Universidade Estadual de Campinas
Instituto de Computação

Thiago da Silva Arruda

O Problema da Ordenação por Reversões Ponderadas

CAMPINAS
2016

Thiago da Silva Arruda

O Problema da Ordenação por Reversões Ponderadas

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Zanoni Dias

Este exemplar corresponde à versão final da Dissertação defendida por Thiago da Silva Arruda e orientada pelo Prof. Dr. Zanoni Dias.

CAMPINAS
2016

Agência(s) de fomento e nº(s) de processo(s): CNPq, 132198/2012-6; CNPq, 1611593/2012-7

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

	Arruda, Thiago da Silva, 1988-
Ar69p	O problema da ordenação por reversões ponderadas / Thiago da Silva Arruda. – Campinas, SP : [s.n.], 2016.
	Orientador: Zanoni Dias.
	Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.
	1. Biologia computacional. 2. Genomas. 3. Algoritmos heurísticos. 4. Meta-heurísticas. 5. Permutações (Matemática). I. Dias, Zanoni,1975-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Sorting by length-weighted inversions
Palavras-chave em inglês:
Computational biology
Genomes
Heuristic algorithms
Metaheuristics
Permutations (Mathematics)
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Zanoni Dias [Orientador]
Maria Emilia Machado Telles Walter
Fábio Luiz Usberti
Data de defesa: 28-04-2016
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Thiago da Silva Arruda

O Problema da Ordenação por Reversões Ponderadas

Banca Examinadora:

- Prof. Dr. Zanoni Dias
  Instituto de Computação - UNICAMP
- Profa. Dra. Maria Emilia Machado Telles Walter
  Departamento de Ciência da Computação - CIC - UnB
- Prof. Dr. Fábio Luiz Usberti
  Instituto de Computação - UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 28 de abril de 2016

Agradecimentos

A todos os membros da minha família, em especial aos meus pais Ademir e Lucilene, e meus irmãos Joabe e Jorge, por todo o incentivo, confiança e suporte.

Ao meu orientador Zanoni Dias e também ao Ulisses Dias, por todo a ajuda e determinação ao longo dessa jornada, fundamental para a conclusão deste trabalho.

Aos amigos que conheci na Unicamp e que enriqueceram muito minha experiência durente este período, em especial os amigos do laboratório LOCo.

Aos meus amigos de diversos lugares e momentos que me apoiram até aqui.

Resumo

Rearranjos de genomas são eventos de mutação globais que agem sobre grandes trechos de um genoma, diferentemente das mutações mais comuns que afetam pontualmente o genoma. Na maioria dos casos, computar a distância de rearranjo de genomas entre dois genomas resulta em problemas NP-difíceis. Em um problema da ordenação por Rearranjo de Genomas desejamos calcular a sequência de eventos de rearranjos necessários para ordenar blocos conservados de genomas com custo mínimo, onde os genomas são representados por permutações. Um modelo de rearranjo determina quais eventos de rearranjo são permitidos para ordenar uma permutação ou transformar uma permutação em outra. Neste trabalho abordamos o modelo de eventos de reversão, que ocorre quando um segmento do genoma é revertido. No problema clássico de ordenação por reversões, o custo para qualquer reversão em uma permutação é unitário, independente do tamanho do segmento a ser revertido. Novos estudos mostraram que a mecânica de rearranjo de genomas sugere que reversões com tamanho de segmento menor ocorrem com maior frequência. Desta forma, abordamos neste trabalho o problema de ordenação por reversões ponderadas, onde o custo de cada reversão é igual ao tamanho do segmento revertido. Para obter soluções para o problema, inicialmente nós desenvolvemos um algoritmo heurístico polimomial para a variação do problema em que a orientação dos genes não é considerada (que resulta em uma permutação sem sinal), e obtivemos resultados experimentais superiores aos algoritmos anteriormente conhecidos, considerando todas as permutações de tamanho até 10 e amostras de permutações maiores de tamanho até 100. Em seguida desenvolvemos uma Meta-heurística iterativa que segue a estratégia GRASP (Greedy Randomized Adaptive Search Procedure). Este método foi aplicado também à variação do problema em que a orientação dos genes é considerada (que resulta em uma permutação com sinal). Para ambas variações nós executamos experimentos para amostras de permutações de tamanho no intervalo de 10 a 100 e mostramos que nosso método apresenta resultados significativamente superiores a todas as outras abordagens anteriores.

Abstract

Genome rearrangements are global mutation events that act on large stretches of a genome. Those are unlike the more common mutations, that affect the genome punctually. In many cases, the computation of genome rearrangement distance between two genomes induces NP-Hard problems. Genome sorting problems aim to calculate a sequence of rearrangement events required to sort conserved genomes blocks with minimum cost, where genomes are represented by permutations. A rearrangement model determines which rearrangement events are allowed to sort a permutation or transform a permutation into another. In this work, we deal with inversion events, which occur when a segment of DNA sequence in the genome is reversed. In the classic problem of sorting by inversions, the cost of any inversion in a permutation is unitary, independent of segment size to be reversed. Further studies showed that reversals with lower segment size occur with greater frequency. In this way, we address the sorting by weighted inversions problem and, in our model, each inversion costs the number of elements in the reversed segment. In order to generate solutions for this problem, we initially developed a polynomial heuristic algorithm for the problem variation where the gene orientation is not taken in consideration (where we use unsigned permutations). With this first method we ran experiments, taking as input all permutations of size up to 10 and sample permutations with size up 100, and we obtained results that outperform previously known algorithms. Then we developed an iterative meta-heuristic that follows the GRASP strategy (Greedy Randomized Adaptive Search Procedure). This time the method supported as well the variation of the problem that takes genes orientation in consideration (which results in a signed permutation). For both variations we ran experiments with samples of permutation with size in the range from 10 to 100, and we show that our method provided significantly better results than all the other previously known approaches.

Lista de Figuras

2.1 Este gráfico mostra a porcentagem de soluções exatas produzidos pela configuração de parâmetro $(1, B, 0)$, onde B é o valor dado no eixo-X. Nós observamos que para $n \leq 7$ os melhores resultados são encontrados quando definimos B entre 1 e 2. Por outro lado, para $n = 8$ e $n = 9$, escolhas de B no intervalo $[1, 7]$ levam a aproximadamente a mesma porcentagem de soluções exatas. Para $n = 10$, podem ser encontradas mais soluções exatas fixando B entre 5 e 6. 35
2.2 Este gráfico mostra a taxa de aproximação média produzida pela configuração de parâmetro $(1, B, 0)$, onde B é o valor dado no eixo-X. É observado que para $n \leq 7$ este gráfico mostra concordância com a Figura 2.1, e a melhor configuração de parâmetro tem o valor de B entre 1 e 2. Em todos estes casos, nós concluímos que os piores resultados são obtidos quando o valor de B é maior do que 7. Nós também notamos que $B = 0$ produz resultados ruins. 36
2.3 Este gráfico mostra a melhoria obtida executando nossa heurística várias vezes em permutações pequenas. Cada curva no gráfico mostra a taxa de aproximação média, quando consideramos um dado número de configurações de parâmetros, escolhidas aleatoriamente. Também é mostrada a curva `Bender`, que corresponde ao algoritmo de aproximação $O(\lg n)$ [10]. Analisando este gráfico nós verificamos que, quando são escolhidas aleatoriamente 10 configurações de parâmetros, é obtida uma boa relação de custo-benefício entre qualidade da solução e tempo de execução, sendo que é obtido uma taxa de aproximação significativamente melhor do que quando usamos somente uma configuração de parâmetros. Também notamos que quando empregado mais do que 10 configurações de parâmetros os melhoramentos passam a ser menos substanciais. 37
2.4 Este gráfico mostra o resultado obtido ao executar experimentos com nossa heurística em permutações grandes. Os parâmetros A e B são definidos no intervalo [0..10], e o parâmetro C é fixado em 0. Cada curva no gráfico cuja legenda é um número mostra a taxa de aproximação média quando consideramos esse número de configurações aleatórias de parâmetros. A curva `All` representa o melhor resultado quando todas as configurações de parâmetros são consideradas. A curva `2-Approx` corresponde aos resultados do algoritmo de Kececioglu e Sankoff [31]. Enquanto a curva `Bender` corresponde aos resultados do algoritmo de aproximação $O(\lg n)$ [10]. . . . 38

3.1 Este gráfico mostra uma análise comparativa do custo médio entre os resultados para permutações com sinal. A curva `PARAMETRICO` refere-se aos resultados gerados para permutações com sinal a partir do método paramétrico apresentado no Capítulo 2. A curva `GRIMM` refere-se aos resultados fornecidos pelo algoritmo GRIMM [44]. 44
3.2 Este gráfico mostra uma análise comparativa do número de reversões entre os resultados para permutações com sinal. A curva `PARAMETRICO` refere-se aos resultados gerados para permutações com sinal a partir do método paramétrico apresentado no Capítulo 2. A curva `PARAMETRICO` refere-se aos resultados fornecidos pelo algoritmo GRIMM [44]. 45
3.3 Este gráfico mostra uma análise comparativa do custo médio entre os resultados para permutações com sinal. A curva `GRIMM` representa o algoritmo para o problema de ordenação de permutações com custos unitários, a curva `PARAMETRICO` representa o método apresentado anteriormente na Seção 3.1.1, a curva `ANCORAS` representa o método *GRASP* com âncoras, que apresentamos nesta seção. 46
3.4 Este gráfico mostra uma análise comparativa do número de reversões entre os resultados para permutações com sinal. A curva `GRIMM` representa o algoritmo para o problema de ordenação de permutações com custos unitários, a curva `PARAMETRICO` representa o método nós apresentamos anteriormente na Seção 3.1.1, a curva `ANCORAS` representa o método *GRASP* com âncoras que apresentamos nesta seção. 46
3.5 Este gráfico mostra a porcentagem de vezes que nosso algoritmo melhora a solução inicial. Em geral, a solução inicial foi melhorada em 94.0% dos casos. Se nós considerarmos somente permutações grandes tais que $n \geq 50$, nós observamos que 99.3% da soluções iniciais foram melhoradas. Nesse gráfico, os padrões de preenchimento das barras representam tamanhos de janelas e barras nos mostram quais tamanhos de janelas foram responsáveis pela primeira melhoria na solução inicial.Notamos que para o tamanho $N = 10$, nas iterações para $f = 12$ e $f = 14$ foi utilizado $f = 10$ na prática. 56
3.6 Este gráfico mostra a porcentagem de melhoramento obtido por nosso algoritmo. Observamos que nossa solução custa em torno 12.6% menos que a solução inicial. Nesse gráfico, os padrões de preenchimento das barras representam tamanhos de janelas e as barras mostram qual tamanho de janela foi responsável pelo melhoramento em média. Notamos que para o tamanho $N = 10$, nas iterações para $f = 12$ e $f = 14$ foi utilizado $f = 10$ na prática. 57
3.7 Este gráfico mostra uma análise comparativa entre 3 algoritmos. O eixo-Y representa a média de custo e o eixo-X representa o tamanho da permutação. O rótulo `Swidan` representa o algoritmo proposto por Swidan *et al.* [40]. `GRIMM` é uma solução ótima para o problema de ordenação por reversões de custo unitario. `GRASP` representa nossa abordagem. 58
3.8 Este gráfico mostra a média do número de reversões de soluções obtidas por cada um dos três algoritmos usados na nossa análise. O eixo-Y representa a média no número de reversões e o eixo-X representa o tamanho da permutação. Os rótulos para algoritmos são os mesmo usados na Figura 3.7. 59

