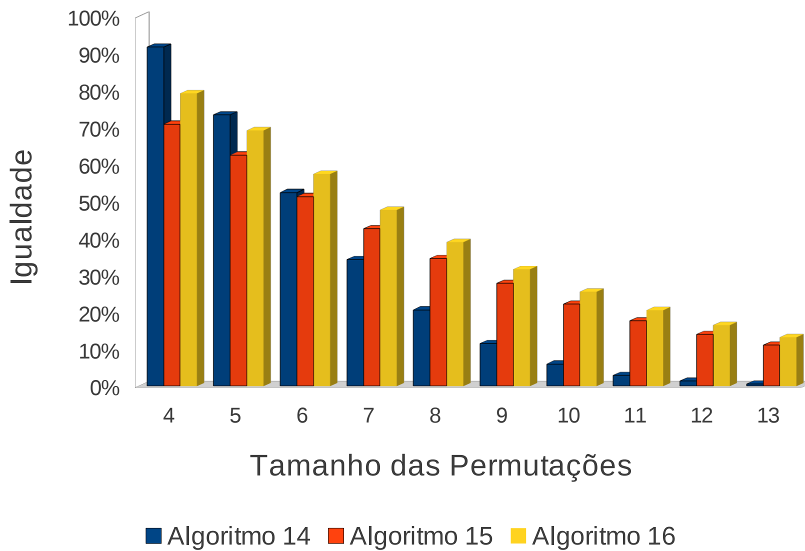


Figura 4.2: Comparação entre os algoritmos 14, 15 e 16 com base nos resultados produzidos pelo GRAAu.

Teorema 7. *O fator de aproximação do Algoritmo 15 é justo.*

Demonstração. Seja $\pi = (1\ 7\ 8\ 2\ 4\ 3\ 9\ 5\ 6)$ uma permutação sem sinal. Como $d_{rp}(\pi) \geq b_{rp}(\pi) = 6$ e a sequência de reversões de prefixo $rp(3)$, $rp(6)$, $rp(2)$, $rp(7)$, $rp(9)$, $rp(6)$ ordena π, nós podemos concluir que $d_{rp}(\pi) = 6$. Por outro lado, o Algoritmo 15 ordena π da seguinte maneira:

1. Note que $e = (\pi_1, \pi_4)$ é uma aresta azul (Figura 4.3a), porém ela não satisfaz a condição 1 pois (π_0, π_1) não é uma aresta vermelha. Logo, não existe uma aresta azul satisfazendo a condição 1. Em contrapartida, e satisfaz a condição 2 uma vez que as arestas (π_1, π_2) e (π_4, π_5) são vermelhas. Assim, o Algoritmo 15 aplica a reversão de prefixo $rp(4)$ seguida da reversão de prefixo $rp(3)$, produzindo a permutação $\pi = (7\ 8\ 2\ 1\ 4\ 3\ 9\ 5\ 6)$;

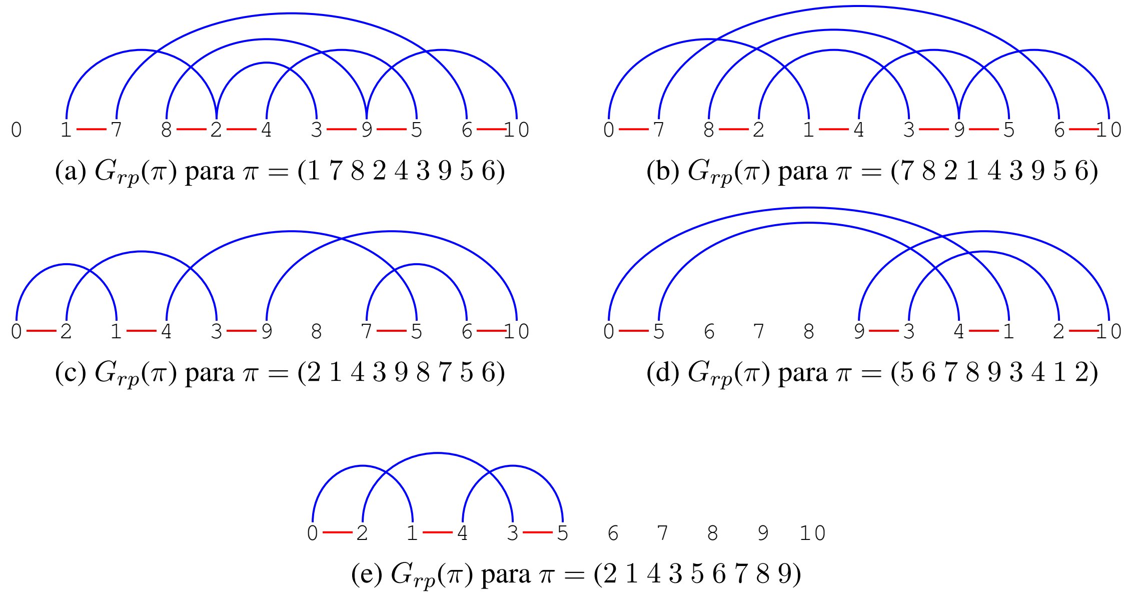
2. Note que $e = (\pi_1, \pi_9)$ é uma aresta azul (Figura 4.3b), porém ela não satisfaz a condição 1 pois (π_8, π_9) não é uma aresta vermelha. Como não existe uma aresta azul satisfazendo a condição 1, o algoritmo procura por uma aresta azul que satisfaça a condição 2. A aresta azul (π_2, π_7) satisfaz a condição 2 uma vez que as arestas (π_1, π_3) e (π_6, π_7) são vermelhas. Assim, o Algoritmo 15 aplica a reversão de prefixo $rp(7)$ seguida da reversão de prefixo $rp(5)$, obtendo a permutação $\pi = (2\ 1\ 4\ 3\ 9\ 8\ 7\ 5\ 6)$;

3. Note que $e = (\pi_1, \pi_4)$ é uma aresta azul (Figura 4.3c), porém ela não satisfaz a condição 1 pois (π_3, π_4) não é uma aresta vermelha. Como não existe uma aresta azul satisfazendo a condição 1, o algoritmo procura por uma aresta azul que satisfaça a condição 2. A aresta azul (π_7, π_9) satisfaz a condição 2 uma vez que as arestas (π_7, π_8) e (π_9, π_{10}) são vermelhas. Assim, o Algoritmo 15 aplica a reversão de prefixo $rp(9)$ seguida da reversão de prefixo $rp(2)$, obtendo a permutação $\pi = (5\ 6\ 7\ 8\ 9\ 3\ 4\ 1\ 2)$;

4. Note que $e = (\pi_1, \pi_7)$ é uma aresta azul (Figura 4.3d), porém ela não satisfaz a condição 1 pois (π_6, π_7) não é uma aresta vermelha. Como não existe uma aresta azul satisfazendo a condição 1, o algoritmo procura por uma aresta azul que satisfaça a condição 2. Como também não existem arestas azuis que satisfaçam a condição 2, ele irá procurar por uma aresta azul que satisfaça a condição 3. A aresta azul (π_5, π_{10}) satisfaz a condição 3 uma vez que as arestas (π_5, π_6) e (π_9, π_{10}) são vermelhas. Assim, o Algoritmo 15 aplica a reversão de prefixo $rp(5)$ seguida da reversão de prefixo $rp(9)$, obtendo a permutação $\pi = (2\ 1\ 4\ 3\ 5\ 6\ 7\ 8\ 9)$;

5. É fácil notar que π não possui uma aresta azul boa (Figura 4.3e). Sendo assim, seja $\sigma = (\pi_1\ \pi_2\ \pi_3\ \pi_4) = (2\ 1\ 4\ 3)$ a permutação formada pelos elementos fora de posição de π (note que os elementos π_5, π_6, π_7, π_8 e π_9 já estão na posição correta). Logo, o Algoritmo 15 irá ordenar π aplicando a sequência de $2b_{rp}(\sigma) = 4$ reversões de prefixo $rp(4)$, $rp(2)$, $rp(4)$ e $rp(2)$.

Assim, denotando por $A_{15}(\pi)$ o número de reversões de prefixo aplicadas pelo Algoritmo 15 para ordenar a permutação π, nós temos que $A_{15} = 12$. Seja $\gamma = (1\ 7\ 8\ 2\ 4\ 3\ 9\ 5\ 6\ 10\ 11\ \ldots\ n)$, $n \geq 10$, uma permutação sem sinal. Como os elementos γ_i, $10 \leq i \leq n$, estão na posição correta, o Algoritmo 15 irá ordenar γ da mesma forma que ele ordena π. Além disso, nós temos que $d_{rp}(\gamma) = d_{rp}(\pi) = 6$. Logo, $\frac{A_{15}(\gamma)}{d_{rp}(\gamma)} = \frac{12}{6} = 2$. $\square$



(a) $G_{rp}(\pi)$ para $\pi = (1\ 7\ 8\ 2\ 4\ 3\ 9\ 5\ 6)$

(b) $G_{rp}(\pi)$ para $\pi = (7\ 8\ 2\ 1\ 4\ 3\ 9\ 5\ 6)$

(c) $G_{rp}(\pi)$ para $\pi = (2\ 1\ 4\ 3\ 9\ 8\ 7\ 5\ 6)$

(d) $G_{rp}(\pi)$ para $\pi = (5\ 6\ 7\ 8\ 9\ 3\ 4\ 1\ 2)$

(e) $G_{rp}(\pi)$ para $\pi = (2\ 1\ 4\ 3\ 5\ 6\ 7\ 8\ 9)$

Figura 4.3: Grafos de *breakpoints* das permutações que são produzidas pelo Algoritmo 15 ao ordenar a permutação $\pi = (1\ 7\ 8\ 2\ 4\ 3\ 9\ 5\ 6)$.

Fischer e Ginzinger [24] também implementaram um algoritmo que utiliza a estratégia gulosa descrita na Seção 3.2, porém conduziram um experimento diferente para medir a qualidade das respostas desse algoritmo. Eles computaram a distância de reversão de prefixo de 10000 permutações aleatórias de tamanho até 71 utilizando um método de *branch-and-bound* e as compararam com as respostas produzidas pelo algoritmo.

Os resultados que eles obtiveram os levaram a considerar a hipótese de que o fator de aproximação teórico do algoritmo que eles implementaram poderia ser menor do que 2. Nossos resultados contrapõem essa hipótese, uma vez que implementamos dois algoritmos distintos que utilizam a estratégia gulosa descrita na Seção 3.2 e mostramos que o fator de aproximação de um deles é justo e que o fator de aproximação do outro dá claras evidências de que é justo.

4.3 Problema da Ordenação por Reversões de Prefixo com Sinal

Os resultados da auditoria dos algoritmos 17, 18 e 19 são dados pelas tabelas 4.6, 4.7 e 4.8 respectivamente. Como podemos ver na Figura 4.4, o Algoritmo 17 apresentou os piores resultados, enquanto que o Algoritmo 19 apresentou resultados muitos melhores em relação ao Algoritmo 18.

A razão máxima obtida para os algoritmos 17 e 19 não se igualou ao fator de aproximação

Tabela 4.6: Resultado da auditoria do Algoritmo 17.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	1	0,50	1,00	1,00	100,00%
2	4	2,00	1,00	1,00	100,00%
3	7	3,75	1,08	1,75	79,17%
4	10	5,66	1,17	2,25	52,86%
5	13	7,67	1,25	2,60	31,17%
6	16	9,76	1,32	2,67	16,47%
7	19	11,91	1,38	2,71	7,86%
8	22	14,11	1,43	2,75	3,43%
9	25	16,33	1,48	2,78	1,38%
10	28	18,59	1,52	2,80	0,52%

Tabela 4.7: Resultado da auditoria do Algoritmo 18.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	1	0,50	1,00	1,00	100,00%
2	4	2,00	1,00	1,00	100,00%
3	6	3,65	1,05	1,67	83,33%
4	8	5,38	1,12	1,75	60,94%
5	10	7,17	1,17	2,00	40,86%
6	12	8,99	1,22	2,00	25,01%
7	14	10,83	1,26	2,00	14,08%
8	16	12,70	1,29	2,00	7,34%
9	18	14,58	1,32	2,00	3,56%
10	20	16,48	1,34	2,00	1,61%

teórico, que é igual a 3 e 2 respectivamente, porém ela parece estar convergindo para isso. Por exemplo, para $5 \leq n \leq 10$, nós temos que a razão máxima obtida para o Algoritmo 17 é dada por $\frac{3n-2}{n}$ e a razão máxima obtida para o Algoritmo 19 é dada por $\frac{2n+1}{n}$. Apesar de não termos conseguido encontrar uma família de permutações que confirme a convergência da razão máxima de nenhum dos dois algoritmos, nós conjecturamos que o fator de aproximação de ambos é justo.