3.9 Este gráfico mostra a porcentagem de vezes que a nossa heurística melhorou a solução inicial em permutações com sinal. Em geral, a solução inicial foi melhorada em 97.1% dos casos. Se nós considerarmos somente permutações grandes, tal como $n \geq 50$, nós observamos que 99.9% das soluções iniciais são melhoradas. Em nosso gráfico, nós especificamos o intervalo de iterações que geraram o primeiro melhoramento na solução inicial, o que nos permitiu observar que 91.2% dos melhoramentos ocorreram com 50 iterações. 61
3.10 Este gráfico mostra a porcentagem de vezes que a nossa heurística melhora a solução inicial em permutações sem sinal. Em geral, a solução inicial foi melhorada em 96.8% dos casos. Se considerarmos somente permutações grandes como $n \geq 50$, observamos que 99.7% das soluções iniciais foram melhoradas. No nosso gráfico nós especificamos o intervalo de iterações que gerou o primeiro melhoramento na solução inicial, o que nos permite obserar que 86.2% dos melhoramentos ocorrem com 50 iterações. 62
3.11 Este gráfico mostra a porcentagem de melhoria obtida usando nosso método em permutações com sinal. Observamos que nossas soluções custam por volta de 17.9% menos do que a solução inicial. O histograma também mostra que o melhoramento médio após um dado número de iterações ter sido executado. Observamos que a tendência de que mais iterações devem ser executadas conforme se aumenta o tamanho das permutações. 63
3.12 Este gráfico mostra a porcentagem de melhoramento obtida usando nosso método em permutações com sinal. Nós observamos que as soluções fornecidas pelo nosso método custam por volta de 16.6% menos do que a solução inicial. Em geral o comportamento mostrado aqui é similar ao caso com sinal. 64
3.13 Este gráfico mostra uma análise comparativa entre três algoritmos considerando somente permutações com sinal. O eixo-Y representa o custo médio e o eixo-X representa o tamanho da permutação. O rótulo `Swidan` representa o algoritmo proposto por Swidan *et al.* [40]. `GRIMM` [44] fornece soluções ótimas para o problema de ordenação por reversões em que cada reversão possui custos unitários. Curvas rotuladas com números representam nossa abordagem, onde cada número indica o número de iterações que usamos. . 65
3.14 Este gráfico mostra uma análise comparativa entre cinco algoritmos. O eixo-Y representa o custo médio e o eixo-X representa o tamanho da permutação. O rótulo `GRIMM` representa a solução inicial, `Bender` representa o algoritmo proposto por Bender *et al.* [10], `Christie` representa o algoritmo proposto por Christie [18] para ordenação de reversões de custos unitários, o rótulo `Arruda` representa o método guloso previamente proposto por Arruda *et al.* [2] para ordenação por reversões ponderadas, e os números indicam quantas iterações foram executas para serem obtidos os resultados apresentados. 66

Lista de Tabelas

2.1 Resultados para todas as permutações pequenas ($2 \geq n \leq 10$). A tabela é dividida em 9 partes e cada parte corresponde ao conjunto de todas as permutações de um tamanho n. Observamos que ambos os métodos $\texttt{TEST}_1$ e $\texttt{TEST}_2$ superam os demais. Eles fornecem, em geral, uma porcentagem maior de soluções exatas (%EX), e os custos médios de ordenação são menores do que dos outros métodos. 32

2.2 As 50 melhores configurações de parâmetro (A, B, C) da heurística em relação à aproximação média para todas as pemutações de tamanho $n = 10$, o que totaliza 10! permutações. A tabela mostra que os melhores resultados possuem baixo valor para o parâmetro C. Em especial $C = 0$ ocorre frequentemente dentre estas configurações. 34

2.3 Custo médio quando é empregado um dado número de configurações. Melhor identifica os casos onde o dado número de melhores configurações foi empregado. Aleatório identifica os casos onde as configurações foram escolhidas aleatoriamente. Gap mostra a distância entre os grupos de configurações Melhor e Aleatório. 40

3.1 Tempo médio (em segundos) para processar cada permutação de um dado tamanho. Para cada tamanho, nós geramos 1000 permutações aleatórias. . 55

3.2 Esta tabela mostra uma análise comparativa do custo médio entre algoritmos para a variação do problema para permutações com sinal. Esta tabela mostra resultados para permutações de tamanho 10 até 100. Os números 50, 250 e 2000 indicam quantas iterações do nosso método foram executas para serem obtidos os respectivos resultados. O rótulo Swidan representa o algoritmo proposto por Swidan *et al.* [40]. 66

3.3 Esta tabela mostra uma análise comparativa do custo médio entre algoritmos para a variação do problema para permutações sem sinal. Esta tabela mostra resultados para permutações de tamanho 10 até 100. GRIMM é o rótulo para solução inicial. Os números 50, 250 e 2000 indicam quantas iterações do nosso método foram executas para serem obtidos os respectivos resultados. Arruda [2] é o rótulo para os resultados do nosso primeiro trabalho para o problema. O rótulo Kececioglu representa o algoritmo 2-aproximado proposto por Kececioglu [31]. O rótulo Swidan representa o algoritmo proposto por Swidan *et al.* [40]. 67

Sumário

1 Introdução **13**

1.1 Rearranjo de genomas 13

1.1.1 Ordenação por reversões 14

1.1.2 Ordenação por reversões ponderadas 15

1.1.3 Propriedades de permutações 17

1.2 GRASP 19

1.3 Objetivos 21

1.4 Organização da Dissertação 22

2 Algoritmo Heurístico Paramétrico **23**

2.1 Heurística paramétrica 23

2.1.1 Análise do algoritmo 25

2.2 Algoritmo para gerar soluções exatas 28

2.3 Resultados 29

2.3.1 Resultados para permutações pequenas 30

2.3.2 Análise de configurações de parâmetros 32

2.3.3 Resultados para permutações grandes 37

2.4 Publicação 41

3 Meta-Heurística GRASP **42**

3.1 Histórico da pesquisa 43

3.1.1 Adaptação do método paramétrico 43

3.1.2 Método GRASP com âncoras 44

3.2 Vizinhança 47

3.3 Estrutura da Meta-heurística 48

3.4 Construção de soluções 49

3.5 Resultados 54

3.5.1 Primeira abordagem 54

3.5.2 Segunda abordagem 58

3.6 Publicações 64

4 Conclusões **68**

Referências Bibliográficas **70**

Capítulo 1

Introdução

Em termos biológicos, o genoma de um indivíduo é constituído de cromossomos, os quais são formados por genes ou blocos conservados. Um genoma representa o conjunto hereditário de informações genéticas de um indivíduo. Durante a evolução, diferentes espécies surgiram devido a mudanças genéticas.

Um evento de rearranjo de genomas é uma mutação que afeta um trecho do genoma. Os tipos de rearranjos (denominados operações) mais comuns são: reversão, transposição, translocação, fusão e fissão. Durante um evento de rearranjo, o cromossomo é quebrado e os segmentos resultantes são unidos, de forma que o conjunto inicial de genes é preservado, mas a ordem dos mesmos pode ser alterada.

Um problema de rearranjo de genomas consiste em comparar genomas de dois indivíduos para encontrar a menor sequência de eventos de rearranjo que transforma um genoma em outro. O tamanho desta sequência é denominado distância evolucionária entre indivíduos, considerando o princípio da máxima parcimônia [1].

Modelos teóricos podem ser construídos para determinar relações de parentesco entre espécies, analisando seus respectivos códigos genéticos. Neste contexto, o foco da pesquisa em Biologia Computacional consiste em desenvolver algoritmos que determinem o custo mínimo de operações necessárias para que, em um dado modelo, um genoma seja transformado em outro.

Nas seções a seguir serão introduzidos, com uma revisão bibliográfica, o tópico rearranjo de genomas e mais especificamente os modelos de reversões com custos unitários e reversões ponderadas. Também serão apresentados os objetivos deste trabalho.

1.1 Rearranjo de genomas

Para a formulação de algoritmos, um genoma com n genes é classicamente representado como uma *n-tupla* de números inteiros, na qual cada número representa um gene. Considerando que esta *n-tupla* não possui repetição, então pode ser definida como uma permutação $\pi = (\pi_1\, \pi_2\, \pi_3 \ldots \pi_n)$, $1 \leq |\pi_i| \leq n$, $|\pi_i| = |\pi_j| \leftrightarrow i = j$, em que cada elemento π_i possui um sinal, que indica a orientação do gene correspondente. Quando não se

[1] Método estatístico baseado na noção filosófica de William Ockham: a melhor hipótese para explicar um processo é aquela que requer o menor número de suposições.

conhece a orientação dos genes, o sinal é omitido.

Um modelo de rearranjo é um conjunto de operações permitidas para transformar um genoma em outro. As principais operações propostas na literatura são: *reversão*, *transposicão*, *troca de bloco*, *translocação*, *fissão* e a *fusão*.

Dado um modelo de rearranjo M, a distância de rearranjo entre duas permutações π e σ é o número mínimo t de operações ρ_i, pertencentes a M, necessário para transformar π em σ:

$$\sigma = (((\pi \, \rho_1) \, \rho_2 \,) \, ...) \, \rho_t$$

Esta distância pode ser denotada por $d_M(\pi, \sigma)$. A maior distância entre quaisquer duas permutações de tamanho n, em relação ao modelo M, é denominada diâmetro, denotada por $D_M(n)$.

A permutação identidade ι é definida como $(1\,2\,3\,...\,n)$. A ordenação de uma permutação α consiste no processo de transformar α na permutação identidade. A distância de ordenação de α, em relação a um modelo M, é denotada por $d_M(\alpha, \iota) = d_M(\alpha)$. A distância de rearranjo entre π e σ é equivalente à distância entre $\alpha = \pi\sigma^{-1}$ e $\iota = \sigma\sigma^{-1}$, logo $d_M(\pi, \sigma) = d_M(\alpha) = d_M(\alpha, \iota)$.

1.1.1 Ordenação por reversões

Um evento de rearranjo de genomas chamado reversão ocorre quando um segmento do genoma é invertido. Em bactérias, reversões são notadas como o mais frequente evento de rearranjo. Dada a ordem dos genes de dois genomas contemporâneos, a tarefa de encontrar o número mínimo de reversões que transforma um genoma em outro é chamada de problema de ordenação por reversões.

Uma operação de reversão em uma permutação π consiste em inverter a ordem dos elementos de um segmento. Dessa forma, a reversão $\rho(i, j)$, onde $1 \leq i \leq j \leq n$, resulta em:

$$\pi \cdot \rho = (\pi_1 \; ... \; \pi_{i-1} \; \underline{\pi_j \; \pi_{j-1} \; ... \; \pi_{i+1} \; \pi_i} \; \pi_{j+1} \; ... \; \pi_n).$$

Quando não se conhece a orientação dos genes, tem-se o problema de ordenação por reversões sem sinal, que foi provado ser NP-Difícil por Caprara [15]. Bafna e Pevzner [8] fizeram os primeiros estudos sobre este problema, o que resultou em um algoritmo de aproximação de fator 1,75. Posteriormente, Christie [17] apresentou um algoritmo de aproximação de fator 1,5. O melhor algoritmo conhecido atualmente foi desenvolvido por Berman e coautores [12] e possui fator de aproximação 1,375.

Para o caso em que se conhece a orientação dos genes, denominado problema de ordenação por reversões com sinal, existem algoritmos polinomiais exatos, sendo que o primeiro foi apresentado por Hannenhalli e Pevzner [28]. Atualmente o algoritmo mais eficiente disponível na literatura possui complexidade sub-quadrática [42]. Para o caso em que se deseja saber apenas o valor distância de reversão, há um algoritmo linear disponível [6].

Uma variação do problema, bastante estudada na literatura, consiste em um modelo de rearranjo em que as reversões sempre ocorrem no prefixo da permutação, conhecido como Problema de Ordenação de Panquecas [26, 30]. Atualmente o melhor algoritmo

conhecido, desenvolvido por Fischer e Ginzinger [25], possui fator de aproximação 2.

1.1.2 Ordenação por reversões ponderadas

No problema clássico de ordenação por reversões, o custo para qualquer reversão em uma permutação é unitário, independente do tamanho do segmento a ser revertido. Este modelo foi aplicado, por exemplo, para algumas pesquisas genéticas em plantas [32] e vírus [27]. Entretanto, a mecânica de rearranjo de genomas sugere que reversões com tamanho de segmento menor ocorrem com maior frequência [36].

No trabalho de Darling, Miklós e Ragan [19] foram estudados oito genomas da família *Yersinia* e foram encontradas reversões mais curtas do que esperado considerando um modelo neutro. O entendimento de que reversões curtas são preferidas não é inteiramente novo. Resultados anteriores sugerem que a sequência de operações que têm maior probabilidade de ocorrer durante a evolução podem não envolver o movimento de muitas sequências longas [13]. Sankoff [37] encontrou evidências que a frequência de reversões em segmentos curtos é alta na evolução de genomas microbiais. Por outro lado, isto parece menos prevalente em plantas e animais. Lefebvre *et al.* [33] e Sankoff *et al.* [38] encontraram uma proporção muito maior de reversões pequenas, especialmente aquelas envolvendo um único gene, o que contrasta com a hipótese nula de que as duas extremidades de uma reversão ocorrem aleatoriamente e independentemente.

Desta forma, novos trabalhos passaram a considerar o problema da ordenação tal que o custo de uma operação de reversão é dado em função do tamanho do segmento que sofrerá mutação [7, 34, 41].

Esta linha de pesquisa trata do problema de ordenação por reversões ponderadas. Alguns resultados foram apresentados tanto para o caso em que a orientação dos genes é considerada [40], quanto para o caso quando não é considerada [10, 34].

O objetivo do problema consiste em minimizar o custo total das reversões realizadas para ordenar uma permutação π, ou seja, minimizar $d_l(\pi) = f(\rho_1) + f(\rho_2) + ... + f(\rho_k)$, tal que $\iota = \pi \cdot \rho_1 \cdot \rho_2 \cdot ... \cdot \rho_k$. A Definição 1 especifica o modelo geral do problema [10], em que o custo de uma reversão ρ é obtido pela função $f(\rho) = |\rho|^k$. Neste trabalho é tratado o problema particular em que $k = 1$ e, desta forma, $f(\rho) = |\rho| = l$.

Definição 1. *No modelo de ordenação por reversões ponderadas, o custo de uma reversão ρ é uma função polinomial: $f(\rho) = l^k$, onde $l = |\rho|$ e $k \in \mathbb{R}_*^+$, tal que, para uma reversão $\rho = (i, j)$, $|\rho| = j - i + 1$.*

Como exemplo, considere a sequência de eventos de reversão que ordena o genoma $\pi = (1\ 7\ 6\ 5\ 2\ 3\ 4\ 8\ 9)$. Neste caso, temos que $d_l(\pi) = f(\rho(5, 7)) + f(\rho(2, 7)) = 3 + 6 = 9$, e $\pi \cdot \rho(5, 7) \cdot \rho(2, 7) = \iota$:

$$\pi = (\ 1\ 7\ 6\ 5\ 2\ 3\ 4\ 8\ 9\ )$$

$$(\ 1\ 7\ 6\ 5\ \boxed{2\ 3\ 4}\ 8\ 9\ ) \rightarrow \rho(5,7),\ f(\rho) = 7 - 5 + 1 = 3$$

$$(\ 1\ 7\ 6\ 5\ 4\ 3\ 2\ 8\ 9\ )$$

$$(\ 1\ \boxed{7\ 6\ 5\ 4\ 3\ 2}\ 8\ 9\ ) \rightarrow \rho(2,7),\ f(\rho) = 7 - 2 + 1 = 6$$

$$(\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ ) = \iota$$

A seguir, nós apresentamos o algoritmo $O(\lg n)$ aproximado apresentado por Bender e coautores [10] e também os limitantes obtidos naquele trabalho.

Algoritmo aproximado

Bender e coautores [10] primeiramente abordaram o problema de ordenação por reversões ponderadas de sequências binárias. Uma sequência T é dita binária se $T = t_1, t_2, ..., t_n$, com $t_i \in \{0, 1\}$. Uma sequência binária é dita *ordenada* se for composta por uma única subsequência de $0's$, seguida por uma única subsequência de $1's$. Para ordenar sequências binárias por reversões ponderadas, o trabalho apresenta um algoritmo exato que emprega programação dinâmica.

Este algoritmo, denominado *zerOneSort*, possui complexidade de tempo $O(n^4)$ e complexidade de espaço $O(n^2)$, contudo, modificando a função de custo de programação dinâmica, é possível reduzir a complexidade de tempo para $O(n^3)$. Mais detalhes desses algoritmos podem ser encontrados no trabalho de Bender e coautores [10].

Seja s a mediana da permutação π, π pode ser convertida em uma sequência binária T da seguinte forma:

$$t_i = \begin{cases} 0, & \text{se } \pi_i < s, \\ 1, & \text{caso contrário.} \end{cases}$$

No mesmo trabalho é apresentado o algoritmo $O(\lg n)$-aproximado para a ordenação de permutações, o qual utiliza, como sub-rotina, a ordenação de sequência binárias (Algoritmo 1), onde ϱ representa uma sequência de reversões. Neste algoritmo é empregada a técnica de divisão e conquista, de forma semelhante ao *Quicksort*, onde a operação equivalente à partição é executada pelo algoritmo *zerOneSort* sobre a sequencia binária T, que, como vimos, pode ser extraída da permutação π. O algoritmo retorna, como saída, a sequência de reversões resultante da concatenação das reversões de particionamento e de cada sequência que ordena uma metade da permutação. O operador $\|$ representa a concatenação de sequências.

Limitantes

Quando a orientação dos genes não é considerada, Pinter e Skiena [34] desenvolveram um algoritmo que garante uma fator de aproximação $O(\lg n)$. Bender *et al.* [10] propuseram algoritmos de aproximação para uma grande classe de funções de custo $f(l) = l^k$, para

Algoritmo 1 SortPermtuationDivideAndConquer

1: **Data:** π
2: **se** π está ordenada **então**
3: **return** $\emptyset$
4: **else**
5: $T \leftarrow permToBin(\pi)$
6: $\varrho \leftarrow zerOneSort(T)$
7: $\pi \leftarrow \pi \cdot \varrho$
8: $\varrho_{left} \leftarrow$ **SortPermuationDivideAndConquer**$(\pi_1, ..., \pi_{\lfloor \frac{n}{2} \rfloor})$
9: $\varrho_{right} \leftarrow$ **SortPermuationDivideAndConquer**$(\pi_{\lfloor \frac{n}{2} \rfloor + 1}, ..., \pi_n)$
10: **return** $\varrho \parallel \varrho_{left} \parallel \varrho_{right}$
11: **fim se**

$k \geq 0$. Quando $k = 1$ eles garantiram um algoritmo com fator de aproximação $O(\lg n)$. Quando $k \geq 2$, eles garantiram um algoritmo com fator de aproximação 2.

Quando a orientação dos genes é considerada, alguns dos resultados propostos por Bender *et al.* continuam valendo. Em particular, Swidan *et al.* [40] foi capaz de garantir a mesma de aproximação $O(\lg n)$ para o problema de ordenação por reversões ponderadas com sinais.

1.1.3 Propriedades de permutações

Ao longo deste trabalho são utilizadas várias propriedades de permutações, as quais são úteis para a construção de métricas e heurísticas e também para a prova de lemas e teoremas. Algumas propriedades mais específicas ou introduzidas neste trabalho são apresentadas nos próximos capítulos. As propriedades mais gerais são apresentadas a seguir.

Considerando uma permutação $\pi = (\pi_1 \; ... \; \pi_{n-2} \; \pi_{n-1} \; \pi_n)$ de tamanho n, a Definição 2 apresenta a posição de um elemento $p(\pi, i)$ e a Definição 3 apresenta a permutação inversa π^{-1}. Para permutações com sinal, nós definimos $s(\pi, i)$ (Definição 4) como o *Sinal* do elemento i na permutação π.

Definição 2. Posição*:* $p(\pi, i) = k \Leftrightarrow |\pi_k| = i$, $p(\pi, i) \in \{1, 2, \ldots, n\}$.

Definição 3. Permutação Inversa*: dada uma permutação π de tamanho n, a permutação inversa π^{-1} também possui n elementos tal que $\pi_i^{-1} = p(\pi, i)$, conforme mostrado abaixo:*

$$\pi^{-1} = (p(\pi, 1) \; p(\pi, 2) \; ... \; p(\pi, n-1) \; p(\pi, n))$$

Definição 4. Sinal*:* $s(\pi, i) = \begin{cases} 1, & Se \; \pi_i < 0 \\ 0, & Se \; \pi_i > 0 \end{cases}$

Em seguida nós definimos duas propriedades clássicas de problemas de rearranjo de genomas: *breakpoints* e *strips*. Um *breakpoint* determina uma descontinuidade entre elementos de uma permutação π, enquanto uma *strip* (Definição 7) determina um bloco de elementos entre *breakpoints*.

As definições de *breakpoints* entre permutações com sinal e sem sinal são ligeiramente diferentes, como mostram as Definições 5 e 6, entretanto, o número de *breakpoints* possui

a mesma representação $b(\pi)$ para ambos os casos. Já para *strips*, nós apresentamos a Definição 7. Os Exemplos 1 e 2 mostram *breakpoints* de uma permutação sem sinal e com sinal, respectivamente.

Definição 5. *Número de breakpoints para permutações* sem sinal*: Considerando a* permutação estendida *de π, em que são inseridos os novos elementos $\pi_0 = 0$ e $\pi_{n+1} = n+1$. Então o número de breakpoints de π é definido pela seguinte formulação:*

$$b(\pi) = \sum_{i=0}^{|\pi|} \begin{cases} 1, & se\ |\pi_{i+1} - \pi_i| \neq 1. \\ 0, & caso\ contrário. \end{cases}$$

Definição 6. *Número de breakpoints para permutações* com sinal*: Considerando a* permutação estendida *de π, em que são inseridos os novos elementos $\pi_0 = 0$ e $\pi_{n+1} = n+1$. Então o número de breakpoints de π é definido pela seguinte formulação:*

$$b(\pi) = \sum_{i=0}^{|\pi|} \begin{cases} 1, & se\ \pi_{i+1} - \pi_i \neq 1. \\ 0, & caso\ contrário. \end{cases}$$

Definição 7. *Seja π uma permutação sem sinal, uma strip de π é um intervalo maximal $[\pi_i, \ldots \pi_j]$ sem nenhum breakpoint interno tal que (π_{i-1}, π_i) e (π_j, π_{j+1}) são breakpoints.*

Exemplo 1. *Considerando a permutação sem sinal $\pi = (2\ 1\ 5\ 4\ 3\ 6\ 7)$, nós estendemos π e identificamos os breakpoints da forma a seguir:*

$$(0 \bullet 2\,1 \bullet 5\,4\,3 \bullet 6\,7\,8)$$

Neste caso, $b(\pi) = 3$.

Exemplo 2. *Considerando a permutação com sinal $\pi = (+2\ +1\ -5\ -4\ -3\ +6\ +7)$, nós estendemos π e idetificamos os breakpoints da forma a seguir:*

$$(0 \bullet +2 \bullet +1 \bullet -5\ -4\ -3 \bullet +6\ +7\ +8)$$

Neste caso, $b(\pi) = 4$.

Lema 1. *$b(\pi) = 0$, se e somente se $\pi = \iota$.*

Demonstração. A prova deste lema é direta pela própria definição. Assumindo que π é uma permutação estendida, para $\pi = \iota$, temos que $\pi_{i+1} = \pi_i + 1$, para $0 \leq i \leq |\pi|$. Desta forma, não existe um índice i tal que $|\pi_{i+1} - \pi_i| \neq 1$, o que resulta em $b(\pi) = 0$.

Para $\pi \neq \iota$, denotamos por a como o menor índice i tal que $\pi_i \neq i$, para $1 \leq i \leq |\pi|$. Neste caso, $|\pi_a - \pi_{a-1}| \neq 1$. Deste modo, $b(\pi) > 0$. □

O nível de entropia $ent(\pi)$, Definição 8, representa a soma das distâncias de cada elemento para a respectiva posição correta, o que equivale a $|\rho_j| - 1$, onde ρ_j é a reversão que posiciona o elemento j na posição correta π_j. Conforme o Lema 2 e a respectiva prova, vemos que a entropia somente tem valor zero para a permutação identidade ι.

Para permutações com sinal, nós temos uma outra formulação para a entropia, que considera adicionalmente os sinais da permutação, pois uma permutação com todos os

elementos na posição correta, mas com alguns elementos com sinal negativo ainda não está ordenada.

Definição 8. Entropia *para permutações* ***sem sinal*** *é definida pela formulação abaixo, onde* π^{-1} *é a permutação inversa (Definição 3).*

$$ent(\pi) = \sum_{i=1}^{|\pi|} |i - \pi_i^{-1}|$$

Lema 2. $ent(\pi) = 0$*, se e somente se* $\pi = \iota$*.*

Demonstração. Para $\pi = \iota$, temos que $\pi_i^{-1} = i, 1 \leq i \leq |\pi|$. Portando cada termo $|i - \pi_i^{-1}|$ na Definição 8 é igual a zero. Para $\pi \neq \iota$, há pelo menos dois elementos tais que $|i - \pi_i^{-1}| > 0$. Como nenhum dos termos pode produzir elementos negativos, temos que $\sum_{i=1}^{|\pi|} |i - \pi_i^{-1}| > 0$ para $\pi \neq \iota$. □

Definição 9. Entropia *para permutações* ***com sinal*** *é definida pela formulação abaixo.*
$ent(\pi) = \sum_{i=1}^{n} |i - \pi_i^{-1}| + \sum_{i=1}^{n} s(\pi, i)$

Para a ordenação de permutações com sinal, Hannenhalli e Pevzner [29] desenvolveram um algoritmo polinomial ótimo que utiliza o conceito de reversões seguras. Reversões seguras para uma permutação π são aquelas que levam a uma solução ótima, ou seja, se a cada etapa for aplicada uma reversão segura será obtida uma ordenação ótima.