Conjectura 4. *O fator de aproximaçãod o Algoritmo 17 é justo.*

Conjectura 5. *O fator de aproximaçãod o Algoritmo 19 é justo.*

Tabela 4.8: Resultado da auditoria do Algoritmo 19.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	1	0,50	1,00	1,00	100,00%
2	4	2,00	1,00	1,00	100,00%
3	6	3,50	1,01	1,25	93,75%
4	8	5,00	1,04	1,40	82,29%
5	10	6,50	1,06	1,67	69,90%
6	12	7,99	1,08	1,71	57,38%
7	14	9,49	1,10	1,75	45,80%
8	16	11,00	1,11	1,78	35,71%
9	18	12,50	1,13	1,80	27,29%
10	20	14,01	1,14	1,82	20,49%

Por outro lado, a razão máxima obtida para o Algoritmo 18 se igualou ao fator de aproximação teórico para $5 \leq n \leq 10$. De fato, o Teorema 8 mostra que o fator de aproximação do Algoritmo 18 é justo.

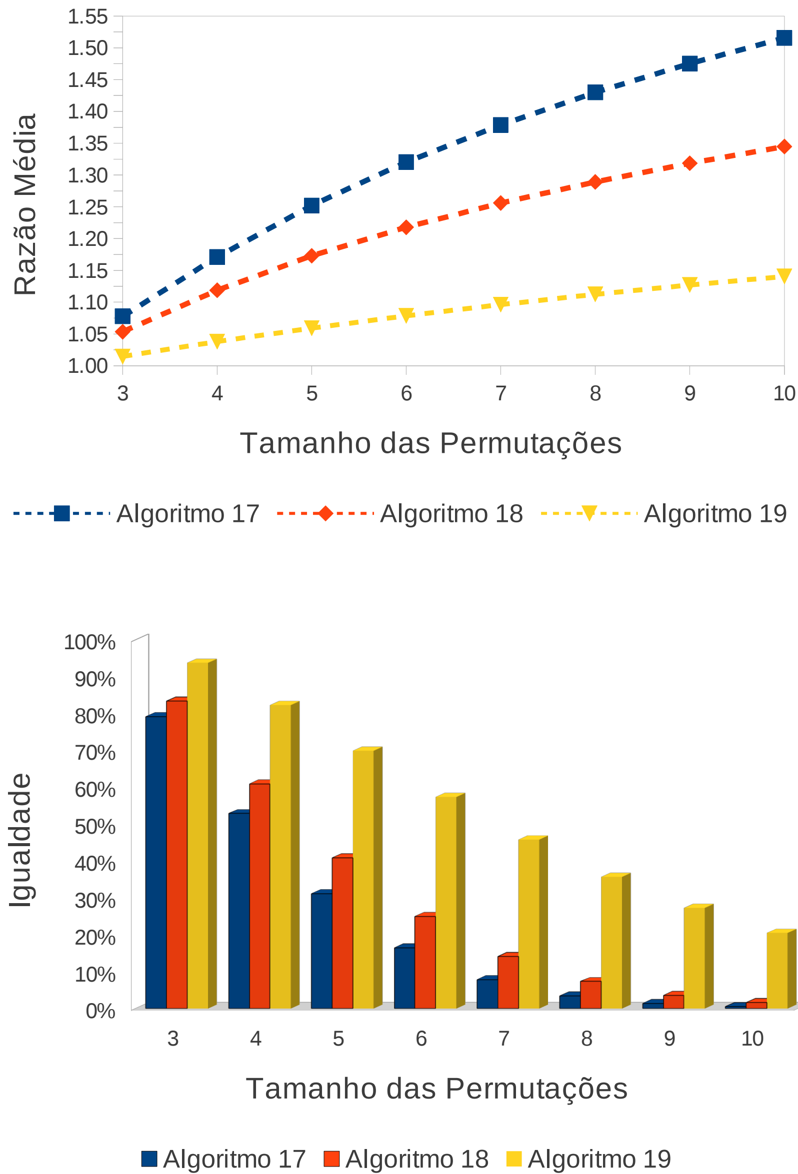


Figura 4.4: Comparação entre os algoritmos 17, 18 e 19 com base nos resultados produzidos pelo GRAAu.

Teorema 8. *O fator de aproximação do Algoritmo 18 é justo.*

Demonstração. Seja $\pi = (-5\ +1\ +4\ -3\ +2)$ uma permutação com sinal. Como $d_{rps}(\pi) \geq b_{rps}(\pi) = 5$ e a sequência de reversões de prefixo com sinal $rps(5)$, $rps(1)$, $rps(3)$, $rps(1)$, $rps(4)$ ordena π, nós podemos concluir que $d_{rps}(\pi) = 5$. Por outro lado, o Algoritmo 18 ordena π da seguinte maneira:

1. Note que π possui mais do que um elemento positivo fora de posição e o maior deles é $\pi_3 = +4$. Como $\pi_1 = -5$, nós temos que o Algoritmo 18 aplica a reversão de prefixo com sinal $rps(3)$ seguida da reversão de prefixo com sinal $rps(2)$, produzindo a permutação $\pi = (+1\ +4\ +5\ -3\ +2)$;

2. Note que π possui mais do que um elemento positivo fora de posição e o maior deles é $\pi_3 = +5$. Como π_3 é o elemento fora de posição com o maior valor absoluto, nós temos que o Algoritmo 18 aplica a reversão de prefixo com sinal $rps(3)$ seguida da reversão de prefixo com sinal $rps(5)$, produzindo a permutação $\pi = (-2\ +3\ +1\ +4\ +5)$;

3. Note que π possui mais do que um elemento positivo fora de posição e o maior deles é $\pi_2 = +3$. Como π_2 é o elemento fora de posição com o maior valor absoluto, nós temos que o Algoritmo 18 aplica a reversão de prefixo com sinal $rps(2)$ seguida da reversão de prefixo com sinal $rps(3)$, produzindo a permutação $\pi = (-1\ -2\ +3\ +4\ +5)$;

4. Note que π não possui um elemento positivo fora de posição. Assim, seja $\sigma = (-1\ -2)$ a permutação formada apenas pelos elementos fora de posição de π. Claramente, σ possui a forma descrita pelo Lema 5, portanto o Algoritmo 18 a ordena aplicando a sequência de $2b_{rps}(\pi) = 4$ reversões de prefixo com sinal $rps(2)$, $rps(1)$, $rps(2)$ e $rps(1)$.

Assim, denotando por $A_{18}(\pi)$ o número de reversões de prefixo com sinal aplicadas pelo Algoritmo 18 para ordenar a permutação π, nós temos que $A_{18}(\pi) = 10$. Seja $\gamma = (-5\ +1\ +4\ -3\ +2\ +6\ +7\ \ldots\ +n)$, $n \geq 6$, uma permutação com sinal. Como os elementos γ_i, $6 \leq i \leq n$, estão na posição correta, o Algoritmo 18 irá ordenar γ da mesma forma que ele ordena π. Além disso, nós temos que $d_{rps}(\gamma) = d_{rps}(\gamma) = 5$. Logo, $\frac{A_{18}(\gamma)}{d_{rps}(\gamma)} = \frac{10}{5} = 2$. □

4.4 Problema da Ordenação por Reversões Curtas

Os resultados da auditoria dos algoritmos 20, 23 e 25 são dados pelas tabelas 4.9, 4.10 e 4.11 respectivamente. Como podemos ver na Figura 4.5, o algoritmo 23 apresentou os piores resultados, enquanto que os algoritmos 20 e 25 apresentaram resultados praticamente iguais. Este é um fato interessante pois trata-se de um bom contraexemplo para a ideia de que quanto menor o fator de aproximação, melhor a qualidade das respostas do algoritmo.

A razão máxima obtida para o Algoritmo 20 não se igualou ao fator de aproximação teórico, que é igual a 3. Apesar da razão máxima ter apresentado valores crescentes, eles não são comportados o suficiente para que possamos inferir para que valor estão convergindo. Por outro lado, a razão máxima obtida para o Algoritmo 23 se igualou ao fator de aproximação teórico para $4 \leq n \leq 12$. De fato, o Teorema 9 mostra que o fator de aproximação do Algoritmo 23 é justo.

Tabela 4.9: Resultado da auditoria do Algoritmo 20.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,17	1,00	1,00	100,00%
4	4	2,08	1,00	1,00	100,00%
5	6	3,22	1,01	1,67	97,50%
6	9	4,60	1,03	1,67	92,50%
7	12	6,21	1,05	1,67	84,82%
8	16	8,05	1,07	2,00	75,20%
9	20	10,12	1,08	2,00	63,97%
10	24	12,43	1,09	2,14	52,20%
11	29	14,96	1,11	2,14	40,74%
12	35	17,73	1,12	2,25	30,39%

Tabela 4.10: Resultado da auditoria do Algoritmo 23.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,17	1,00	1,00	100,00%
4	4	2,17	1,04	2,00	95,83%
5	6	3,43	1,09	2,00	86,67%
6	9	5,01	1,13	2,00	72,50%
7	12	6,87	1,16	2,00	55,60%
8	16	9,04	1,20	2,00	38,90%
9	20	11,49	1,23	2,00	24,90%
10	25	14,24	1,25	2,00	14,60%
11	30	17,29	1,28	2,00	7,90%
12	36	20,63	1,30	2,00	3,96%

Tabela 4.11: Resultado da auditoria do Algoritmo 25.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,17	1,00	1,00	100,00%
4	4	2,08	1,00	1,00	100,00%
5	5	3,17	1,00	1,00	100,00%
6	9	4,53	1,01	1,50	96,25%
7	11	6,09	1,03	1,50	90,34%
8	16	7,93	1,05	1,67	80,67%
9	19	9,99	1,07	1,67	68,96%
10	25	12,31	1,08	1,75	55,60%
11	29	14,86	1,10	1,75	42,48%
12	36	17,67	1,11	1,80	30,54%

Teorema 9. *O fator de aproximação do Algoritmo 23 é justo.*

Demonstração. Seja $\pi = (3\ 4\ 1\ 2)$ uma permutação sem sinal. Não é difícil ver que $|V_{rc}(\pi)| = 8$, então $d_{rc}(\pi) \geq \frac{|V_{rc}(\pi)|}{4} \geq 2$ (Lema 11). Como $\pi \circ rc(1,\ 3) \circ rc(2,\ 4) = \iota$, nós podemos concluir que $d_{rc}(\pi) = 2$. Por outro lado, o Algoritmo 23 ordena π da seguinte maneira:

1. O Algoritmo *EncontraOpostos* irá retornar o par $(2,\ 3)$ e o Algoritmo *TrocaComReversoesCurtas* irá permutar os elementos π_2 e π_3 aplicando a reversão curta $rc(2,\ 3)$, produzindo a permutação $\pi = (3\ 1\ 4\ 2)$;

2. O Algoritmo *EncontraOpostos* irá retornar o par $(3,\ 4)$ e o Algoritmo *TrocaComReversoesCurtas* irá permutar os elementos π_3 e π_4 aplicando a reversão curta $rc(3,\ 4)$, produzindo a permutação $\pi = (3\ 1\ 2\ 4)$;

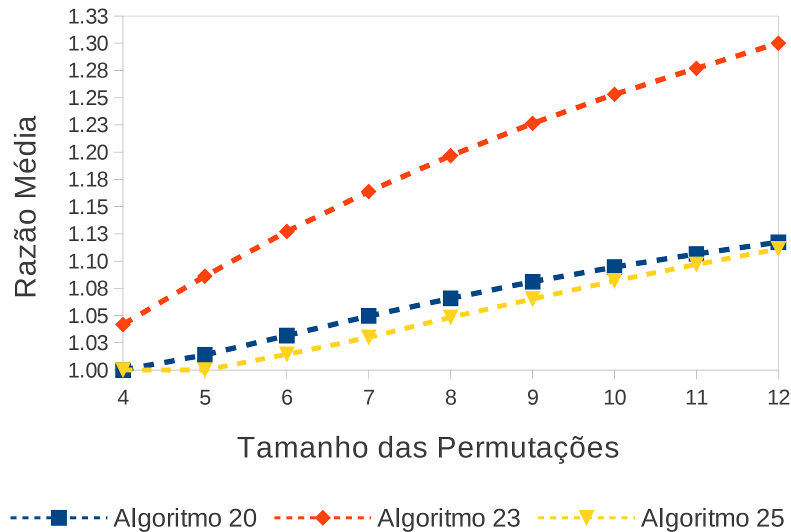
3. O Algoritmo *EncontraOpostos* irá retornar o par $(1,\ 2)$ e o Algoritmo *TrocaComReversoesCurtas* irá permutar os elementos π_1 e π_2 aplicando a reversão curta $rc(1,\ 2)$, produzindo a permutação $\pi = (1\ 3\ 2\ 4)$;

4. Por fim, o Algoritmo *EncontraOpostos* irá retornar o par $(2,\ 3)$ e o Algoritmo *TrocaComReversoesCurtas* irá permutar os elementos π_2 e π_3 aplicando a reversão curta $rc(2,\ 3)$, produzindo a permutação $\pi = (1\ 2\ 3\ 4)$.