Posteriormente, Bergeron [11] apresentou um método simplificado para identificar um conjunto de reversões seguras para uma permutação π. Tais reversões são obtidas a partir de pares de elementos que são **orientados**, conforme a Definição 10. O número de pares orientados, em uma permutação π, $nop(\pi)$, é apresentado na Definição 11.

Definição 10. *Um par orientado* (π_i, π_j) *é um par de elementos de* π *tal que* $||\pi_i| - |\pi_j|| = 1$ *e* $\pi_i \times \pi_j < 0$*.*

Definição 11. *É denotado por* $nop(\pi)$ *o número de pares orientados de uma permutação* π*, conforme a Definição 10.*

1.2 GRASP

Nesta seção, nós apresentamos uma revisão bibliográfica da meta-heurística *Greedy Randomized Adaptive Search Procedure* (GRASP) introduzida por Feo e Resende [22], a qual já foi utilizada para encontrar soluções para outros problemas de otimização combinatória [1, 14, 20, 23, 35]. O leitor é aconselhado a revisar o trabalho de Festa e Resende [24] para uma explicação mais detalhada sobre GRASP.

De forma genérica, um problema de otimização combinatória pode ser definido por um conjunto base de elementos $E = \{1, ..., n\}$, um conjunto de soluções viáveis $F \subseteq 2^E$ e uma função objetiva $f : 2^E \longrightarrow \mathbb{R}$. Considerando que se trata de um problema de minimização, nós procuramos pela solução ótima $s^* \in F$ tal que $f(s^*) \leq f(s), \forall s \in F$.

Nós podemos utilizar como exemplo o clássico problema do caixeiro viajante, onde o conjunto base E contém todas as arestas que conectam cidades, a função $f(s)$ representa o custo de todas as arestas em s, e F é formado por todos os subconjuntos de arestas que determinam um ciclo *Hamiltoniano*.

Já para o problema em que se busca ordenar uma permutação π por reversões, o conjunto base E é o conjunto formado por reversões ρ que transformam uma permutação π em uma permutação α, tais reversões podem ser vistas como arestas no grafo G_n, o qual é um grafo que contém todas as $n!$ permutações de tamanho n para permutações sem sinal ou $2^n n!$ para permutações com sinal. A função $f(s)$ é definida como a soma dos custos das reversões em s, e F é formado por todos os subconjunto de reversões que definem um caminho de π para ι no grafo G_n.

Em essência, *GRASP* é um modelo de meta-heurística iterativa que procura por boas soluções s em um determinado espaço de busca. Nós destacamos que este modelo é caracterizado por realizar escolhas aleatórias em uma distribuição de probabilidade definida por uma função gulosa.

Neste trabalho, nosso método tem como base o modelo genérico de *GRASP* proposto por Feo e Resende [22]. O Algoritmo 2 descreve este modelo, onde é dada como entrada uma solução inicial *semente* e um número de iterações $Max_Iteracoes$. A cada iteração uma nova solução é construída a partir da solução *semente*, utilizando para isto o algoritmo guloso-aleatório $Construcao_Gulosa_Randomizada$. Em seguida, é empregado um algoritmo de busca local na solução gerada. Em alguns casos, a solução gerada pode não ser viável, o que torna necessário executar a reparação da solução. Caso esta nova solução supere a melhor solução corrente ($Melhor_Solucao$), esta passa a ser adotada como a melhor solução.

O procedimento $Construcao_Gulosa_Randomizada$, utilizado para a construção de soluções no Algoritmo 2, é mostrado no Algoritmo 3. Este algoritmo realiza a ordenação de elementos candidatos a cada iteração e adiciona uma determinada quantidade dos elementos melhores classificados à lista restrita de elementos candidatos *RCL* (*Restricted Elements List*), em seguida um elemento é escolhido aleatoriamente da *RCL* para compor a solução corrente. Como Feo e Resende [22] destacaram, este algoritmo possui as seguintes 3 características:

- **Guloso**: Constrói a lista *RCL* utilizando os melhores elementos.
- **Aleatório**: Escolhe aleatoriamente um elemento da *RCL*.
- **Adaptativo**: Reconstrói a lista *RCL* a cada iteração.

O procedimento de busca local utilizado no Algoritmo 2 é mostrado no Algoritmo 4. Segundo Feo e Resende [22], a qualidade e a rapidez de execução de uma meta-heurística *GRASP* depende de vários aspectos, dentre eles estão a estrutura da vizinhança, a técnica utilizada para explorar a vizinhança, a estratégia para avaliar a função de custo das soluções vizinhas e a própria solução inicial. Duas estratégias usuais para realizar buscas na vizinhança são *best-improving* e *first-improving*. Para a estratégia *best-improving*, todos as soluções na vizinhança são exploradas, e a solução corrente é substituída pela melhor

delas. No caso da estratégia *first-improving*, a solução corrente é movida para o primeiro vizinho que possui valor menor do que a solução corrente.

Novamente segundo Feo e Resende [22], foi observado na prática que em muitas aplicações frequentemente as duas estratégias levam ao mesmo resultado, mas com menor tempo computacional quando a estratégia *first-improving* é utilizada. Também foi observado que é mais frequente ocorrer convergência prematura para um mínimo local ruim quando empregada a estratégia *best-improving*.

O algoritmo *roulette wheel mechanism* (Algoritmo 5) apresenta uma forma de escolher um elemento em um conjunto de forma aleatória, considerando que cada elemento possui uma probabilidade associada de ser escolhido (pontuação). Este algoritmo implementa de forma simples as características gulosas e aleatórias do *GRASP*. Nós utilizamos esta estratégia em diferentes partes deste trabalho.

Algoritmo 2 ClassicalGRASP

1: **Data:** $Max_Iteracoes$, $Semente$
2: $Melhor_Solucao \leftarrow +\infty$
3: **para** $k = 1, ..., Max_Iteracoes$ **faça**
4: $s \leftarrow Construcao_Gulosa_Randomizada(Semente)$
5: **se** s não é viável **então**
6: $s \leftarrow Repair(s)$
7: **fim se**
8: $s \leftarrow Busca_Local(s)$
9: $Melhor_Solucao \leftarrow Atualizar_Solucoes(s, Melhor_Solucao)$
10: $Semente \leftarrow s$
11: **fim para**
12: **return** $Melhor_Solucao$

Algoritmo 3 $Construcao_Gulosa_Randomizada$

1: **Data:** $Semente$
2: $s \leftarrow \emptyset$
3: Inicializar o conjunto de elementos candidatos
4: Avaliar o custo incremental dos elementos candidatos
5: **enquanto** existir pelo menos um elemento na lista de candidatos **faça**
6: Construir a lista restrita de elementos candidatos (RCL)
7: Selecionar um elemento e de RCL aleatoriamente
8: $s \leftarrow s \cup \{e\}$
9: Atualizar o conjunto de elementos candidatos
10: Reavaliar os custos incrementais
11: **fim enquanto**
12: **return** s

1.3 Objetivos

Como mostramos na revisão bibliográfica, o modelo de ordenação por reversões ponderadas é mais adequado do que modelos anteriores para várias aplicações práticas. No

Algoritmo 4 *Busca_Local*

1: **Data:** s
2: **enquanto** s não é localmente ótima **faça**
3: Procure $s^* \in N(s)$ tal que $f(s^*) < f(s)$
4: $s \leftarrow s^*$
5: **fim enquanto**
6: **return** s

Algoritmo 5 selectByRouletteWheel

1: **Data:** $elementos$, $pontuacoes$
2: $soma_pontuacao \leftarrow soma(pontuacoes)$
3: $R \leftarrow \textbf{random}(1, soma_pontuacao)$
4: $k \leftarrow 1$
5: $SomaCorrente \leftarrow 0$
6: **enquanto** $SomaCorrente < R$ **faça**
7: $SomaCorrente \leftarrow SomaCorrente + pontuacoes[k]$
8: $k \leftarrow k + 1$
9: **fim enquanto**
10: **return** $elementos[k]$

entanto, esta linha de pesquisa encontra-se ainda pouco explorada, com poucos trabalhos disponíveis na literatura. Além disto, dentre todos esses trabalhos, não encontramos nenhum estudo experimental sobre algoritmos específicos para este problema. Desta forma, este trabalho tem os seguintes objetivos principais:

- Contribuir para expansão da pesquisa no problema, buscando superar o estado da arte corrente.
- Desenvolver algoritmos que forneçam boas soluções para o problema.
- Realizar estudos experimentais sobre estes algoritmos.
- Disponibilizar ao público uma ferramenta de código aberto, que contempla os estudos deste trabalho.

1.4 Organização da Dissertação

O conteúdo desta dissertação está dividido em relação à abordagem algorítmica que foi empregada. O Capítulo 2 trata do algoritmo paramétrico desenvolvido para permutações sem sinal. Já o Capítulo 3 trata da meta-heurística *GRASP* desenvolvida tanto para permutações com sinal quanto sem sinal. Por fim, são apresentadas as conclusões no Capítulo 4.

Capítulo 2

Algoritmo Heurístico Paramétrico

Neste capítulo, nós apresentamos o método heurístico que desenvolvemos para o problema de Distância de Ordenação por Reversões Ponderadas para permutações sem sinal, como já mencionamos no capítulo de introdução.

Primeiramente apresentamos os elementos que utilizamos como base para elaborar nossa heurística paramétrica. Em seguida, apresentamos o algoritmo guloso que utiliza essa heurística como função objetivo. Também apresentamos um algoritmo para obter soluções exatas para o problema.

Na seção de resultados, nós exploramos as características do nosso método em diversos cenários. E, em uma análise comparativa, mostramos resultados que superam trabalhos previamente disponíveis na literatura.

Por fim, nós apresentamos a publicação resultante do trabalho apresentado neste capítulo.

2.1 Heurística paramétrica

Nesta seção,k primeiramente iremos apresentar 3 métricas que podem ser utilizadas como função objetivo de um algoritmo heurístico guloso para a ordenação de permutações por reversões. Estas métricas são: nível de entropia, número de *breakpoints* e número de inversões.

Conforme vimos no Capítulo 1, o nível de entropia $ent(\pi)$ (Definição 8) representa a soma das distâncias de cada elemento para a respectiva posição correta, o que equivale a $|\rho_j| - 1$, onde ρ_j é a reversão que posiciona o elemento j na posição correta π_j. Conforme o Lema 2 e a sua respectiva prova, vimos que a entropia somente tem valor zero para a permutação identidade ι.

Também no Capítulo 1, vimos que o número de *breakpoints* $b(\pi)$ representa a quantidade de descontinuidades em uma permutação, conforme mostrado na Definição 5. Número de *breakpoints* tem sido uma métrica importante em muitos trabalhos para ordenação por reversões com custos unitários, com resultados obtidos mostrando proporcionalidade entre $d(\pi)$ e $b(\pi)$. O Lema 1 mostra que somente a permutação identidade tem zero *breakpoints*.

O número de inversões $inv(\pi)$ representa a quantidade de pares de elementos na permutação π que estão relativamente invertidos, conforme formulação da Definição 12. O Lema 3 mostra que somente a permutação identidade não possui pares de elementos invertidos.

Definição 12. O Número de inversões *é definido pela formulação abaixo:*

$$inv(\pi) = \sum_{i=1}^{|\pi|} \sum_{j=i+1}^{|\pi|} \begin{cases} 1, & se\ \pi_i > \pi_j. \\ 0, & caso\ contrário. \end{cases}$$

Lema 3. $inv(\pi) = 0$*, se e somente se* $\pi = \iota$.

Demonstração. Para $\pi = \iota$, temos que $\pi_i < \pi_j$ para cada par i, j tal que $i < j$. Neste caso o valor da função $inv(\iota)$ é igual a soma de zeros.