Assim, denotando por $A_{23}(\pi)$ o número de reversões curtas aplicadas pelo Algoritmo 23 para ordenar a permutação π, nós temos que $A_{23}(\pi) = 4$. Seja $\sigma \in S_n$ uma permutação tal que

$\sigma_i = \pi_i$ para $1 \leq i \leq 4$ e $\sigma_i = i$ para $5 \leq i \leq n$. Não é difícil notar que $d_{rc}(\sigma) = d_{rc}(\sigma) = 2$ e que o Algoritmo 23 irá ordenar σ da mesma forma que ele ordena π. Logo, $\frac{A_{23}(\sigma)}{d_{rc}(\sigma)} = \frac{4}{2} = 2$. $\square$

A razão máxima obtida para o Algoritmo 25 também não se igualou ao fator de aproximação teórico, que é igual a 2, porém ela parece estar convergindo para isso. De fato, o Teorema 10 mostra que o fator de aproximação do Algoritmo 25 é justo.



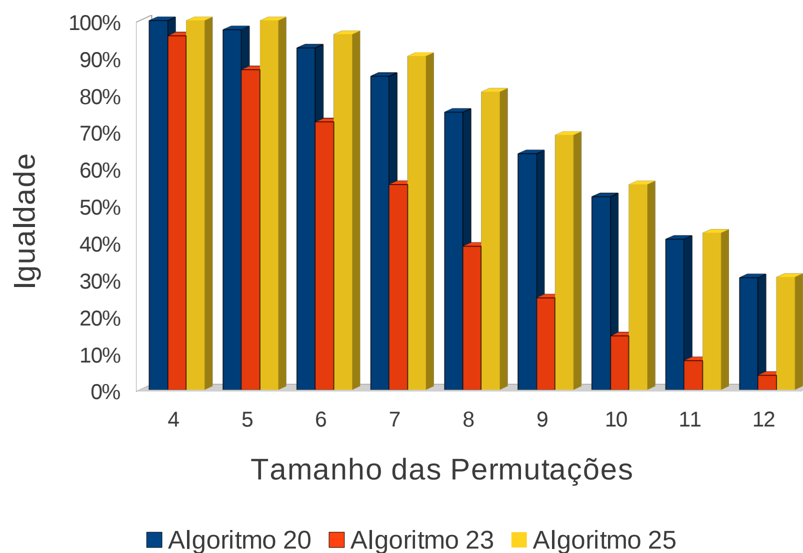


Figura 4.5: Comparação entre os algoritmos 20, 23 e 25 com base nos resultados produzidos pelo GRAAu.

Lema 17. *Seja $\pi \in S_n$, $n \geq 8$ e n par, uma permutação tal que*

$$\pi_i = \begin{cases} n & \textit{se } i = 1 \\ 1 & \textit{se } i = 2 \\ i+2 & \textit{se } i = 3, 5, \ldots, n-5 \\ i-2 & \textit{se } i = 4, 6, \ldots, n-2 \\ n-2 & \textit{se } i = n-3 \\ n-1 & \textit{se } i = n-1 \\ 3 & \textit{se } i = n \end{cases}$$

Então, nós temos que $d_{rc}(\pi) = n - 2$.

Demonstração. Nós iremos mostrar que a sequência de $n-2$ reversões curtas $rc(1, 2)$, $rc(2, 4)$, $rc(4, 6)$, $rc(6, 8)$, ..., $rc(n-2, n)$, $rc(n-3, n-2)$, $rc(n-5, n-3)$, $rc(n-7, n-5)$, $rc(n-9, n-7)$, ..., $rc(3, 5)$ é uma sequência que ordena π otimamente.

Primeiramente, note que o vetor de cada elemento de π é dado por

$$v(\pi_i) = \begin{cases} n-1 & \text{se } i = 1 \\ -1 & \text{se } i = 2 \\ 2 & \text{se } i = 3, 5, \ldots, n-5 \\ -2 & \text{se } i = 4, 6, \ldots, n-2 \\ -1 & \text{se } i = n-3 \\ 0 & \text{se } i = n-1 \\ -(n-3) & \text{se } i = n \end{cases}$$

Logo, nós temos que $|V_{rc}(\pi)| = 4n - 12$. Ademais, seja π^i a permutação produzida após a aplicação das primeiras i reversões curtas daquela sequência, sendo $\pi^0 = \pi$. Então, é possível observar que:

1. Os elementos $\pi_1^0 = n$ e $\pi_2^0 = 1$ são 1-vetorialmente opostos em π^0, portanto a reversão curta $rc(1, 2)$ diminui $|V_{rc}(\pi^0)|$ de 2 unidades;

2. Os elementos $\pi_{2i}^i = n$ e $\pi_{2i+2}^i = 2i$ são 2-vetorialmente opostos em π^i para todo $1 \leq i \leq \frac{n-4}{2}$, portanto a reversão curta $rc(2i, 2i+2)$ diminui $|V_{rc}(\pi^i)|$ de 4 unidades;

3. Os elementos $\pi_{n-2}^{\frac{n-2}{2}} = n$ e $\pi_n^{\frac{n-2}{2}} = 3$ são 2-vetorialmente opostos em $\pi^{\frac{n-2}{2}}$, portanto a reversão curta $rc(n-2, n)$ diminui $|V_{rc}(\pi^{\frac{n-2}{2}})|$ de 4 unidades.

4. Os elementos $\pi_{n-3}^{\frac{n}{2}} = n-2$ e $\pi_{n-2}^{\frac{n}{2}} = 3$ são 1-vetorialmente opostos em $\pi^{\frac{n}{2}}$, portanto a reversão curta $rc(n-3, n-2)$ diminui $|V_{rc}(\pi^{\frac{n}{2}})|$ de 2 unidades;

5. Os elementos $\pi_{n-(2i+3)}^{\frac{n}{2}+i} = n-(2i+1)$ e $\pi_{n-(2i+1)}^{\frac{n}{2}+i} = 3$ são 2-vetorialmente opostos em π^i para todo $1 \leq i \leq \frac{n-6}{2}$, portanto a reversão curta $rc(n-(2i+3), n-(2i+1))$ diminui $|V_{rc}(\pi^{\frac{n}{2}+i})|$ de 4 unidades.

EntÃ£o, a sequência é composta por $n-2$ reversões curtas, das quais 2 diminuem a soma dos tamanhos dos vetores de 2 unidades e $n-4$ diminuem a soma dos tamanhos dos vetores de 4 unidades. Isso implica que, no total, a soma dos tamanhos dos vetores é diminuída de $4n-12$ unidades, portanto a sequência ordena π. Falta mostrar que ela é ótima.

Pelo Lema 11, nós temos que $d_{rc}(\pi) \geq \frac{|V_{rc}(\pi)|}{4} \geq n-3$. Se fosse possível ordenar π aplicando $n-3$ reversões curtas corretivas, então cada uma delas teria que diminuir a soma dos tamanhos dos vetores de 4 unidades. Contudo, não é possível aplicar uma reversão curta corretiva em π que diminua $|V_{rc}(\pi)|$ de 4 unidades pois não existem elementos 2-vetorialmente opostos em π. Logo, $d_{rc}(\pi) > n-3$ e o lema está provado. □

Lema 18. *Seja $A_{25}(\pi)$ o número de reversões curtas aplicadas pelo Algoritmo 25 para ordenar a permutação π do Lema 17. Entao, nós temos que $A_{25}(\pi) = 2n-6$.*

Demonstração. Não é difícil observar que não existem elementos m-vetorialmente opostos em π tais que m é par, portanto os pares de elementos que o Algoritmo *EncontraOpostosGuloso* pode escolher na primeira iteração são (π_1, π_2), (π_3, π_4), (π_5, π_6), . . ., (π_{n-3}, π_{n-2}), todos eles 1-vetorialmente opostos. Como, na nossa implementação, ele sempre seleciona o par mais a direita em caso de empate, podemos concluir que $rc(n-3,\, n-2)$ é a primeira reversão curta aplicada pelo Algoritmo *TrocaComReversoesCurtas*. Seja $\gamma^0 = \pi \circ rc(n-3,\, n-2)$ e seja γ^i a permutação produzida após as próximas i iterações do Algoritmo 25. Podemos provar por indução que

$$\gamma_i^{2k} = \begin{cases} i & \text{se } n-2-2k \leq i \leq n-2 \\ n-4-2k & \text{se } i = n-3-2k \\ \pi_i & \text{caso contrário} \end{cases}$$

para todo $0 \leq k \leq \frac{n-6}{2}$.

Para a base da indução, nós temos que $\gamma^0 = \pi \circ rc(n-3,\, n-2)$. Isso significa que $\gamma_i^0 \neq \pi_i \iff i \in \{n-3, n-2\}$. Além disso, $\gamma_{n-3}^0 = \pi_{n-2} = n-4$ e $\gamma_{n-2}^0 = \pi_{n-3} = n-2$.

Para o passo da indução, vamos assumir que γ^{2k} possui aquela forma para $0 < k < \frac{n-6}{2}$. Nós temos que

$$v(\gamma_i^{2k}) = \begin{cases} 0 & \text{se } n-2-2k \leq i \leq n-2 \\ -1 & \text{se } i = n-3-2k \\ v(\pi_i) & \text{caso contrário} \end{cases}$$

Isso implica que os pares de elementos vetorialmente opostos de γ^{2k} que o Algoritmo *EncontraOpostosGuloso* pode escolher nada mais são do que subconjunto dos pares de elementos 1-vetorialmente opostos que ele poderia escolher na primeira iteração. Isto é, (γ_1^{2k}, γ_2^{2k}), (γ_3^{2k}, γ_4^{2k}), (γ_5^{2k}, γ_6^{2k}), . . ., (γ_{n-5-2k}^{2k}, γ_{n-4-2k}^{2k}). Como o Algoritmo *EncontraOpostosGuloso* escolhe o mais a direita, o Algoritmo *TrocaComReversoesCurtas* aplica a reversão curta $rc(n-5-2k,$ $n-4-2k)$ em γ^{2k}, produzindo a permutação γ^{2k+1}.

Note que $\gamma_i^{2k+1} = \gamma_i^{2k}$ para todo $i \notin \{n-5-2k, n-4-2k\}$, $\gamma_{n-5-2k}^{2k+1} = \gamma_{n-4-2k}^{2k} = \pi_{n-4-2k}$ $= n-6-2k$ e $\gamma_{n-4-2k}^{2k+1} = \gamma_{n-5-2k}^{2k} = \pi_{n-5-2k} = n-3-2k$. Portanto, os elementos γ_{n-4-2k}^{2k+1} $= n-3-2k$ e $\gamma_{n-3-2k}^{2k+1} = n-4-2k$ formam o par de elementos 1-vetorialmente opostos mais a direita de γ^{2k+1}, implicando que $\gamma^{2k+2} = \gamma^{2k+1} \circ rc(n-4-2k, n-3-2k)$. Assim, nós temos

$$\gamma_i^{2(k+1)} = \begin{cases} i & \text{se } n-2-2(k+1) \leq i \leq n-2 \\ n-4-2(k+1) & \text{se } i = n-3-2(k+1) \\ \pi_i & \text{caso contrário.} \end{cases}$$

Quando $k = \frac{n-6}{2}$, não existem elementos m-vetorialmente opostos em $\gamma^{2k} = \gamma^{n-6}$ tais que m é par e existe apenas um par de elementos 1-vetorialmente opostos, o par $(\gamma_1^{n-6}, \gamma_2^{n-6})$. Logo, $\gamma^{n-5} = \gamma^{n-6} \circ rc(1, 2)$. Similarmente, não existem elementos m-vetorialmente opostos em γ^{n-5} tais que m é par e existe apenas um par de elementos 1-vetorialmente opostos, o par $(\gamma_2^{n-5}, \gamma_3^{n-5} = 2)$. Logo, $\gamma^{n-4} = \gamma^{n-5} \circ rc(2, 3)$. Assim, nós temos que

$$\gamma_i^{n-4} = \begin{cases} n & \text{se } i = 3 \\ 3 & \text{se } i = n \\ i & \text{caso contrário.} \end{cases}$$

Como $\gamma_3^{n-4} = n$ e $\gamma_n^{n-4} = 3$ são os únicos elementos vetorialmente opostos de γ^{n-4}, o Algoritmo *TrocaComReversoesCurtas* irá trocar tais elementos de posição aplicando $n-3$ reversões curtas. Como resultado, a permutação identidade será produzida e o Algoritmo 25 termina.

Para transformar π em γ^0, foi aplicada uma reversão curta. Para transformar γ^0 em γ^{n-4}, foram aplicadas $n-4$ reversões curtas. Finalmente, para transformar γ^{n-4} em ι, foram aplicadas $n-3$ reversões curtas. Portanto, o Algoritmo 25 aplicou $2n-6$ reversões curtas para ordenar π. □

Teorema 10. *O fator de aproximação do Algoritmo 25 é justo.*

Demonstração. Seja π a permutação do Lema 17. De acordo com os lemas 17 e 18, nós temos que $\frac{A_{25}(\pi)}{d_{rc}(\pi)} = \frac{2n-6}{n-2}$. Como esta razão converge para 2 no limite, o teorema procede. □

4.5 Problema da Ordenação por Transposições

Os resultados da auditoria dos algoritmos 26, 27 e 28 são dados pelas tabelas 4.9, 4.10 e 4.11 respectivamente. Como podemos ver na Figura 4.6, o algoritmo 28 apresentou resultados muitos melhores do que os outros dois algoritmos.

Antes de analisarmos a razão máxima obtida para o Algoritmo 26, devemos notar que Walter, Dias e Meidanis [63] também realizaram uma auditoria do Algoritmo 26, entretanto os resultados que eles obtiveram não foram iguais aos nossos. Por exemplo, a razão máxima que

Tabela 4.12: Resultado da auditoria do Algoritmo 26.

n	**Diâmetro**	**Distância Média**	**Razão Média**	**Razão Máxima**	**Igualdade**
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,00	1,00	1,00	100,00%
4	3	1,54	1,00	1,00	100,00%
5	3	2,08	1,00	1,00	100,00%
6	4	2,61	1,00	1,33	99,17%
7	5	3,14	1,00	1,33	98,57%
8	6	3,66	1,01	1,50	97,12%
9	6	4,19	1,01	1,50	96,06%
10	7	4,70	1,01	1,50	94,15%
11	8	5,22	1,01	1,60	92,84%
12	9	5,73	1,02	1,60	90,68%
13	9	6,24	1,02	1,60	89,30%

Tabela 4.13: Resultado da auditoria do Algoritmo 27.

n	**Diâmetro**	**Distância Média**	**Razão Média**	**Razão Máxima**	**Igualdade**
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,00	1,00	1,00	100,00%
4	3	1,54	1,00	1,00	100,00%
5	4	2,13	1,02	1,50	95,00%
6	5	2,75	1,06	1,67	85,00%
7	6	3,42	1,10	2,00	71,77%
8	7	4,13	1,14	2,00	56,41%
9	8	4,87	1,18	2,00	41,62%
10	9	5,63	1,22	2,25	28,80%
11	10	6,42	1,25	2,25	18,74%
12	11	7,22	1,29	2,25	11,57%
13	12	8,05	1,32	2,40	6,77%

eles obtiveram para $n = 11$ foi $\frac{10}{5}$, enquanto que a razão máxima que nós obtivemos foi $\frac{8}{5}$. O problema é que a razão $\frac{A_{26}(\pi)}{d_t(\pi)} = \frac{10}{5}$ não está de acordo com o limitante superior que eles demonstraram para $A_{26}(\pi)$. Dada uma permutação $\pi \in S_n$, nós temos que $0 \leq b_t(\pi) \leq n + 1$. Como discutido na Seção 3.5, $A_{26}(\pi) \leq \frac{3}{4} b_t(\pi)$, portanto $A_{26}(\pi) \leq \frac{3n+3}{4}$. Para $n = 11$, nós

Tabela 4.14: Resultado da auditoria do Algoritmo 28.

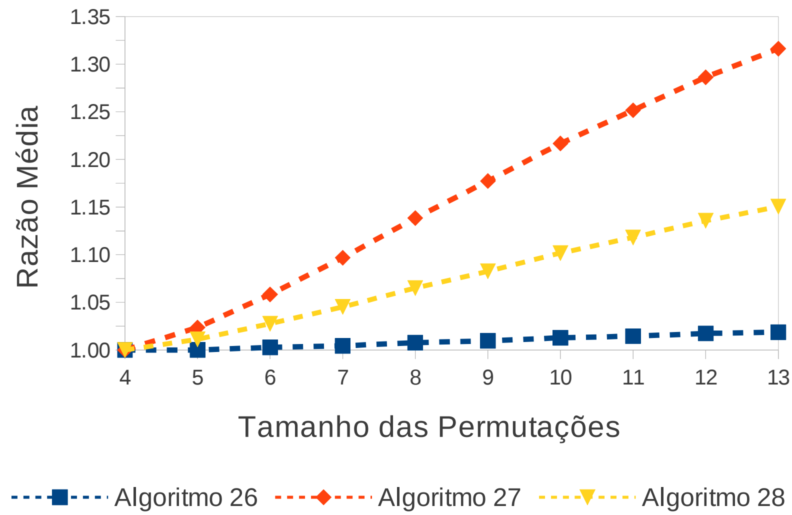
n	**Diâmetro**	**Distância Média**	**Razão Média**	**Razão Máxima**	**Igualdade**
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,00	1,00	1,00	100,00%
4	3	1,54	1,00	1,00	100,00%
5	4	2,10	1,01	1,50	97,50%
6	5	2,67	1,03	1,50	92,78%
7	6	3,26	1,05	1,67	86,45%
8	7	3,86	1,06	1,67	77,93%
9	8	4,48	1,08	2,00	69,06%
10	9	5,10	1,10	2,00	58,94%
11	10	5,73	1,12	2,00	49,61%
12	11	6,38	1,14	2,00	40,23%
13	12	7,03	1,15	2,25	32,18%

temos que $A_{26}(\pi) \leq \frac{36}{4} = 9$. Ou seja, $A_{26}(\pi)$ não poderia ser igual a 10.

A razão máxima obtida para o Algoritmo 26 não se igualou ao fator de aproximação teórico, que é igual a 2.25. Além disso, os valores obtidos parecem estar crescendo em uma progressão que converge para 2, isto é, $\frac{2}{2}$, $\frac{4}{3}$, $\frac{6}{4}$, $\frac{8}{5}$, $\ldots$, $\frac{2k}{k+1}$. Isso pode indicar que talvez o fator de aproximação do Algoritmo 26 não seja justo e pode ser baixado para 2.

A razão máxima obtida para o Algoritmo 27 também não se igualou ao fator de aproximação teórico, que é igual a 3. Em todo caso, se considerarmos as razões máximas obtidas para $n \in \{7, 10, 13\}$, podemos observar que elas seguem a progressão $\frac{6}{3}$, $\frac{9}{4}$, $\frac{12}{5}$, $\ldots$, $\frac{3k}{k+1}$. Nós realizamos alguns experimentos para averiguar a validade dessa hipótese e encontramos permutações π de tamanho $3m + 1$, $m \in \{5, 6, 7\}$, tais que $\frac{p(\pi)}{d_t(\pi)} = \frac{3m}{m+1}$ (essas permutações são dadas na Tabela 4.15).

Apesar de não termos conseguidos determinar uma família de permutações que confirme a convergência da razão máxima, nossos resultados contradizem a hipótese levantada por Benoît-Gagné e Hamel [5] de que o fator de aproximação do Algoritmo 27 “tende a um número significantemente menor do que 3”. Na verdade, nós conjecturamos que o fato de aproximação do Algoritmo 27 é justo.



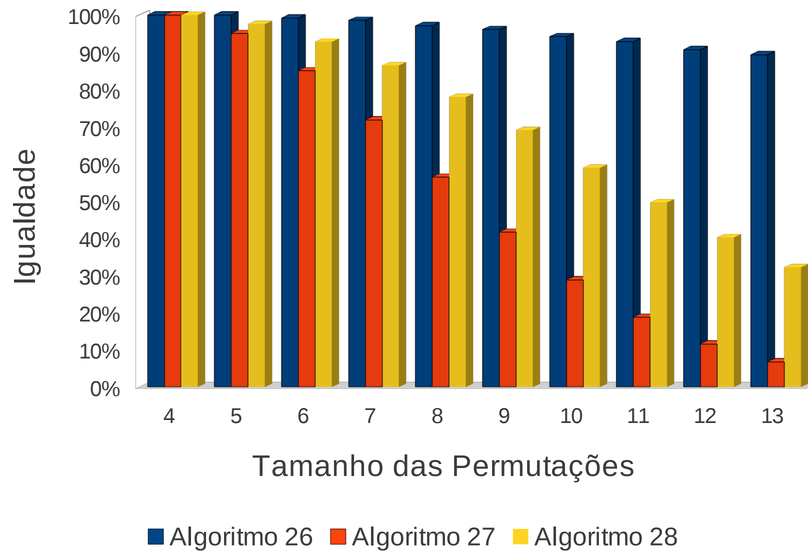


Figura 4.6: Comparação entre os algoritmos 26, 27 e 28 com base nos resultados produzidos pelo GRAAu.

Conjectura 6. *O fator de aproximação do Algoritmo 27 é justo.*

Por fim, a razão máxima obtida para o Algoritmo 28 não se igualou ao fator de aproximação teórico, que é igual a 3. Apesar dessa razão apresentar valores crescentes, eles não são comportados o suficiente para que possamos inferir para qual valor eles estão convergindo.

Tabela 4.15: Permutações π de tamanho $3m + 1$, $m \in \{5, 6, 7\}$, tais que $\frac{p(\pi)}{d_t(\pi)} = \frac{3m}{m+1}$. Note que $d_t(\pi) \geq \frac{b_t(\pi)}{3} \geq m + 1$.

Permutação	**Sequência Ótima de Transposições**
$\pi =$ (16 9 4 11 6 15 8 2 12 7 5 3 14 13 10 1)	t(5, 9, 12), t(1, 7, 10), t(3, 9, 14), t(5, 10, 17), t(6, 10, 14), t(1, 7, 11)
$\pi =$ (19 11 4 18 6 14 8 13 10 2 15 5 9 7 3 17 16 12 1)	t(4, 12, 17), t(7, 12, 16), t(6, 9, 15), t(1, 5, 11), t(3, 8, 20), t(4, 13, 17), t(1, 5, 13)
$\pi =$ (22 13 4 21 6 17 8 16 10 15 12 2 18 5 11 9 7 3 20 19 14 1)	t(4, 14, 20), t(8, 13, 19), t(7, 10, 18), t(6, 9, 17), t(1, 5, 11), t(3, 8, 23), t(4, 16, 20), t(1, 5, 15)

4.6 Problema da Ordenação por Transposições de Prefixo

Os resultados da auditoria dos algoritmos 29 e 30 são dados pelas tabelas 4.16 e 4.17 respectivamente. Como podemos ver na Figura 4.7, o Algoritmo 30 apresentou resultados muito melhores que o Algoritmo 29.