Para $\pi \neq \iota$, denotamos a como o menor índice i tal que $\pi_i \neq i$, para $1 \leq i \leq |\pi|$. Neste caso $\pi_a > a$, pois todos os elementos menores que a estão posicionados à esquerda de a. Seja $\pi_b = a$, temos que $b > a$, caso contrário a não seria o menor dentre tal conjunto de indices. Deste modo, existe um par a, b tal que $a < b$ e $\pi_a > \pi_b$. Portanto, pelo menos um termo da Definição 12 terá valor 1. Como nenhum termo pode produzir elementos negativos, temos que $inv(\pi) > 0$ para $\pi \neq \iota$. □

A partir do estudo experimental destas métricas, desenvolvemos uma heurística que combina todas estas em uma só formulação. Esta heurística, denominada $hc(\pi)$, é apresentada na Definição 13 e, como pode ser observado, empregamos a soma ponderada das métricas. A entropia $ent(\pi)$ é ponderada pelo parâmetro A. Enquanto o número de *Breakpoints* $b(\pi)$ é ponderado pelo parâmetro B. E por fim, o número de inversões $inv(\pi)$ é ponderado pelo parâmetro C. Uma dada atribuição de valores a estes parâmetros é denominada aqui de *configuração de parâmetros*.

O Lema 4 mostra que somente a permutação identidade possui valor zero para a nossa heurística $hc(\pi)$, o que é um resultado direto das métricas usadas possuirem essa mesma propriedade.

A partir de $hc(\pi)$, nós formulamos uma outra heurística, desta vez para comparar quantitativamente reversões ρ aplicáveis a uma permutação π. Denominamos esta heurística por $\delta_{hc}(\pi, \rho)$, apresentada na Definição 14. Neste trabalho, o valor de $\delta_{hc}(\pi, \rho)$ representa o benefício da reversão ρ em relação à permutação π.

Notamos que a definição paramétrica de $hc(\pi)$ não é apenas uma heurística, mas de um método para a geração de heurísticas. Assim, cada configuração de parâmetros é uma heurística distinta, a qual atribui diferentes prioridades para cada métrica e, devido a isto, possui maior eficácia para algumas instâncias do problema e menor para outras (Detalhado na seção de Resultados deste caítulo).

Definição 13. *Função heurística para estimar o custo de ordenação de uma permutação* π *é mostrada abaixo, onde* A*,* B *e* C *são parâmetros de ponderação tal que* $A, B, C \in \mathbb{R}$ *e* $A, B, C \geq 0$.

$$hc(\pi) = A \times ent(\pi) + B \times b(\pi) + C \times inv(\pi)$$

Lema 4. $hc(\iota) = 0$*, se e somente se* $\pi = \iota$.

Demonstração. Este lema pode ser provado diretamente dos lemas 1, 2 e 3. □

Definição 14. *Seja ρ uma reversão aplicável à permutação π, o benefício de ρ, denotado por $\delta_{hc}(\pi, \rho)$, é definido pela seguinte formulação:*

$$\delta_{hc}(\pi, \rho) = \frac{hc(\pi) - hc(\pi \cdot \rho)}{|\rho|}$$

Consideramos importante destacar que, para uma dada permutação π, se tivermos uma sequência de reversões $\rho_1, \rho_2, \ldots \rho_k$, em que toda reversão ρ_i resulta em benefício positivo, tal sequência de reversões poderá ordenar π se for suficientemente grande, pois, indutivamente, verificamos que a cada reversão obtemos uma permutação mais próxima da identidade ι (conforme o Lema 4).

Contudo, foram identificadas algumas raras permutações, em algumas configurações de parâmetro, para as quais não existe reversão que resulte em benefício positivo, o que implica que algumas permutações não possuem uma sequência de reversões com benefício positivo, suficientemente grande para ordená-las. Por exemplo, assumindo que $A = B = C = 1$ em hc (Definição 13), todas as possíveis permutações π para $|\pi| < 8$ possuem pelo menos uma reversão com benefício positivo. Para $|\pi| = 8$, somente uma permutação não possui nenhuma reversão com benefício positivo (desconsiderando a permutação identidade ι). Para $|\pi| = 9$ e $|\pi| = 10$ este número aumenta para 3.

A única permutação de tamanho 8 que não possui reversão com benefício positivo quando $A = B = C = 1$ é a permutação $\pi = (8\ 2\ 3\ 4\ 5\ 6\ 7\ 1)$. Esta permutação está somente duas reversões de distância da permutação identidade, pois $\pi \cdot \rho(1, 8) \cdot \rho(2, 7) = \iota$. Contudo, a primeira reversão gera a permutação $\sigma = \pi \cdot \rho(1, 8) = (1\ 7\ 6\ 5\ 4\ 3\ 2\ 8)$, a qual possui um valor maior para a heurística hc, conforme é mostrado abaixo.

$\pi = (8\ 2\ 3\ 4\ 5\ 6\ 7\ 1)$	$\sigma = (1\ 7\ 6\ 5\ 4\ 3\ 2\ 8)$
$hc(\pi) = 31 \begin{cases} ent(\pi) = 14 \\ b(\pi) = 4 \\ inv(\pi) = 13 \end{cases}$	$hc(\sigma) = 35 \begin{cases} ent(\sigma) = 18 \\ b(\sigma) = 2 \\ inv(\sigma) = 15 \end{cases}$

Neste caso, o benefício de $\delta_{hc}(\pi, \sigma)$ é dado por

$$\delta_{hc}(\pi, \sigma) = \frac{hc(\pi) - hc(\sigma)}{f(\rho(1, 8))} = \frac{31 - 35}{8} = -0.5$$

É importante destacar que esta reversão poderia ter benefício positivo se atribuíssemos diferentes valores para A, B e C. Por exemplo, se definirmos $A = 1$, $B = 5$ e $C = 0$, então $hc(\pi) = 34$ e $hc(\sigma) = 28$. Neste caso, o benefício seria $\delta_{hc}(\pi, \sigma) = 0.75$.

Na subseção a seguir são apresentados os detalhes do algoritmo que desenvolvemos.

2.1.1 Análise do algoritmo

O algoritmo que desenvolvemos para ordenar permutações π é descrito no pseudocódigo do Algoritmo 6. Este algoritmo realiza, a cada iteração, uma escolha gulosa da próxima

reversão a ser aplicada, considerando como função objetivo $\delta_{hc}(\pi, \rho)$ gerada por uma dada configuração de parâmetros.

Algoritmo 6 greedyHeuristic

```
1: Data: π, A, B, C
2: ϱ ← ()
3: enquanto π ≠ ι faça
4:    ρ ← chooseReversalMaxBenefit(π, A, B, C)
5:    se ρ ≠ ∅ então
6:       ϱ ← ϱ || ρ
7:       π ← π · ρ
8:    else
9:       nbk ← b(π)
10:      enquanto b(π) = nbk faça
11:         ρ ← sortByReversionsTwoAprox(π)
12:         ϱ ← ϱ || ρ
13:         π ← π · ρ
14:      fim enquanto
15:   fim se
16: fim enquanto
17: return ϱ
```

Como destacamos anteriormente, nem sempre é possível obter uma reversão com benefício positivo. Devido a isso, o Algoritmo 6 realiza uma chamada para o Algoritmo 7 para lidar com esses casos. O Algoritmo 7 corresponde a uma etapa do algoritmo iterativo proposto por Kececioglu e Sankoff [31], que é 2-aproximado para o problema de ordenação por reversões com custos unitários. Conforme as definições 6 e 7, *breakpoints* dividem a permutação em *strips*, que são intervalos maximais sem *breakpoints*. Kececioglu e Sankoff mostram que se π tem uma *strip* decrescente, então há uma reversão que remove pelo menos um *breakpoint*. Além disso, toda reversão em uma permutação sem *strips* decrescentes cria uma permutação com uma *strip* decrescente. Portanto, dois cenários podem ocorrer quando nós executamos Algoritmo 7.

1. Se π tem uma *strip* decrescente, então as linhas 2–9 no Algoritmo 7 irão encontrar a reversão que reduz o número de *breakpoints*. Além disso, é escolhida a reversão ρ tal que $\pi \cdot \rho$ tem uma *strip* decrescente.

2. Se π não tem *strip* decrescente, então a linha 10 escolhe uma reversão ρ que corta dois *breakpoints*, então o número de *breakpoints* não muda. Além disso, a permutação $\pi \cdot \rho$ terá uma *strip* decrescente.

Algoritmo 7 sortByReversionsTwoAprox

```
1: Data: π
2: se π possui pelo menos uma strip decrescente então
3:    k ← o menor elemento em uma strip decrescente
4:    ρ ← a reversão que corta depois de k e depois de k − 1
5:    se π · ρ não possui strips decrescentes então
6:       l ← o maior elemento dentre todas as strips decrescentes
7:       ρ ← a reversão que corta antes de l e antes de l + 1
8:    fim se
9: else
10:   ρ ← a reversão que corta os dois primeiros breakpoints
11: fim se
12: return ρ
```

Observamos que nós temos que executar no máximo duas chamadas para o Algoritmo 7 para diminuir o número de *breakpoints*. Portanto, o laço de repetição nas linhas $9 - 12$ no Algoritmo 6 é executado no máximo duas vezes.

Algoritmo 8 chooseReversalMaxBenefit

```
1: Data: π, A, B, C
2: P ← Sequência de breakpoints, em ordem crescente, que marcam o início ou o final
   de uma strip (sem repetição) em π.
3: maxBeneficio ← 0
4: ρ* ← ∅
5: para i ← 1 to |P| faça
6:    para j ← i + 1 to |P| faça
7:       ρ ← (P[i], P[j])
8:       beneficio ← δ_hc(π, ρ, A, B, C)
9:       se beneficio > maxBeneficio então
10:         maxBeneficio ← beneficio
11:         ρ* ← ρ
12:      fim se
13:   fim para
14: fim para
15: return ρ*
```

O Algoritmo 6 possui complexidade $O(n^4 \lg^3 n)$, como mostra a Propriedade 5.

Propriedade 5. *O Algoritmo 6 possui complexidade* $O(n^4 \lg^3 n)$.

Demonstração.

- A cada iteração é necessário calcular cada uma das seguintes métricas para $O(n^2)$ reversões:
  - $ent(\pi)$, com custo $O(n)$

  - $inv(\pi)$, com custo $O(n \lg(n))$ (Algoritmo de divisão e conquista)
  - $b(\pi)$, com custo $O(n)$

  O que resulta em uma complexidade $O(n^3 \lg(n))$ por iteração.

- Considerando que o limite superior para o número de reversões é $O(n \lg(n)^2)$, segundo Bender et al [9], temos que a complexidade total do algoritmo é $O(n^4 \lg(n)^3)$.

□

2.2 Algoritmo para gerar soluções exatas

Seja $G_n = (V, E)$ um grafo ponderado nas arestas em que o conjunto de vértices V é formado por todas as permutações π de tamanho n e o conjunto de arestas E contém arestas $e_{ij} = (\pi, \sigma)$ se existe uma reversão ρ que transforma π em σ, tal que $w(e_{ij}) = |\rho|$. O caminho mais curto de π para σ em G_n é equivalente à sequência ótima de reversões ponderadas que transforma π em σ.

Para computar as sequências de ordenação de todas as permutações de tamanho n nós desenvolvemos o seguinte método:

- A distância de ordenação para todas permutações de tamanho n é pré-computada pelo Algoritmo 9, que é uma variação do clássico algoritmo de *Dijkstra* [21] para caminho mínimo em grafos. De forma que a permutação identidade ι é definida como o vértice inicial e iterativamente é computado a distância para todas as permutações de tamanho n.
- Em seguida, para construir a sequência de ordenação de uma permutação π, foi desenvolvido o Algoritmo 10, que utiliza os dados computados na etapa anterior.

O Algoritmo 9 possui complexidade de tempo $O(n!n^2 lg\, n)$, o que o torna proibitivo para tamanhos de permutações grandes. Em nossos testes este algoritmo mostrou-se adequado para permutações de tamanho $n \leq 10$.