Tabela 4.16: Resultado da auditoria do Algoritmo 29.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,17	1,00	1,00	100,00%
4	3	1,92	1,06	1,50	87,50%
5	4	2,72	1,12	1,50	70,83%
6	5	3,55	1,16	1,67	54,72%
7	6	4,41	1,20	1,67	39,60%
8	7	5,28	1,24	1,75	26,92%
9	8	6,17	1,27	1,75	17,33%
10	9	7,07	1,29	1,80	10,55%
11	10	7,98	1,32	1,80	6,07%
12	11	8,90	1,34	1,83	3,32%
13	12	9,82	1,36	1,83	1,73%

A razão máxima dos algoritmos 29 e 30 não se igualou ao fator de aproximação teórico desses algoritmos, que é igual a 2. Apesar disso, podemos notar que os valores parecem estar convergindo para 2. Antes de demonstrar que isso ocorre de fato, precisamos introduzir alguns conceitos.

Tabela 4.17: Resultado da auditoria do Algoritmo 30.

n	Diâmetro	Distância Média	Razão Média	Razão Máxima	Igualdade
1	0	0,00	1,00	1,00	100,00%
2	1	0,50	1,00	1,00	100,00%
3	2	1,17	1,00	1,00	100,00%
4	3	1,79	1,00	1,00	100,00%
5	4	2,45	1,01	1,33	97,50%
6	5	3,10	1,01	1,33	95,28%
7	6	3,77	1,02	1,50	91,11%
8	7	4,43	1,03	1,50	86,61%
9	8	5,10	1,04	1,60	81,31%
10	9	5,77	1,05	1,60	75,55%
11	10	6,44	1,06	1,67	69,64%
12	11	7,12	1,07	1,67	63,56%
13	12	7,79	1,07	1,71	57,58%

Nós dizemos que uma transposição de prefixo $t(i, j)$ não corta uma *strip* de uma permutação sem sinal π se os pares de elementos adjacentes (π_{i-1}, π_i) e (π_{j-1}, π_j) são *breakpoints*. Além disso, nós dizemos que uma permutação sem sinal é reduzida se ela contém somente *strips* de tamanho 1.

Dado que uma *strip* de tamanho maior do que 1 pode ser reduzida a uma *strip* de tamanho 1, podemos reduzir uma permutação sem sinal que contém uma ou mais *strips* de tamanho maior do que 1 reduzindo sucessivamente tais *strips*. Por exemplo, seja $\pi_i, \ldots, \pi_j$, $i < j$, uma *strip* de uma permutação sem sinal π. Nós reduzimos essa *strip* da seguinte maneira: remova todos os elementos π_k tal que $i + 1 \leq k \leq j$ e troque todos os elementos π_l tal que $\pi_l > \pi_j$ por $\pi_i + (\pi_l - \pi_j)$.

Exemplo 21. *Seja* $(3\ 4\ 5\ 1\ 2\ 6)$ *uma permutação sem sinal. Para reduzirmos a strip* $3\ 4\ 5$*, removemos os elementos* 4 *e* 5 *e trocamos o elemento* 6 *por* 4*, obtendo como resultado a permutação* $(3\ 1\ 2\ 4)$*. Depois, para reduzimos a strip* $1\ 2$*, removemos o elemento* 2 *e trocamos os elementos* 3 *e* 4 *por* 2 *e* 3 *respectivamente, obtendo como resultado a permutação* $(2\ 1\ 3)$*, que é uma permutação reduzida.*

Lema 19. *Se uma permutação sem sinal* π *é reduzida a uma permutação sem sinal* σ*, então* $d_{tp}(\pi) = d_{tp}(\sigma)$.

Demonstração. Este lema segue diretamente do fato de que, como provado por Dias e Meidanis [16], qualquer permutação sem sinal pode ser ordenada otimamente por meio da aplicação de transposições de prefixo que não cortam *strips*. □

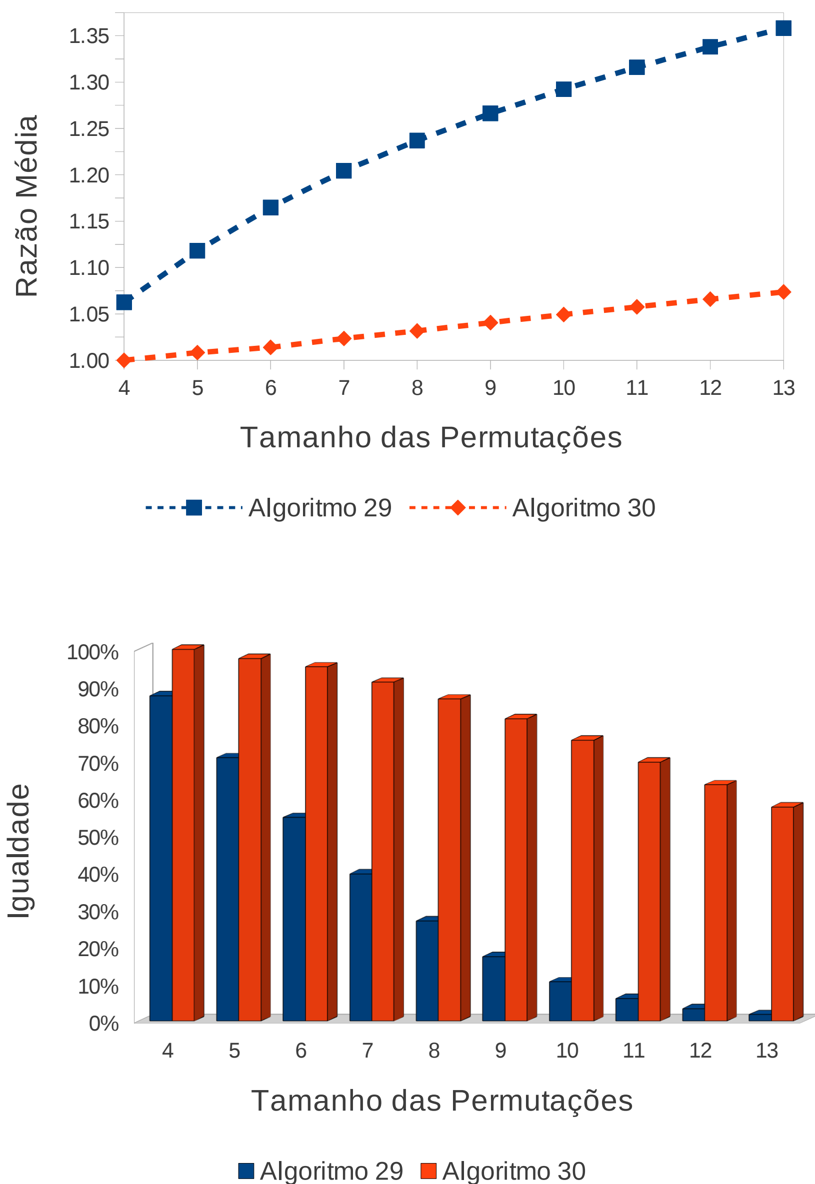


Figura 4.7: Comparação entre os algoritmos 29 e 30 com base nos resultados produzidos pelo GRAAu.

Lema 20. *Sejam $A_{29}(\pi)$ e $A_{30}(\pi)$ o número de transposições de prefixos aplicadas, respectivamente, pelos algoritmos 29 e 30 para ordenar uma permutação sem sinal π. Se π for reduzida a uma permutação sem sinal σ, então $A_{29}(\pi) = A_{29}(\sigma)$ e $A_{30}(\pi) = A_{30}(\sigma)$.*

Demonstração. Este lema segue diretamente do fato de que os algoritmos 29 e 30 nunca aplicam transposições de prefixo que cortam *strips*. □

Lema 21. *Seja H^n uma permutação sem sinal de tamanho n, sendo n um número inteiro par, tal que*

$$H_i^n = \begin{cases} n - \frac{i-1}{2} & \textit{se } i \in \{1, 3, 5, \ldots, n-1\} \\ \frac{i}{2} & \textit{se } i \in \{2, 4, 6, \ldots, n\}. \end{cases}$$

Então, nós temos que $d_{tp}(H^n) = \frac{n}{2}$.

Demonstração. A prova segue por indução em n. Para o caso base, nós temos que $d_{tp}(H^2)$ $= 1$. Para o passo da indução, vamos assumir que $d_{tp}(H^k) = \frac{k}{2}$ para $2 \leq k \leq n$. Aplicando a transposição de prefixo $tp(3, 4)$ em H^{n+2}, produzimos a permutação sem sinal $\pi = H^{n+2} \circ tp(3, 4)$ tal que

$$\pi_i = \begin{cases} n + i & \text{se } i \in \{1, 2\} \\ i - 2 & \text{se } i \in \{3, 4\} \\ n - \frac{i-1}{2} & \text{se } i \in \{5, 7, 9, \ldots, n+1\} \\ \frac{i}{2} & \text{se } i \in \{6, 8, 10, \ldots, n+2\}. \end{cases}$$

Reduzindo as *strips* π_1 π_2 e π_3 π_4 de π, obtemos a permutação H^n. Logo, $d_{tp}(H^{n+2}) \leq 1$ $+ d_{tp}(H^n) = \frac{n+2}{2}$. Como $d_{tp}(H^{n+2}) \geq \frac{b_{tp}(H^{n+2})}{2} = \frac{n+2}{2}$, podemos concluir que $d_{tp}(H^{n+2}) = \frac{n+2}{2}$. □

Lema 22. $A_{29}(H^n) = n - 1$.

Demonstração. A prova segue por indução em n. Para o caso base, nós temos que $A_{29}(H^2)$ $= 1$. Para o passo da indução, vamos assumir que $A_{29}(H^k) = k - 1$ para $2 \leq k \leq n$. Caso H^{n+2} seja dada como entrada para o Algoritmo 29, nós temos que ele aplicará a transposição de prefixo $tp(2, n+3)$ seguida da transposição de prefixo $tp(2, 3)$, produzindo a permutação π $= H^{n+2} \circ tp(2, n+3) \circ tp(2, 3)$ tal que

$$\pi_i = \begin{cases} n + 1 & \text{se } i = 1 \\ i - 1 & \text{se } i \in \{2, 3\} \\ n - \frac{i}{2} & \text{se } i \in \{4, 6, 8, \ldots, n\} \\ \frac{i+1}{2} & \text{se } i \in \{5, 7, 9, \ldots, n+1\} \\ n + 2 & \text{se } i = n + 2. \end{cases}$$

Reduzindo a *strip* π_2 π_3 de π, obtemos a permutação reduzida σ tal que

$$\sigma_i = \begin{cases} n - \frac{i-1}{2} & \text{se } i \in \{1, 3, 5, \ldots, n-1\} \\ \frac{i}{2} & \text{se } i \in \{2, 4, 6, \ldots, n\} \\ n + 1 & \text{se } i = n + 1. \end{cases}$$

Como $\sigma_i = H_i^n$ para todo $1 \leq i \leq n$ e $\sigma_{n+1} = n + 1$, não é difícil notar que $A_{29}(\sigma) = A_{29}(H^n)$. Logo, $A_{29}(H^{n+2}) = A_{29}(H^n) + 2 = n + 1$. □

Teorema 11. *O fator de aproximação do Algoritmo 29 é justo.*

Demonstração. De acordo com os lemas 21 e 22, nós temos que $\frac{A_{29}(H^n)}{d_{tp}(H^n)} = \frac{2n-2}{n}$. Como esta razão converge para 2 no limite, o teorema procede. □

Lema 23. *Seja J^n uma permutação sem sinal de tamanho n, sendo n um número inteiro par, tal que*

$$J_i^n = \begin{cases} 2i & \textit{se } i \in \{1, 2, 3, \ldots, \frac{n}{2}\} \\ 2(i - \frac{n}{2}) - 1 & \textit{se } i \in \{\frac{n}{2} + 1, \frac{n}{2} + 2, \frac{n}{2} + 3, \ldots, n\}. \end{cases}$$

Então, nós temos que $d_{tp}(J^n) = \frac{n}{2}$.

Demonstração. A prova segue por indução em n. Para o caso base, nós temos que $d_{tp}(J^2) = 1$. Para o passo da indução, vamos assumir que $d_{tp}(J^k) = \frac{k}{2}$ para $2 \leq k \leq n$. Aplicando a transposição de prefixo $tp(2, \frac{n+2}{2} + 2)$ em J^{n+2}, produzimos a permutação sem sinal $\pi = J^{n+2} \circ tp(2, \frac{n+2}{2} + 2)$ tal que

$$\pi_i = \begin{cases} 2(i+1) & \text{se } i \in \{1, 2, 3, \ldots, \frac{n}{2}\} \\ 1 & \text{se } i = \frac{n+2}{2} \\ 2 & \text{se } i = \frac{n+2}{2} + 1 \\ 2(i - \frac{n+2}{2}) - 1 & \text{se } i \in \{\frac{n+2}{2} + 2, \frac{n}{2} + 3, \frac{n}{2} + 4, \ldots, n+2\}. \end{cases}$$

Reduzindo a *strip* $\pi_{\frac{n+2}{2}}\ \pi_{\frac{n+2}{2}+1}\ \pi_{\frac{n+2}{2}+2}$ de π, obtemos a permutação J^n. Logo, $d_{tp}(J^{n+2}) \leq 1 + d_{tp}(J^n) = \frac{n+2}{2}$. Como $d_{tp}(J^{n+2}) \geq \frac{b_{tp}(J^{n+2})}{2} = \frac{n+2}{2}$, podemos concluir que $d_{tp}(J^{n+2}) = \frac{n+2}{2}$. □

Lema 24. *Seja K^n uma permutação sem sinal de tamanho n, sendo n um número inteiro par, tal que*

$$K_i^n = \begin{cases} \frac{n}{2} + \frac{i+1}{2} & \textit{se } i \in \{1, 3, 5, \ldots, n-1\} \\ \frac{i}{2} & \textit{se } i \in \{2, 4, 6, \ldots, n\}. \end{cases}$$

Então, nós temos que $d_{tp}(K^n) = \frac{n}{2}$.

Demonstração. A prova segue por indução em n. Para o caso base, nós temos que $d_{tp}(K^2) = 1$. Para o passo da indução, vamos assumir que $d_{tp}(K^k) = \frac{k}{2}$ para $2 \leq k \leq n$. Aplicando a transposição de prefixo $tp(n+2, n+3)$ em K^{n+2}, produzimos a permutação sem sinal $\pi = K^{n+2} \circ tp(n+2, n+3)$ tal que

$$\pi_i = \begin{cases} \frac{n+2}{2} & \text{se } i = 1 \\ \frac{n+2}{2} + \frac{i}{2} & \text{se } i \in \{2, 4, 6, \ldots, n+2\} \\ \frac{i-1}{2} & \text{se } i \in \{3, 5, 7, \ldots, n+1\}. \end{cases}$$

Reduzindo a *strip* $\pi_1\ \pi_2$ de π, obtemos a permutação reduzida σ tal que

$$\sigma_i = \begin{cases} \frac{n}{2} + \frac{i+1}{2} & \text{se } i \in \{1, 3, 5, \ldots, n-1\} \\ \frac{i}{2} & \text{se } i \in \{2, 4, 6, \ldots, n\} \\ n+1 & \text{se } i = n+1. \end{cases}$$

Como $\sigma_i = K_i^n$ para todo $1 \leq i \leq n$ e $\sigma_{n+1} = n+1$, não é difícil notar que $d_{tp}(\sigma) = d_{tp}(K^n)$. Logo, $d_{tp}(K^{n+2}) \leq 1 + d_{tp}(K^n) = \frac{n+2}{2}$. Como $d_{tp}(K^{n+2}) \geq \frac{b_{tp}(K^{n+2})}{2} = \frac{n+2}{2}$, podemos concluir que $d_{tp}(K^{n+2}) = \frac{n+2}{2}$. □

Lema 25. *Seja L^n uma permutação sem sinal de tamanho n, sendo n par e $n \geq 4$, tal que*

$$L_i^n = \begin{cases} n & \textit{se } i = 1 \\ 1 & \textit{se } i = 2 \\ i & \textit{se } i \in \{3, 5, 7, \ldots, n-1\} \\ i-2 & \textit{se } i \in \{4, 6, 8, \ldots, n\}. \end{cases}$$

Então, nós temos que $d_{tp}(L^n) = \frac{n}{2}$.

Demonstração. Primeiramente, note que $d_{tp}(L^n) \geq \frac{b_{tp}(L^n)}{2} = \frac{n}{2}$. Para mostrar que esse limite inferior é justo, dividimos a nossa análise em dois casos:

- Caso 1: $n \equiv 0 \pmod 4$.

  Nesse caso, nós dividimos o processo de ordenação de L^n em duas fases. Primeiramente, transformamos L^n na permutação M^n tal que

$$M_i^n = \begin{cases} 2i+1 & \text{se } i \in \{1, 3, 5, \ldots, \frac{n}{2} - 1\} \\ 2i & \text{se } i \in \{2, 4, 6, \ldots, \frac{n}{2}\} \\ 2(i - \frac{n}{2}) - 1 & \text{se } i \in \{\frac{n}{2}+1, \frac{n}{2}+3, \frac{n}{2}+5, \ldots, n-1\} \\ 2(i - \frac{n}{2}) - 2 & \text{se } i \in \{\frac{n}{2}+2, \frac{n}{2}+4, \frac{n}{2}+6, \ldots, n\}. \end{cases}$$

  Depois, transformamos M^n na permutação identidade. Como transformar L^n em M^n é equivalente a transformar $M^{n^{-1}} \circ L_n$ na permutação identidade, nós podemos concluir que $d_{tp}(L^n) \leq d_{tp}(M^{n^{-1}} \circ L^n) + d_{tp}(M^n)$.

  Nós temos que

$$M_i^{n^{-1}} = \begin{cases} \frac{n}{2} + \frac{i+1}{2} & \text{se } i \in \{1, 5, 9, \ldots, n-3\} \\ \frac{n}{2} + \frac{i+2}{2} & \text{se } i \in \{2, 6, 10, \ldots, n-2\} \\ \frac{i-1}{2} & \text{se } i \in \{3, 7, 11, \ldots, n-1\} \\ \frac{i}{2} & \text{se } i \in \{4, 8, 12, \ldots, n\}, \end{cases}$$

  portanto a permutação $\pi = M^{n^{-1}} \circ L^n$ é tal que

$$\pi_i = \begin{cases} \frac{n}{2} & \text{se } i = 1 \\ \frac{n}{2}+1 & \text{se } i = 2 \\ \frac{i-1}{2} & \text{se } i \in \{3,7,11,\ldots,n-1\} \\ \frac{n}{2}+\frac{i}{2} & \text{se } i \in \{4,8,12,\ldots,n\} \\ \frac{n}{2}+\frac{i+1}{2} & \text{se } i \in \{5,9,13,\ldots,n-3\} \\ \frac{i-2}{2} & \text{se } i \in \{6,10,14,\ldots,n-2\}. \end{cases}$$

Reduzindo as *strips* $\pi_1\ \pi_2$ e $\pi_{2i}\ \pi_{2i+1}$ para $2 \leq i \leq \frac{n-2}{2}$ de π, obtemos a permutação reduzida σ tal que

$$\sigma_i = \begin{cases} \frac{n}{4}+\frac{i+1}{2} & \text{se } i \in \{1,3,5,\ldots,\frac{n}{2}-1\} \\ \frac{i}{2} & \text{se } i \in \{2,4,6,\ldots,\frac{n}{2}\} \\ \frac{n}{2}+1 & \text{se } i = \frac{n}{2}+1. \end{cases}$$

Dado que $\sigma_i = K_i^{\frac{n}{2}}$ para todo $1 \leq i \leq \frac{n}{2}$ e $\sigma_{\frac{n}{2}+1} = \frac{n}{2}+1$, não é difícil notar que $d_{tp}(\sigma) = d_{tp}(K^{\frac{n}{2}})$. Além disso, também não é difícil notar que a permutação M^n pode ser reduzida à permutação $J^{\frac{n}{2}}$. Logo, $d_{tp}(L^n) \leq d_{tp}(M^{n^{-1}} \circ L^n) + d_{tp}(M^n) = d_{tp}(K^{\frac{n}{2}}) + d_{tp}(J^{\frac{n}{2}}) = \frac{n}{2}$.

- Caso 2: $n \equiv 2 \pmod 4$.

  Nesse caso, também dividimos o processo de ordenação de L^n em duas fases. Primeiramente, transformamos L^n na permutação O^n tal que

$$O_i^n = \begin{cases} 2 & \text{se } i = 1 \\ 2i+1 & \text{se } i \in \{2,4,6,\ldots,\frac{n}{2}-1\} \\ 2i & \text{se } i \in \{3,5,7,\ldots,\frac{n}{2}\} \\ 1 & \text{se } i = \frac{n}{2}+1 \\ 2(i-\frac{n}{2})-1 & \text{se } i \in \{\frac{n}{2}+2,\frac{n}{2}+4,\frac{n}{2}+6,\ldots,n-1\} \\ 2(i-\frac{n}{2})-2 & \text{se } i \in \{\frac{n}{2}+3,\frac{n}{2}+5,\frac{n}{2}+7,\ldots,n\}. \end{cases}$$

  Depois, transformamos O^n na permutação identidade. Como transformar L^n em O^n é equivalente a transformar $O^{n^{-1}} \circ L_n$ na permutação identidade, nós podemos concluir que $d_{tp}(L^n) \leq d_{tp}(O^{n^{-1}} \circ L^n) + d_{tp}(O^n)$.

  Nós temos que

$$O_i^{n^{-1}} = \begin{cases} \frac{n}{2}+1 & \text{se } i = 1 \\ 1 & \text{se } i = 2 \\ \frac{n}{2}+\frac{i+1}{2} & \text{se } i \in \{3,7,11,\ldots,n-3\} \\ \frac{n}{2}+\frac{i+2}{2} & \text{se } i \in \{4,8,12,\ldots,n-2\} \\ \frac{i-1}{2} & \text{se } i \in \{5,9,13,\ldots,n-1\} \\ \frac{i}{2} & \text{se } i \in \{6,10,14,\ldots,n\}, \end{cases}$$

portanto a permutação $\pi = O^{n^{-1}} \circ L^n$ é tal que

$$\pi_i = \begin{cases} \frac{n}{2} & \text{se } i = 1 \\ \frac{n}{2} + 1 & \text{se } i = 2 \\ \frac{n}{2} + \frac{i+1}{2} & \text{se } i \in \{3, 7, 11, \ldots, n-3\} \\ \frac{i-2}{2} & \text{se } i \in \{4, 8, 12, \ldots, n-2\} \\ \frac{i-1}{2} & \text{se } i \in \{5, 9, 13, \ldots, n-1\} \\ \frac{n}{2} + \frac{i}{2} & \text{se } i \in \{6, 10, 14, \ldots, n\}. \end{cases}$$

Ao reduzirmos as *strips* $\pi_1\ \pi_2\ \pi_3$ e $\pi_{2i}\ \pi_{2i+1}$, $2 \leq i \leq \frac{n-2}{2}$, de π, obtemos a permutação reduzida σ tal que

$$\sigma_i = \begin{cases} \frac{n-2}{4} + \frac{i+1}{2} & \text{se } i \in \{1, 3, 5, \ldots, \frac{n-2}{2} - 1\} \\ \frac{i}{2} & \text{se } i \in \{2, 4, 6, \ldots, \frac{n-2}{2}\} \\ \frac{n}{2} & \text{se } i = \frac{n}{2}. \end{cases}$$

Dado que $\sigma_i = K_i^{\frac{n-2}{2}}$ para todo $1 \leq i \leq \frac{n-2}{2}$ e $\sigma_{\frac{n}{2}} = \frac{n}{2}$, não é difícil notar que $d_{tp}(\sigma) = d_{tp}(K^{\frac{n-2}{2}})$. Além disso, também não é difícil notar que a permutação O^n pode ser reduzida à permutação $J^{\frac{n+2}{2}}$. Logo, $d_{tp}(L^n) \leq d_{tp}(O^{n^{-1}} \circ L^n) + d_{tp}(O^n) = d_{tp}(K^{\frac{n-2}{2}}) + d_{tp}(J^{\frac{n+2}{2}}) = \frac{n}{2}$.

Como $\frac{n}{2} \leq d_{tp}(L^n) \leq \frac{n}{2}$, nós podemos concluir que $d_{tp}(L^n) = \frac{n}{2}$. □

Lema 26. *Seja P^n uma permutação sem sinal de tamanho n, sendo n par e $n \geq 6$, tal que*

$$P_i^n = \begin{cases} 1 & \textit{se } i = 1 \\ i+1 & \textit{se } i \in \{2, 4, 6, \ldots, n-2\} \\ i-1 & \textit{se } i \in \{3, 5, 7, \ldots, n-1\} \\ n & \textit{se } i = n. \end{cases}$$

Então, nós temos que $d_{tp}(P^n) = \frac{n}{2}$.