Algoritmo 9 ExactSolution

Data: n
$parent \leftarrow []$
$Q \leftarrow \emptyset$ {Fila de prioridade}
Q.insert(ι, 0)
$D[\iota] \leftarrow 0$
$parent[\iota] \leftarrow \emptyset$
enquanto Q.size() > 0 **faça**
  $(\pi, d_\pi) \leftarrow Q$.removeMin()
  para $i \leftarrow 1$ **to** n **faça**
    para $j \leftarrow i + 1$ **to** n **faça**
      $\rho_{ij} \leftarrow \rho(i, j)$
      $\sigma \leftarrow \pi \cdot \rho_{ij}$
      $d_{\sigma*} \leftarrow D[\pi_i] + |\rho_{ij}|$
      se $d_{\sigma*} < D[\sigma]$ **então**
        $D[\sigma] \leftarrow d_{\sigma*}$
        $parent[\sigma] \leftarrow \pi$
        Q.insert(σ, $D[\sigma]$)
      fim se
    fim para
  fim para
fim enquanto
return $(D, parent)$

Algoritmo 10 BuildPath

Data: $parent, \pi$
$\varrho \leftarrow \emptyset$ {Sequência de reversões}
se $\pi = \iota$ **então**
  return ϱ
else
  $\sigma \leftarrow parent[\pi]$
  $\rho \leftarrow \{\rho' | \sigma = \pi \cdot \rho\}$
  $\varrho \leftarrow \varrho \parallel \rho$
  $\varrho* \leftarrow$ **BuildPath**$(parent, \sigma)$
  $\varrho \leftarrow \varrho \parallel \varrho*$
fim se
return ϱ

2.3 Resultados

Para avaliar o algoritmo proposto neste trabalho e outros algoritmos existentes na literatura, foram realizados testes em dois cenários. Primeiramente, foram realizados testes com todas as permutações de tamanho até 10, onde foi possível realizar comparações com

as distâncias exatas. Em seguida, foram realizados testes com uma amostra de permutações aleatórias de tamanho até 100. A análise dos resultados destes cenários é abordada nas seções 2.3.1 e 2.3.3, respectivamente.

2.3.1 Resultados para permutações pequenas

Para o conjunto de todas as permutações de tamanho até 10 (totalizando 4.037.913 permutações no total), a heurística apresentada neste trabalho foi testada em duas variações, $TEST_1$ e $TEST_2$, os resultados obtidos nestes testes são apresentados na Tabela 2.1, juntamente com resultados de outros métodos. Os métodos apresentados na Tabela 2.1 são descritos a seguir.

- ◇ **TEST$_1$**: para cada permutação é escolhida a ordenação de menor custo dentre as fornecidas por execuções da heurística com configurações de parâmetro em que $A \in [0,5]$, $B \in [0,5]$ e $C \in [0,5]$, onde A, B e C são números inteiros.
- ◇ **TEST$_2$**: para cada permutação é escolhida a ordenação de menor custo, dentre as fornecidas por execuções da heurística com configurações de parâmetro em que $A \in [0,10]$, $B \in [0,10]$ e $C = 0$, onde A, B e C são números inteiros.
- ◇ **OPT**: solução exata de ordenação por reversões ponderadas, fornecido pelo Algoritmo 9.
- ◇ **REV**: para cada permutação é usada a solução exata para a ordenação por reversões com custos unitários, ponderada pelos tamanhos das reversões usadas.
- ◇ ***Bender***: para cada permutação é calculado o resultado do algoritmo de aproximação $O(\lg n)$ para a ordenação por reversões ponderadas [10].
- ◇ ***2-aprox***: O Algoritmo 2-aproximado de *Kececioglu e Sankoff* [31] para a ordenação por reversões com custos unitários, ponderada pelo tamanho das reversões usadas.

Para cada método presente na Tabela 2.1 são avaliados as seguintes métricas:

- ◇ **C$_{avg}$**: Custo médio de ordenação.
- ◇ **C$_{max}$**: Custo máximo de ordenação.
- ◇ **R$_{avg}$**: Razão de aproximação média do algoritmo em relação às soluções exatas.
- ◇ **R$_{max}$**: Razão de aproximação máxima do algoritmo em relação às soluções exatas.
- ◇ **%EX**: Porcentagem de soluções exatas fornecidas pelo algoritmo.
- ◇ **NR$_{avg}$**: Número médio de reversões fornecido pelo algoritmo.
- ◇ **NR$_{max}$**: Número máximo de reversões fornecido pelo algoritmo.

A Tabela 2.1 mostra que a heurística deste trabalho, em suas duas variações, apresenta resultados superiores em relação aos demais métodos presentes na literatura e próximos aos resultados exatos. Isto é claramente evidenciado para $n = 10$, em que a heurística que considera a variação $TEST_1$ possui aproximação média 1.04, aproximação máxima 1.85 e 58.75% de soluções exatas.

Em relação às variações das heurísticas, a variação $TEST_2$ apresenta resultados inferiores à $TEST_1$, porém muito próximos a estes. Isto sugere que o parâmetro C da heurística possui menor relevância para o desempenho da heurística.

Propriedade 6. *O método REV fornece soluções ótimas para o problema de ordenação por reversões com custos unitários. Seja ϱ o conjunto de reversões fornecido por REV para ordenar uma permutação π de tamanho n, cada reversão em ϱ possui custo no máximo n. Como o número de reversões em ϱ não pode ser maior do que em uma solução exata para o problema de ordenação por reversões ponderadas, obtemos aproximação $O(n)$ quando considerando reversões ponderadas.*

Conforme a Propriedade 6, o método *REV* é uma aproximação $O(n)$ para o problema, enquanto o algoritmo *Bender* é uma aproximação $O(\lg n)$ [10]. Contudo, os resultados da Tabela 2.1 demonstram que, experimentalmente, o método *REV* fornece melhores resultados que *Bender*, para todos os aspectos avalidados.

n	Método	C_{avg}	C_{max}	R_{avg}	R_{max}	%EX	NR_{avg}	NR_{max}
	2-Approx	1.00	2	1.00	1.00	100.00	0.50	1
	Bender	1.00	2	1.00	1.00	100.00	0.50	1
2	REV	1.00	2	1.00	1.00	100.00	0.50	1
	$TEST_1$	1.00	2	1.00	1.00	100.00	0.50	1
	$TEST_2$	1.00	2	1.00	1.00	100.00	0.50	1
	OPT	1.00	2	1.00	1.00	100.00	0.50	1
	2-Approx	2.67	5	1.04	1.25	83.33	1.17	2
	Bender	2.67	5	1.05	1.25	83.33	1.17	2
3	REV	2.67	5	1.04	1.25	83.33	1.17	2
	$TEST_1$	2.50	4	1.00	1.00	100.00	1.17	2
	$TEST_2$	2.50	4	1.00	1.00	100.00	1.17	2
	OPT	2.50	4	1.00	1.00	100.00	1.17	2
	2-Approx	4.96	9	1.12	1.80	62.50	1.92	3
	Bender	4.83	9	1.10	1.80	66.67	2.00	4
4	REV	4.58	7	1.05	1.25	75.00	1.75	3
	$TEST_1$	4.33	6	1.00	1.00	100.00	1.83	3
	$TEST_2$	4.33	6	1.00	1.00	100.00	1.83	3
	OPT	4.33	6	1.00	1.00	100.00	1.83	3
	2-Approx	7.78	14	1.20	2.00	44.17	2.71	5
	Bender	7.47	13	1.15	2.00	50.83	2.87	5
5	REV	6.88	11	1.07	1.57	68.33	2.39	4
	$TEST_1$	6.43	9	1.00	1.13	97.50	2.49	4
	$TEST_2$	6.43	9	1.00	1.13	97.50	2.49	4
	OPT	6.41	9	1.00	1.00	100.00	2.49	4
	2-Approx	11.07	20	1.26	2.43	30.56	3.51	7
	Bender	10.58	19	1.21	2.14	36.11	3.83	7
6	REV	9.67	19	1.10	2.13	57.78	3.04	5
	$TEST_1$	8.80	13	1.01	1.20	92.92	3.18	5
	$TEST_2$	8.80	13	1.01	1.20	92.64	3.17	5
	OPT	8.71	12	1.00	1.00	100.00	3.15	5
	2-Approx	14.81	27	1.32	2.75	19.96	4.32	9
	Bender	13.93	24	1.24	2.38	24.70	4.88	9
7	REV	12.92	26	1.15	2.17	44.37	3.70	6
	$TEST_1$	11.41	17	1.02	1.78	86.67	3.90	7
	$TEST_2$	11.42	17	1.02	1.78	86.59	3.88	7
	OPT	11.23	15	1.00	1.00	100.00	3.84	6
	2-Approx	19.01	35	1.36	3.10	12.31	5.14	12
	Bender	17.87	32	1.28	2.56	14.97	6.00	12
8	REV	16.59	32	1.19	2.50	31.73	4.38	7
	$TEST_1$	14.28	22	1.02	1.78	78.98	4.64	9
	$TEST_2$	14.29	23	1.02	1.78	78.45	4.63	9
	OPT	13.95	19	1.00	1.00	100.00	4.55	7
	2-Approx	23.67	44	1.40	3.45	7.16	5.98	15
	Bender	21.86	37	1.30	2.60	9.22	7.09	13
9	REV	21.00	43	1.24	2.80	19.90	5.06	8
	$TEST_1$	17.39	26	1.03	1.78	69.49	5.42	10
	$TEST_2$	17.41	28	1.03	1.78	69.04	5.41	11
	OPT	16.86	23	1.00	1.00	100.00	5.27	8
	2-Approx	28.78	54	1.44	3.77	3.94	6.83	18
	Bender	26.60	46	1.33	2.91	5.01	8.23	15
10	REV	25.58	54	1.28	2.83	13.33	5.76	9
	$TEST_1$	20.76	33	1.04	1.85	58.75	6.22	12
	$TEST_2$	20.78	35	1.04	1.85	58.62	6.21	13
	OPT	19.95	27	1.00	1.00	100.00	6.02	9

Tabela 2.1: Resultados para todas as permutações pequenas ($2 \geq n \leq 10$). A tabela é dividida em 9 partes e cada parte corresponde ao conjunto de todas as permutações de um tamanho n. Observamos que ambos os métodos $\texttt{TEST}_1$ e $\texttt{TEST}_2$ superam os demais. Eles fornecem, em geral, uma porcentagem maior de soluções exatas (%EX), e os custos médios de ordenação são menores do que dos outros métodos.

2.3.2 Análise de configurações de parâmetros

Esta seção trata do estudo detalhado do comportamento do Algoritmo 6 em relação a diferentes configurações de parâmetros. Desta forma, são adotadas as Definições 15 e 16. Seguindo estas definições, a configuração de parâmetro $(2, 3, 4)$ é equivalente a $(4, 6, 8)$.

Definição 15. *Uma configuração de parâmetro em que* $A = a$*,* $B = b$ *e* $C = c$ *pode ser representada como uma tripla* (a, b, c)*. De forma semelhante, uma configuração de parâmetro em que* $A = a$*,* $B = b$ *e* $C = 0$ *pode ser representada como um par* (a, b)*.*

Definição 16. *Uma configuração de parâmetro* (a, b, c) *é equivalente a uma outra configuração de parâmetro* (a_1, b_1, c_1) *se existe uma constante* k*,* $k \in \mathbb{R}+$*, tal que* $a = a_1 \times k$*,* $b = b_1 \times k$ *e* $c = c_1 \times k$*. De forma semelhante, uma configuração de parâmetros* (a, b) *tal que* $a \neq 1$ *pode ser transformada em uma configuração equivalente* $(1, \frac{b}{a})$*.*

Considerando a Definição 15, é possível otimizar os experimentos, pois podemos evitar a repetição de configurações de parâmetros equivalentes. De forma que para executar os experimentos $TEST_1$ ($A, B, C \in [0..5]$) seria necessário $6 \times 6 \times 6 = 216$ execuções do Algoritmo 6. Contudo, o número de execuções foi reduzido para 176, pois removemos configurações equivalentes. De forma similar, para o grupo de experimentos $TEST_2$ ($A, B \in [0..10]$ e $C = 0$) seria requerido $11 \times 11 \times 1 = 121$ execuções, removendo configurações equivalentes este número é reduzido para 66.

A Tabela 2.2 mostra as 50 melhores configurações de parâmetros do experimento $TEST_1$, ordenadas em relação à razão de aproximação média, para permutações de tamanho 10 (configurações de parâmetros equivalentes não são mostradas). É notável que os melhores resultados ocorrem com valores menores do parâmetro C, particularmente $C = 0$, isto fortalece a intuição de que este parâmetro possui menor impacto na qualidade dos resultados fornecidos pelo Algoritmo 6.

Por esta razão, nós criamos a segunda rodada de experimentos $TEST_2$, na qual o intervalo dos parâmetros A e B é aumentado para $[0..10]$, enquanto o valor do parâmetro C é fixado em 0.