Demonstração. Aplicando a transposição de prefixo $tp(n-1, n)$ em P^n, obtemos a permutação $\pi = P^n \circ tp(n-1, n)$ tal que

$$\pi_i = \begin{cases} n-2 & \text{se } i = 1 \\ 1 & \text{se } i = 2 \\ i & \text{se } i \in \{3, 5, 7, \ldots, n-3\} \\ i-2 & \text{se } i \in \{4, 6, 8, \ldots, n-2\} \\ n-1 & \text{se } i = n-1 \\ n & \text{se } i = n. \end{cases}$$

Dado que $\pi_i = L_i^{n-2}$ para todo $1 \leq i \leq n-2$ e $\pi_i = i$ para $n-1 \leq i \leq n$, não é difícil notar que $d_{tp}(\pi) = d_{tp}(L^{n-2})$. Logo, $d_{tp}(P^n) \leq 1 + d_{tp}(L^{n-2}) = \frac{n}{2}$. Como $d_{tp}(P^n) \geq \frac{b_{tp}(P^n)}{2} = \frac{n-1}{2}$, podemos concluir que $d_{tp}(P^n) = \frac{n}{2}$. □

Lema 27. $A_{30}(P^n) = n - 2$.

Demonstração. A prova segue por indução em n. Para o caso base, é fácil verificar que $A_{30}(P^6)$ $= 4$. Para o passo da indução, vamos assumir que $A_{30}(P^k) = k - 2$ para $6 \leq k \leq n$. Caso a permutação P^{n+2} seja dada como entrada para o Algoritmo 30, nós temos que ele aplicará a transposição de prefixo $tp(2,\ 3)$ seguida da transposição de prefixo $tp(2,\ 5)$, produzindo a permutação $\pi = P^{n+2} \circ tp(2,\ 3) \circ tp(2,\ 5)$ tal que

$$\pi_i = \begin{cases} 1 & \text{se } i = 1 \\ 2 & \text{se } i = 2 \\ 5 & \text{se } i = 3 \\ 3 & \text{se } i = 4 \\ 4 & \text{se } i = 5 \\ i+1 & \text{se } i \in \{4, 6, 8, \ldots, n\} \\ i-1 & \text{se } i \in \{5, 7, 9, \ldots, n+1\} \\ n+2 & \text{se } i = n+2. \end{cases}$$

Reduzindo as *strips* $\pi_1\ \pi_2$ e $\pi_4\ \pi_5$ de π, obtemos a permutação P^n. Logo, $A_{30}(P^{n+2}) = 2 + A_{30}(P^n) = n$. □

Teorema 12. *O fator de aproximação do Algoritmo 30 é justo.*

Demonstração. De acordo com os lemas 26 e 27, nós temos que $\frac{A_{30}(P^n)}{d_{tp}(P^n)} = \frac{2n-4}{n}$. Como esta razão converge para 2 no limite, o teorema procede. □

Capítulo 5

Considerações Finais

Nesta dissertação, apresentamos uma ferramenta de auditoria, chamada GRAAu, que serve para avaliar as respostas de algoritmos de rearranjo de genomas aproximados ou heurísticos. Tal avaliação é importante tanto para efeitos de comparação quanto para efeitos de validação de tais algoritmos.

Para construir a ferramenta, calculamos a distância de rearranjo de todas as permutações de S_n, $1 \leq n \leq 13$, e de $S_n^{\pm}$, $1 \leq n \leq 10$, com respeito a diversos modelos de rearranjo abordados na literatura que consideram reversões ou transposições. No melhor do nosso conhecimento, essa foi a primeira vez que um cálculo desse tipo foi realizado para tais valores de n. Esse feito se deve, em grande parte, ao fato de termos desenvolvido um algoritmo de busca em largura simples e flexível que é mais eficiente em termos de uso de memória do que qualquer outro algoritmo que encontramos na literatura. Além disso, para melhorar o tempo de execução, que é exponencial no tamanho das permutações, criamos uma maneira de paralelizá-lo.

Analisando a distribuição das distâncias de rearranjo, pudemos observar alguns fatos interessantes. Em primeiro lugar, procuramos por outros trabalhos que também apresentassem as distribuições das distâncias de rearranjo para que pudéssemos confrontar com aquelas calculadas por nós e acabamos notando que a distribuição da distância de reversão apresentada por Kececioglu e Sankoff [43] não está correta. Depois, procuramos por conjecturas a respeito do diâmetro do grupo simétrico (com ou sem sinal) que pudessem ser validadas a partir das distribuições que calculamos. Como resultado, verificamos que a conjectura de Dias e Meidanis [16] a respeito do diâmetro de transposição de prefixo é válida para $n = 12$ e $n = 13$ e que a conjectura de Walter, Dias e Meidanis [62] a respeito do diâmetro de reversão com sinal e transposição não é válida para $n = 7$ e $n = 9$. Em razão disso, apresentamos uma nova conjectura.

Em uma tentativa de melhor capturar a maneira como as distâncias de rearranjo se distribuem em S_n e em $S_n^{\pm}$, propusemos duas medidas, o diâmetro transversal e a longevidade, e apresentamos algumas conjecturas a respeito delas.

A fim de que pudéssemos ilustrar as aplicações do GRAAu, implementamos e auditamos 16 algoritmos de rearranjo de genomas aproximados relativos a 6 variações do Problema da Ordenação por Rearranjo: Problema da Ordenação por Reversões, Problema da Ordenação por Reversões de Prefixo, Problema da Ordenação por Reversões de Prefixo com Sinal, Problema da Ordenação por Reversões Curtas, Problema da Ordenação por Transposições e o Problema da Ordenação por Transposições de Prefixo. Dos 16 algoritmos, 4 deles foram propostos por nós: dois para o Problema da Ordenação por Reversões de Prefixo com Sinal, um para o Problema da Ordenação por Transposições e um para o Problema da Ordenação por Transposições de Prefixo.

A implementação dos algoritmos dependeu de uma revisão aprofundada dos conceitos relativos a cada um deles. Tal revisão acabou resultando em algumas contribuições, como:

- uma demonstração do fator de aproximação do algoritmo $\frac{n-1}{2}$-aproximado desenvolvido por Watterson e colegas [65] para o Problema da Ordenação por Reversões;
- uma descrição formal do algoritmo 3-aproximado proposto por Fischer e Ginzinger [24] para o Problema da Ordenação por Reversões de Prefixo, incluindo uma demonstração do seu fator de aproximação;
- uma demonstração do fator de aproximação do algoritmo 2-aproximado desenvolvido por Cohen e Blum [13] para o Problema da Ordenação por Reversões de Prefixo com Sinal;
- a complementação da demonstração do fator de aproximação do algoritmo 3-aproximado desenvolvido por Benoît-Gagné e Hamel [5] para o Problema da Ordenação por Transposições;
- e a demonstração de um fator de aproximação igual a 3 para uma versão restrita da heurística proposta por Guyer, Heath e Vergara [37] para o Problema da Ordenação por Transposições.

Uma aplicação para as estatísticas produzidas pelo GRAAu é a comparação. Apesar de, na maioria das comparações realizadas, termos obtido resultados razoavelmente claros quanto à superioridade de um algoritmo em relação aos demais, houveram casos em que tal superioridade não pode ser claramente constatada. Ainda que essa não constatação possa ser fruto de uma equivalência entre os algoritmos auditados, nós acreditamos que ela alerta as limitações de uma comparação realizada com base apenas em permutações de tamanhos pequenos. Em razão disso, pretendemos estender o GRAAu de maneira a fazer com que ele também teste os algoritmos utilizando permutações aleatórias de tamanhos arbitrariamente grandes.

Outra aplicação das estatísticas produzidas pelo GRAAu é a validação. Quanto a isso, discutimos principalmente a questão da justeza do fator de aproximação, que é um aspecto praticamente não explorado pela literatura de Rearranjo de Genomas. Baseado nas informações

da Razão Máxima e das permutações que exibiram tal razão, demonstramos que o fator de aproximação de 7 dos 16 algoritmos aproximados considerados é justo. Ademais, conjecturamos que o fator de aproximação de outros 6 algoritmos também é justo e levantamos a hipótese de que o fator de aproximação de um outro algoritmo pode ser diminuído. Assim, dos 16 algoritmos considerados, conseguimos demonstrar ou inferir algum resultado quanto à justeza do fator de aproximação para 14 algoritmos, o que evidencia a eficácia da ferramenta nesse sentido.

Os resultados referentes à justeza do fator de aproximação contrapõem duas hipóteses encontradas na literatura. Uma das hipóteses, levantada por Benoît-Gagné e Hamel [5], é de que o fator de aproximação do algoritmo 3-aproximado desenvolvido por eles para o Problema da Ordenação por Transposições "tende a um número significantemente menor do que 3". Nossos resultados apontaram para uma direção contrária, indicando que a Razão Máxima computada para tal algoritmo pode ser tão próxima de 3 quanto quisermos.

Outra hipótese, levantada por Fischer e Ginzinger [24], é a de que o fator de aproximação do algoritmo 2-aproximado desenvolvido por eles para o Problema da Ordenação por Reversões de Prefixo pode ser diminuído. Apesar deles não terem especificado um algoritmo, nós implementamos dois algoritmos que utilizam a estratégia gulosa proposta por eles e mostramos que o fator de aproximação de um deles é justo e que o fator de aproximação do outro dá claras evidências de que também é justo. Isso significa que, no pior caso, tal estratégia não é capaz de produzir um algoritmo aproximado com um fator de aproximação menor do que 2.

Mostramos que os resultados experimentais apresentados por Walter, Dias e Meidanis [63] a respeito do algoritmo 2.25-aproximado para o Problema da Ordenação por Transposições não são corretos. Apesar dessa conclusão não ter sido derivada das estatísticas produzidas pelo GRAAu, ela poderia ter sido, isto é, se tivéssemos auditado a implementação deles com o GRAAu, certamente obteríamos valores para o Diâmetro que não estariam de acordo com o limite superior derivado na teoria e, consequentemente, detectaríamos o erro.

Na verdade, analisar se os valores do Diâmetro estão de acordo com os limites derivados da teoria foi um dos métodos que utilizamos para verificar se nossas implementações não estavam incorretas. Outro método utilizado foi averiguar se a Razão Máxima não superou o fator de aproximação demonstrado para o algoritmo. Note que este método pode ser utilizado inclusive para verificar a corretude da implemementação de um algoritmo exato, cujo "fator de aproximação" é igual a 1.

Finalmente, gostaríamos de salientar que o GRAAu não precisa ser utilizado exclusivamente para avaliar algoritmos de rearranjo de genomas. Por exemplo, Labarre [46] demonstrou um limite inferior para a distância de transposição de prefixo e, para compará-lo com os limites inferiores demonstrados por Dias e Meidanis [16] e Chitturi e Sudborough [11], ele teve que computar a distância de rearranjo de todas as permutações de S_n para $1 \leq n \leq 12$. Apesar de ele ter utilizado a nossa implementação de 32 bits (apresentada na Seção 2.1.3) a fim de conseguir

realizar os cálculos para $n = 11$ e $n = 12$, ele poderia ter utilizado o GRAAu para realizar a comparação. Com isso, ele se beneficiaria em dois aspectos: ele não gastaria tempo e esforço calculando as distâncias de rearranjo e a comparação poderia ter sido realizada para $n = 13$.

5.1 Publicações

As contribuições apresentadas nesta dissertação foram publicadas, quase na sua totalidade, nos anais de congressos nacionais e internacionais de biologia computacional. Foram 8 publicações ao todo, divididas entre artigos completos e resumos estendidos tal como descrito a seguir:

- um resumo das seções 2.1 e 2.2 foi publicado nos anais do *26th ACM Symposium on Applied Computing - Conference Track on Bioinformatics and Computational Systems Biology (SAC-BIO'2011)* como um resumo estendido [26]. Mais tarde, dois artigos completos [27] e [28], o primeiro relativo ao que foi apresentado na Seção 2.1 e o segundo relativo ao que foi apresentado na Seção 2.2.1, foram publicados nos anais do *6th Brazilian Symposium on Bioinformatics (BSB'2011)*;

- um artigo completo [30] com os resultados das seções 2.3, 3.1 e 4.1 foi publicado nos anais do *4th International Conference on Bioinformatics and Computational Biology (BICoB'2012)*;

- um artigo completo [33] com os resultados das seções 3.2, 3.3, 3.6, 4.2, 4.3 e 4.6 foi publicado nos anais do *4th International Conference on Bioinformatics and Computational Biology (BICoB'2012)*;

- resultados preliminares dos resultados apresentados nas seções 3.5 e 4.5 foram publicados nos anais do *6th Brazilian Symposium on Bioinformatics (BSB'2011)* como um resumo estendido [29]. Mais tarde, uma versão mais completa foi publicada como um artigo completo [32] nos anais do *7th Brazilian Symposium on Bioinformatics (BSB'2012)*;

- por fim, um resumo dos resultados apresentados nas seções 3.4 e 4.4 foi publicado nos anais do *7th Brazilian Symposium on Bioinformatics (BSB'2012)* como um resumo estendido [31].

5.2 Trabalhos Futuros

Uma extensão imediata deste trabalho diz respeito à manutenção do GRAAu, o que inclui a implementação de futuras melhorias, como aumentar o número de modelos de rearranjo cobertos e estender a auditoria para permutações aleatórias de tamanhos arbitrariamente grandes. Além disso, outras possibilidades para trabalhos futuros são:

- estudar mais profundamente as medidas que nós propusemos, o diâmetro transversal e a longevidade, e validar as conjecturas a respeito delas;
- validar as conjecturas a respeito da justeza do fator de aproximação dos algoritmos;
- verificar a hipótese de que o algoritmo 2.25-aproximado para o Problema da Ordenação por Transposições desenvolvido por Walter, Dias e Meidanis [63] possui um fator de aproximação menor, possivelmente igual a 2;
- desenvolver algoritmos aproximados com fatores de aproximação melhorados para as variações do Problema da Ordenação por Rearranjo revisadas nesta dissertação. Em alguns casos, um ponto de partida seria analisar o funcionamento de algoritmos conhecidos quando estes recebem como entrada as permutações que demonstram a justeza de seus fatores de aproximação. Como comentado na Seção 1.1.2, esse tipo de análise pode levar à ideias que dão origem a algoritmos com fatores de aproximação melhores.

Referências Bibliográficas

[1] D. Bader, B. 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] B. Barney. POSIX threads programming. `https://computing.llnl.gov/tutorials/pthreads/`.

[5] M. Benoît-Gagné and S. Hamel. A new and faster method of sorting by transpositions. In *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM'2007)*, volume 4580 of *Lecture Notes in Computer Science*, pages 131–141, London, Ontario, Canada, 2007. Springer-Verlag.

[6] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-approximation algorithm for sorting by reversals. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 200–210, Rome, Italy, 2002. Springer-Verlag.

[7] L. Bulteau, G. Fertin, and I. Rusu. Pancake flipping is hard. In *37th International Symposium on Mathematical Foundations of Computer Science (MFCS'2012)*, volume 7464 of *Lecture Notes in Computer Science*, pages 247–258, Bratislava, Slovakia, 2012. Springer-Verlag.

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

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

[10] B. Chitturi, W. Fahle, Z. Meng, L. Morales, C. Shields, I. H. Sudborough, and W. Voit. An (18/11)n upper bound for sorting by prefix reversals. *Theoretical Computer Science*, 410(36):3372–3390, 2009.

[11] B. Chitturi and I. H. Sudborough. Bounding prefix transposition distance for strings and permutations. *Theoretical Computer Science*, 421:15–24, 2012.

[12] 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, San Francisco, California, United States, 1998. Society for Industrial and Applied Mathematics.

[13] D. S. Cohen and M. Blum. On the problem of sorting burnt pancakes. *Discrete Applied Mathematics*, 61(2):105–120, 1995.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[15] U. Dias and Z. Dias. An improved 1.375-approximation algorithm for the transposition distance problem. In *Proceedings of the First ACM International Conference on Bioinformatics and Computational Biology (BCB'2010)*, pages 334–337, Niagara Falls, New York, 2010. ACM Press.

[16] Z. Dias and J. Meidanis. Sorting by prefix transpositions. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE'2002)*, volume 2476 of *Lecture Notes in Computer Science*, pages 65–76, Lisbon, Portugal, 2002. Springer-Verlag.

[17] T. Dobzhansky and A. H. Sturtevant. Inversions in the chromosomes of *Drosophila pseudoobscura*. *Genetics*, 23(1):28–64, 1938.

[18] H. Dweighter. Problem e2569. *American Mathematical Monthly*, 82:1010, 1975.

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

[20] H. Eriksson, K. Eriksson, J. Karlander, L. Svensson, and J. Wastlund. Sorting a bridge hand. *Discrete Mathematics*, 241(1-3):289–300, 2001.

[21] S. Even and O. Goldreich. The minimum-length generator sequence problem is NP-hard. *Journal of Algorithms*, 2(3):311–313, 1981.

[22] X. Feng, I. H. Sudborough, and E. Lu. A fast algorithm for sorting by short swap. In *IASTED International Conference on Computational and Systems Biology (CASB'2006)*, pages 62–67, Dallas, Texas, USA, 2006. ACTA Press.

[23] G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette. *Combinatorics of Genome Rearrangements*. The MIT Press, 2009.

[24] J. Fischer and S. W. Ginzinger. A 2-approximation algorithm for sorting by prefix reversals. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'2005)*, volume 3669 of *Lecture Notes in Computer Science*, pages 415–425, Mallorca, Spain, 2005. Springer-Verlag.

[25] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

[26] G. R. Galvão and Z. Dias. Computing rearrangement distace of every permutation in the symmetric group. In *Proceedings of the 26th ACM Symposium on Applied Computing - Conference Track on Bioinformatics and Computational Systems Biology (SAC-BIO'2011)*, pages 106–107, Taichung, Taiwan, 2011. ACM Press.

[27] G. R. Galvão and Z. Dias. A flexible framework for computing rearrangement distance of every permutation in the symmetric group. In *Proceedings of the 6th Brazilian Symposium on Bioinformatics (BSB'2011)*, pages 33–40, Brasília, Brazil, 2011.

[28] G. R. Galvão and Z. Dias. On the distribution of rearrangement distances. In *Proceedings of the 6th Brazilian Symposium on Bioinformatics (BSB'2011)*, pages 41–48, Brasília, Brazil, 2011.

[29] G. R. Galvão and Z. Dias. On the performance of sorting by transpositions without using cycle graph. In *Proceedings of the 6th Brazilian Symposium on Bioinformatics (BSB'2011)*, pages 69–72, Brasília, Brazil, 2011.

[30] G. R. Galvão and Z. Dias. GRAAu: Genome Rearrangement Algorithm Auditor. In *Proceedings of the 4th International Conference on Bioinformatics and Computational Biology (BICoB'2012)*, pages 97–101, Las Vegas, Nevada, USA, 2012. Curran Associates, Inc.

[31] G. R. Galvão and Z. Dias. On the approximation ratio for sorting by short swaps. In *Proceedings of the 7th Brazilian Symposium on Bioinformatics (BSB'2012)*, pages 120–125, Campo Grande, Mato Grosso do Sul, Brazil, 2012.

[32] G. R. Galvão and Z. Dias. On the approximation ratio of algorithms for sorting by transpositions without using cycle graphs. In *Proceedings of the 7th Brazilian Symposium on Bioinformatics (BSB'2012)*, volume 7049 of *Lecture Notes in Computer Science*, pages 25–36, Campo Grande, Mato Grosso do Sul, Brazil, 2012. Springer-Verlag.

[33] G. R. Galvão and Z. Dias. On the performance of sorting permutations by prefix operations. In *Proceedings of the 4th International Conference on Bioinformatics and Computational Biology (BICoB'2012)*, pages 102–107, Las Vegas, Nevada, USA, 2012. Curran Associates, Inc.

[34] O. Gascuel. *Mathematics of Evolution and Phylogeny*. Oxford University Press, Inc., New York, NY, USA, 2005.

[35] S. Grusea and A. Labarre. The distribution of cycles in breakpoint graphs of signed permutations. *CoRR*, abs/1104.3353, 2011.

[36] Q. Gu, S. Peng, and I. H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoritical Computer Science*, 210(2):327–339, 1999.

[37] S. A. Guyer, L. S. Heath, and J. P. C. Vergara. Subsequence and run heuristics for sorting by transpositions. Technical Report TR-97-20, Virginia Polytechnic Institute & State University, 1997.

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

[39] T. Hartman and R. Sharan. A 1.5-approximation algorithm for sorting by transpositions and transreversals. *Journal of Computer and System Sciences*, 70(3):300–320, 2005.

[40] L. S. Heath and J. P. C. Vergara. Sorting by short swaps. *Journal of Computational Biology*, 10(5):775–789, 2003.

[41] M. H. Heydari and I. H. Sudborough. On the diameter of the pancake network. *Journal of Algorithms*, 25(1):67–94, 1997.

[42] K. Kawaguchi. bzip2 library from Apache Ant. `http://www.bzip.org/`.

[43] J. D. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1-2):80–110, 1995.

[44] A. Labarre. *Combinatorial aspects of genome rearrangements and haplotype networks*. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, 2008.

[45] A. Labarre. Edit distances and factorisations of even permutations. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'2008)*, volume 5193 of *Lecture Notes in Computer Science*, pages 635–646, Karlsruhe, Germany, 2008. Springer-Verlag.

[46] A. Labarre. Lower bounding edit distances between permutations. *CoRR*, abs/1201.0365, 2012.

[47] D. H. Lehmer. Teaching combinatorial tricks to a computer. *Proceedings of Symposia in Applied Mathematics*, 10:179–193, 1960.

[48] L. Lu and Y. Yang. A lower bound on the transposition diameter. *SIAM Journal on Discrete Mathematics*, 24(4):1242–1249, 2010.

[49] M. Mares and M. Straka. Linear-time ranking of permutations. In L. Arge, M. Hoffmann, and E. Welzl, editors, *Proceedings of the 15th Annual European Symposium on Algorithms (ESA'2007)*, volume 4698 of *Lecture Notes in Computer Science*, pages 187–193, Eilat, Israel, 2007. Springer Berlin, Heidelberg.

[50] J. Meidanis, M. E. M. T. Walter, and Z. Dias. A lower bound on the reversal and transposition diameter. *Journal of Computational Biology*, 9(5):743–745, 2002.

[51] W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

[52] 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.

[53] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, 4(1):406–425, 1987.

[54] J. Seward. bzip2. `http://www.kohsuke.org/bzip2/`.

[55] M. Sharmin, R. Yeasmin, M. Hasan, A. Rahman, and M. S. Rahman. Pancake flipping with two spatulas. *Electronic Notes in Discrete Mathematics*, 36:231–238, 2010.

[56] A. H. Sturtevant and T. Dobzhansky. Inversions in the third chromosome of wild races of drosophila pseudoobscura, and their use in the study of the history of the species. *Proceedings of the National Academy of Sciences of the United States of America*, 22(7):448–450, 1936.

[57] E. Tannier, A. Bergeron, and M. F. Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155(6-7):881–888, 2007.

[58] G. Tesler. GRIMM: genome rearrangements web server. *Bioinformatics*, 18:492–493, 2002.

[59] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[60] J. P. C. Vergara. *Sorting by bounded permutations*. PhD thesis, Virginia Polytechnic Institute & State University, Blacksburg, VA, USA, 1998.

[61] M. E. M. T. Walter. *Algoritmos para Problemas em Rearranjo de Genomas*. PhD thesis, University of Campinas, Brazil, 1999. In Portuguese.

[62] M. E. M. T. Walter, Z. Dias, and J. Meidanis. Reversal and transposition distance of linear chromosomes. In *Proceedings of the 5th International Symposium on String Processing and Information Retrieval (SPIRE'1998)*, pages 96–102, Santa Cruz, Bolivia, 1998. IEEE Computer Society.

[63] M. E. M. T. Walter, Z. Dias, and J. Meidanis. A new approach for approximating the transposition distance. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'2000)*, pages 199–208, Washington, DC, USA, 2000. IEEE Computer Society.

[64] M. E. M. T. Walter, M. C. Sobrinho, E. T. G. Oliveira, L. S. Soares, A. G. Oliveira, T. E. S. Martins, and T. M. Fonseca. Improving the algorithm of Bafna and Pevzner for the problem of sorting by transpositions: a practical approach. *Journal of Discrete Algorithms*, 3(2-4):342–361, 2005.

[65] G. A. Watterson, W. J. Ewens, T. E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99(1):1–7, 1982.