(A,B,C)	C_{avg}	C_{max}	R_{avg}	R_{max}	%EX	N_{avg}	N_{max}
(1,5,0)	22.37	43	1.12	2.28	35.60	6.45	15
(3,5,0)	22.39	50	1.12	2.67	31.58	6.78	15
(2,3,0)	22.41	50	1.12	2.67	31.24	6.79	15
(1,4,0)	22.44	51	1.12	2.67	35.09	6.56	16
(3,4,0)	22.45	50	1.12	2.81	30.75	6.81	15
(4,5,0)	22.47	48	1.12	2.81	30.42	6.82	15
(3,5,1)	22.48	41	1.13	2.36	29.65	6.89	15
(1,2,0)	22.49	49	1.13	2.67	31.93	6.68	16
(5,4,0)	22.49	50	1.13	3.07	28.75	6.83	14
(1,1,0)	22.49	50	1.13	2.81	29.50	6.83	15
(2,5,0)	22.51	52	1.13	2.76	33.31	6.60	16
(1,3,0)	22.51	52	1.13	2.67	34.03	6.60	16
(4,3,0)	22.52	50	1.13	3.07	28.28	6.85	14
(5,5,1)	22.53	41	1.13	2.54	28.09	6.92	16
(2,5,1)	22.55	44	1.13	2.36	30.26	6.99	17
(4,4,1)	22.55	41	1.13	2.54	27.73	6.93	16
(2,4,1)	22.56	42	1.13	2.36	29.45	6.98	17
(4,5,1)	22.56	41	1.13	2.54	28.27	6.92	15
(3,2,0)	22.58	50	1.13	3.07	27.40	6.87	14
(3,4,1)	22.59	41	1.13	2.54	27.93	6.94	15
(2,3,1)	22.61	41	1.13	2.54	27.52	6.97	15
(5,4,1)	22.62	41	1.13	2.54	26.51	6.95	15
(3,3,1)	22.62	41	1.13	2.54	26.71	6.98	16
(3,5,2)	22.63	42	1.13	2.50	27.29	7.00	16
(5,3,0)	22.64	50	1.13	3.07	26.66	6.90	14
(4,5,2)	22.66	42	1.13	2.46	26.61	7.01	15
(4,3,1)	22.66	41	1.13	2.54	25.89	6.97	15
(5,5,2)	22.67	42	1.14	2.46	26.10	7.02	16
(1,3,1)	22.68	42	1.14	2.36	28.69	7.10	17
(2,1,0)	22.70	50	1.14	3.07	25.72	6.93	14
(5,3,1)	22.71	40	1.14	2.54	24.82	6.98	14
(3,4,2)	22.74	42	1.14	2.54	25.56	7.07	16
(2,5,2)	22.74	42	1.14	2.50	27.62	7.13	17
(2,2,1)	22.75	42	1.14	2.54	25.07	7.07	16
(5,4,2)	22.75	43	1.14	2.54	24.62	7.05	16
(5,2,0)	22.77	50	1.14	3.07	24.76	6.96	14
(1,5,1)	22.78	44	1.14	2.45	31.29	6.73	16
(3,2,1)	22.79	43	1.14	2.54	24.15	7.06	16
(1,4,1)	22.79	44	1.14	2.45	30.09	6.93	17
(4,5,3)	22.80	42	1.14	2.54	24.60	7.13	16
(5,5,3)	22.81	42	1.14	2.54	24.29	7.11	16
(4,2,1)	22.82	43	1.14	2.54	23.57	7.05	16
(1,2,1)	22.83	42	1.14	2.54	25.93	7.16	17
(3,1,0)	22.83	50	1.14	3.07	24.04	6.99	14
(3,5,3)	22.84	42	1.14	2.54	24.58	7.13	16
(3,3,2)	22.85	41	1.14	2.54	23.77	7.13	16
(4,3,2)	22.85	42	1.14	2.54	23.33	7.11	16
(2,3,2)	22.86	42	1.14	2.54	24.16	7.15	16
(3,4,3)	22.86	41	1.15	2.54	23.87	7.17	17
(5,3,2)	22.87	42	1.15	2.54	23.05	7.11	16

Tabela 2.2: As 50 melhores configurações de parâmetro (A, B, C) da heurística em relação à aproximação média para todas as pemutações de tamanho $n = 10$, o que totaliza 10! permutações. A tabela mostra que os melhores resultados possuem baixo valor para o parâmetro C. Em especial $C = 0$ ocorre frequentemente dentre estas configurações.

Para estudar o impacto do parâmetro B na qualidade das soluções, nós produzimos dois gráficos. O gráfico da Figura 2.1 mostra a porcentagem de soluções exatas produzida por cada par equivalente $(1, \frac{b}{a})$, tal que $a \in [1..10]$ e $b \in [0..10]$. O gráfico da Figura 2.2 mostra a taxa de aproximação média para este mesmo conjunto de pares. Em cada figura é desenhada uma curva para n no intervalo $[1..10]$, onde n é o número de elementos da permutação.

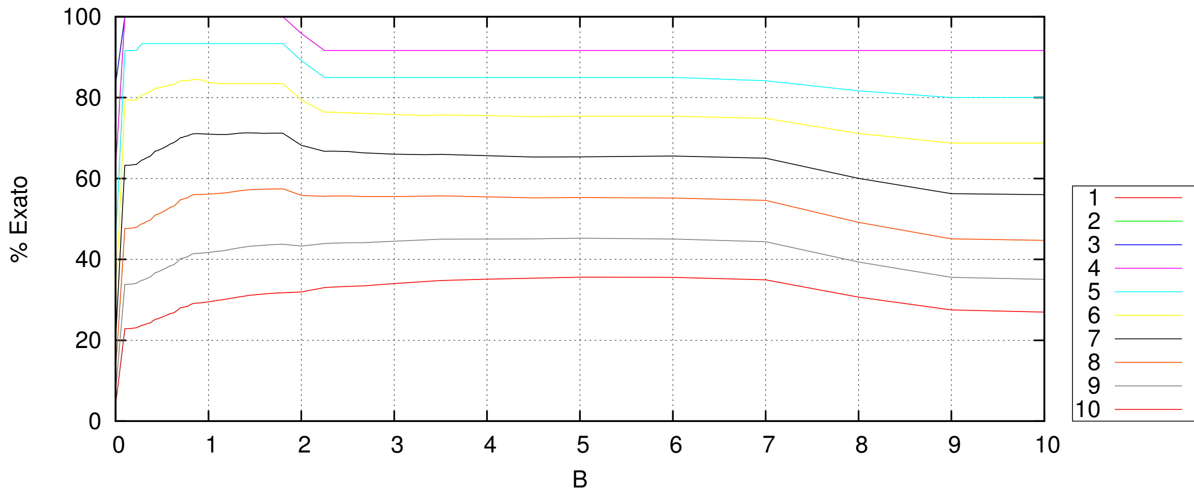

Figura 2.1: Este gráfico mostra a porcentagem de soluções exatas produzidos pela configuração de parâmetro $(1, B, 0)$, onde B é o valor dado no eixo-X. Nós observamos que para $n \leq 7$ os melhores resultados são encontrados quando definimos B entre 1 e 2. Por outro lado, para $n = 8$ e $n = 9$, escolhas de B no intervalo $[1, 7]$ levam a aproximadamente a mesma porcentagem de soluções exatas. Para $n = 10$, podem ser encontradas mais soluções exatas fixando B entre 5 e 6.

Para $n \leq 7$, são encontrados os melhores resultados quando o valor de B está entre 1 e 2. Como mostra a Figura 2.1, definindo parâmetros seguindo esta regra leva a uma porcentagem maior de soluções exatas. Esta regra também leva a menor fator de aproximação médio, o que é notado na Figura 2.2.

A preferência pela região entre 1 e 2 diminui conforme o tamanho de n aumenta. Para $n = 8$ e $n = 9$, para qualquer escolha de valor para o parâmetro B no intervalo $[1, 7]$ obtemos aproximadamente a mesma porcentagem de soluções exatas. Contudo, fixando o valor de B próximo a 1.8 leva a soluções com melhor taxa de aproximação média, pois a Figura 2.2 mostra que este é o mínimo global em ambos os casos.

Para $n = 10$, vemos um padrão diferente dos casos anteriores. Na Figura 2.1 observamos que o valor de B entre 4 e 6 resulta em 35.6% de soluções exatas, o que é uma taxa de soluções exatas um pouco maior do que o encontrado quando o valor do parâmetro B é definido entre 1 e 2.

Dentre todos estes casos, observamos que atribuindo 0 para B leva aos piores resultados, o que evidencia a importância da utilização *breakpoints* na nossa heurística.

Nas figuras 2.1 e 2.2 são mostrados resultados considerando todas as combinações não equivalentes dos parâmetros A e B no intervalo $[1..10]$. Contudo, não esperamos que sejam executadas todas estas configurações de parâmetro para obter um bom resultado,

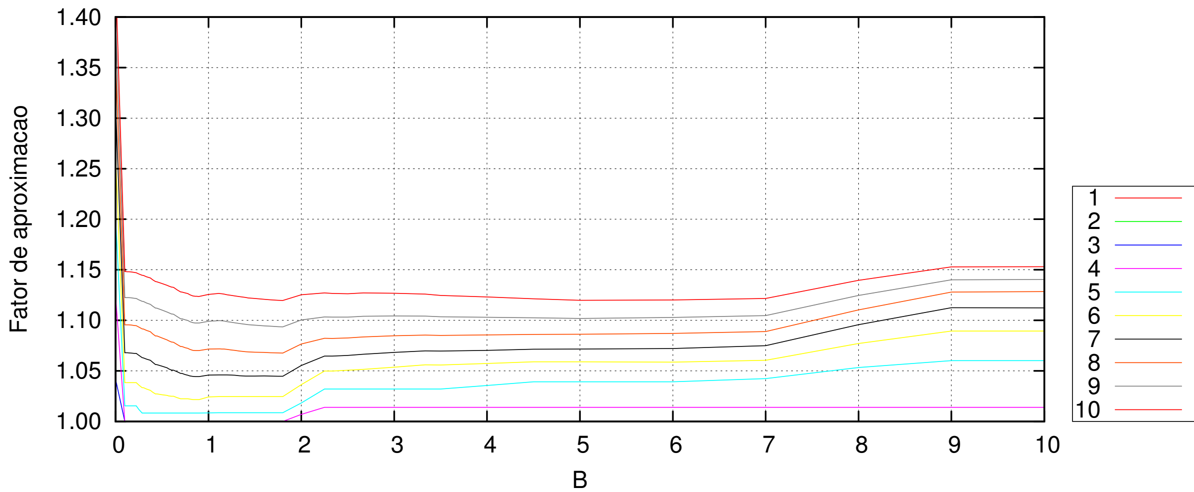

Figura 2.2: Este gráfico mostra a taxa de aproximação média produzida pela configuração de parâmetro $(1, B, 0)$, onde B é o valor dado no eixo-X. É observado que para $n \leq 7$ este gráfico mostra concordância com a Figura 2.1, e a melhor configuração de parâmetro tem o valor de B entre 1 e 2. Em todos estes casos, nós concluímos que os piores resultados são obtidos quando o valor de B é maior do que 7. Nós também notamos que $B = 0$ produz resultados ruins.

pois isso seria certamente computacionalmente caro. A Figura 2.3 mostra o gráfico de melhoramento obtido executando um certo número de configurações aleatórias de parâmetros e escolhendo o resultado de custo mínimo dentre os gerados. Como as configurações são escolhidas aleatoriamente, cada execução pode fornecer um valor diferente. Por esta razão, o mesmo número de configurações é executado 10 vezes e a média destes resultados é mostrada no gráfico. Consideramos este gráfico útil para escolher um número de configurações de acordo com a qualidade desejada para as soluções.

Analisando as curvas relativas ao número de configurações empregadas, notamos que usar 10 configurações apresenta uma boa relação de custo-benefício entre qualidade de solução e tempo de execução, pois estes resultados são superiores a quando empregamos apenas 1 e não obtemos resultados substancialmente melhores quando empregamos mais de 10 configuração de parâmetros. Esta análise fornece algumas dicas sobre quantas configurações de parâmetros podem ser utilizadas para se obter bons resultados, mas neste ponto não pretendemos definir critérios definitivos para a quantidade de números de configurações, cabendo esta decisão ao usuário.

A Figura 2.3 também apresenta a curva `Bender`, que corresponde ao algoritmo de aproximação $O(\lg n)$. Notamos que o nosso método supera este algoritmo até mesmo quando somente uma configuração de parâmetros é escolhida aleatoriamente.

A curva `2-Approx` na Figura 2.3 refere-se ao Algoritmo 2-aproximato de *Kececioglu e Sankoff* [31] para o problema de Ordenação por reversões com custos unitários. Dado que o algoritmo `2-Approx` é novamente inferior aos outros, concluímos que a melhoria na performance deve-se ao emprego da função $hc(\pi)$ no Algoritmo 6.

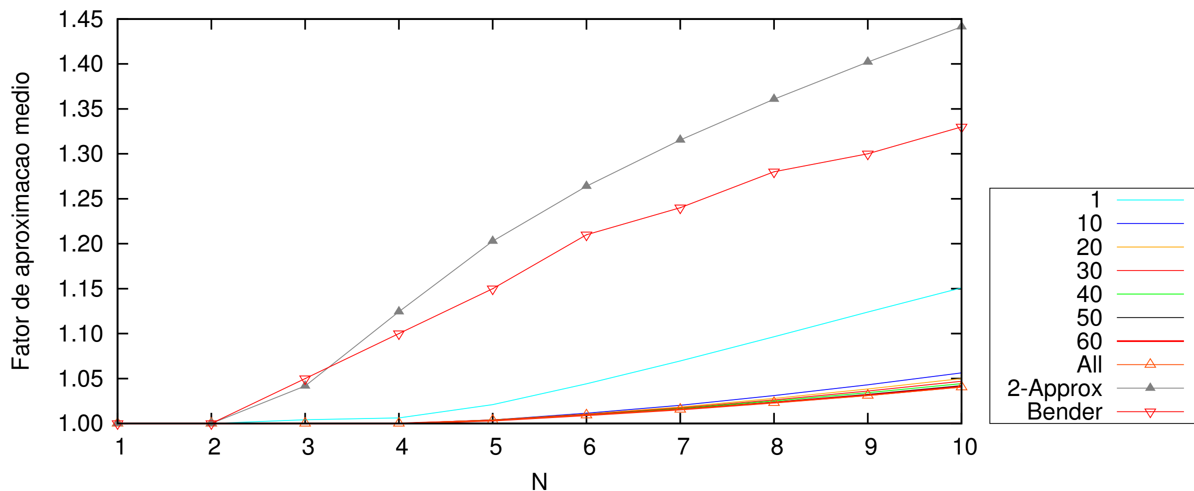

Figura 2.3: Este gráfico mostra a melhoria obtida executando nossa heurística várias vezes em permutações pequenas. Cada curva no gráfico mostra a taxa de aproximação média, quando consideramos um dado número de configurações de parâmetros, escolhidas aleatoriamente. Também é mostrada a curva `Bender`, que corresponde ao algoritmo de aproximação $O(\lg n)$ [10]. Analisando este gráfico nós verificamos que, quando são escolhidas aleatoriamente 10 configurações de parâmetros, é obtida uma boa relação de custo-benefício entre qualidade da solução e tempo de execução, sendo que é obtido uma taxa de aproximação significativamente melhor do que quando usamos somente uma configuração de parâmetros. Também notamos que quando empregado mais do que 10 configurações de parâmetros os melhoramentos passam a ser menos substanciais.

2.3.3 Resultados para permutações grandes

Na Seção 2.3.1 foi apresentado o custo mínimo, e consequentemente a razão de aproximação, e o número de soluções exatas fornecida por cada algoritmo para permutações pequenas ($n \leq 10$). Contudo, para permutações grandes o tempo computacional para calcular o custo mínimo é proibitivo (devido à complexidade fatorial do Algoritmo 9). Desta forma, algumas análises feitas na seção anterior não podem ser replicadas para permutações grandes. Por exemplo, não é possível afirmar a porcentagem de soluções exatas nos resultados fornecidos pelo Algoritmo 6.

Para permutações grandes, nós adotamos a comparação dos métodos em relação ao custo médio. Neste caso não é possível afirmar o quão longe cada solução está da solução exata, mas é possível identificar qual algoritmo apresenta a melhor performance.

Na Figura 2.4 é apresentado o custo médio para uma amostra de permutações grandes cujos tamanhos estão no conjunto $\{10, 15, 20, \ldots, 100\}$. Foram geradas 1000 permutações aleatórias para cada tamanho. Foram adotadas permutações de tamanho até 100 devido ao tempo de execução, pois, em média, é necessário cerca de 45 segundos para ordenar uma única permutação de tamanho 100. Como cada permutação deve ser ordenada para várias configurações de parâmetro diferentes, ordenar muitas permutações com mais do que 100 elementos seria proibitivo. Por exemplo, em média, para ordenar uma permutação com 110 elementos (usando o Algoritmo 6) são necessários 94 segundos, para 130 elementos são 214 segundos e para 150 elementos são necessários 5 minutos.

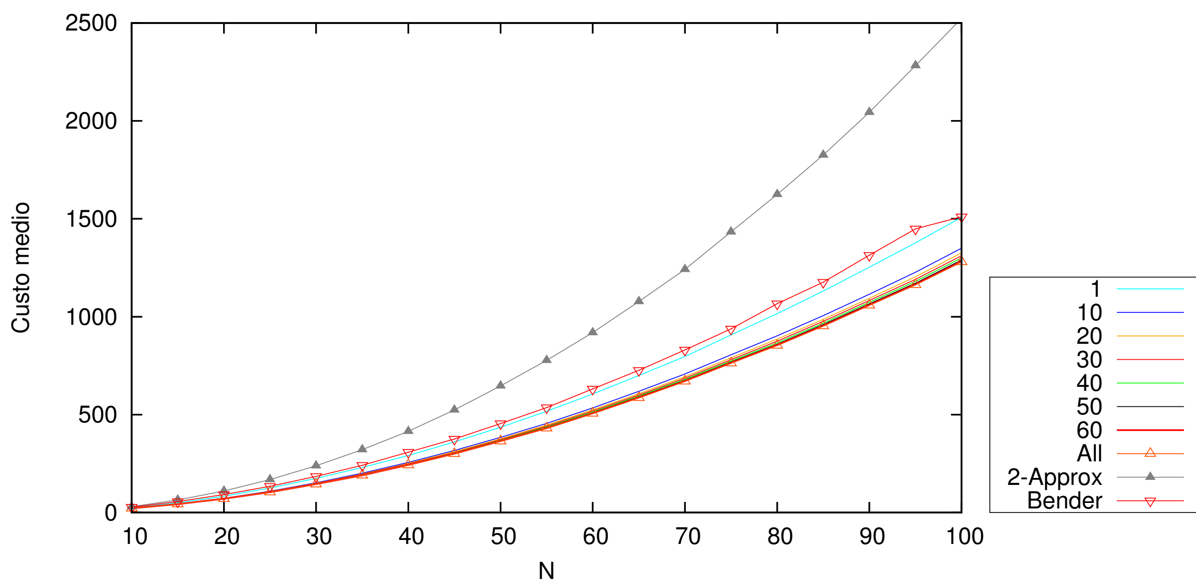

Figura 2.4: Este gráfico mostra o resultado obtido ao executar experimentos com nossa heurística em permutações grandes. Os parâmetros A e B são definidos no intervalo [0..10], e o parâmetro C é fixado em 0. Cada curva no gráfico cuja legenda é um número mostra a taxa de aproximação média quando consideramos esse número de configurações aleatórias de parâmetros. A curva `All` representa o melhor resultado quando todas as configurações de parâmetros são consideradas. A curva `2-Approx` corresponde aos resultados do algoritmo de Kececioglu e Sankoff [31]. Enquanto a curva `Bender` corresponde aos resultados do algoritmo de aproximação $O(\lg n)$ [10].

Os parâmetros A e B são definidos no intervalo [0..10], e o parâmetro C é fixado em 0, de forma semelhante ao que foi feito na segunda rodada de experimentos para permutações pequenas ($TEST_2$). Neste ponto, poderíamos ter utilizado cada combinação dos parâmetros A, B, e C no conjunto $\{0, 1, 2, \ldots 5\}$, mas decidimos seguir com a ideia de que A e B são os parâmetros mais importantes, conforme foi notado na seção anterior. Novamente foram escolhidas configurações de parâmetros aleatoriamente, de forma que este processo foi repetido 10 vezes e o resultado considerado é a media destas execuções.

A Figura 2.4 apresenta para permutações grandes uma análise semelhante à mostrada na Figura 2.3 para permutações pequenas. Contudo, neste caso empregamos o custo médio para a análise. A curva para o algoritmo de $O(\lg n)$-aproximação é identificado por `Bender`.

A análise para permutações grandes reforça nossa conclusão que a abordagem desenvolvida nesse trabalho fornece melhores resultados quando comparada com outras abordagens disponíveis na literatura. De fato, executando apenas um conjunto pequeno de configurações de parâmetros, escolhidas aleatoriamente, é produzido um resultado muito melhor. Notamos na Figura 2.4 que a curva de `Bender` esta acima da curva 1 (execução de uma configuração escolhida aleatoriamente), o que implica que somente uma execução já pode ser suficiente para encontrar uma solução melhor do que qualquer outra abordagem conlhecida anteriormente.

Neste ponto nós obtivemos as seguintes conclusões:

– Algumas configurações apresentam maior eficácia. A análise sobre permutações pequenas mostra que a melhor configuração de parâmetros depende do tamanho da permutação.

– A qualidade de uma solução depende da quantidade de diferentes configurações de parâmetros que são empregadas.

– Há uma relação de custo-benefício entre a qualidade da solução e o tempo de execução.

– O método desenvolvido neste trablho supera outros métodos conhecidos na literaturam, até mesmo quando executamos apenas um número pequeno de configurações de parâmetros.

Como última conclusão, destacamos que mesmo sabendo que algumas configurações de parâmetros são melhores do que outras, escolher configurações aleatoriamente permite gerar resultados de boa qualidade, como é mostrado nas Figuras 2.3 e 2.4. A Tabela 2.3.3 ajuda a sustentar este argumento, pois mostra o custo médio quando um dado número de configurações de parâmetros é empregado.

Na tabela, seja x o número de configurações, nós usamos a legenda `Melhor` para idenficar os casos onde as x melhores configurações foram usadas. As melhores configurações de parâmetro foram encontradas executando todas as configurações possíveis e ordenando-as pelo custo médio das soluções. A legenda `Aleatório` é usada para mostrar os casos onde x configurações foram escolhidas aleatoriamente.

Para avaliar os resultados de `Melhor` em comparação com `Aleatório`, é mostrado o `Gap` entre estes dois resultados na Tabela 2.3.3. Seja *best* o custo médio quando a

		Número de execuções							
N	**Tipo**	**1**	**10**	**20**	**30**	**40**	**50**	**60**	**all**
	Melhor	23.23	22.03	21.89	21.84	21.81	21.71	21.69	21.56
10	**Aleatório**	23.90	21.88	21.74	21.68	21.63	21.59	21.57	21.56
	GAP(%)	2.9	-0.7	-0.7	-0.7	-0.8	-0.5	-0.6	0.0
	Melhor	47.78	44.34	43.86	43.68	43.63	43.35	43.33	42.84
15	**Aleatório**	49.52	44.14	43.61	43.30	43.09	42.95	42.87	42.84
	GAP(%)	3.6	-0.5	-0.6	-0.8	-1.2	-0.9	-1.1	0.0
	Melhor	79.97	73.05	72.07	71.65	71.52	70.95	70.92	70.09
20	**Aleatório**	83.25	72.91	71.79	71.17	70.68	70.38	70.16	70.09
	GAP(%)	4.1	-0.2	-0.4	-0.7	-1.2	-0.8	-1.1	0.0
	Melhor	120.74	109.10	107.26	106.36	106.16	105.36	105.34	103.84
25	**Aleatório**	125.51	109.27	107.11	105.89	104.98	104.42	103.98	103.84
	GAP(%)	3.9	0.2	-0.1	-0.4	-1.1	-0.9	-1.3	0.0
	Melhor	169.85	152.11	149.74	148.65	148.20	146.67	146.62	144.64
30	**Aleatório**	174.36	152.02	149.21	147.51	146.21	145.49	144.85	144.64
	GAP(%)	2.7	-0.1	-0.4	-0.8	-1.3	-0.8	-1.2	0.0
	Melhor	223.21	200.69	197.37	195.79	195.26	193.11	193.07	189.84
35	**Aleatório**	230.03	201.13	197.07	194.55	192.58	191.23	190.16	189.84
	GAP(%)	3.1	0.2	-0.1	-0.6	-1.4	-1.0	-1.5	0.0
	Melhor	282.61	254.24	250.47	248.51	247.90	245.82	245.68	242.25
40	**Aleatório**	291.28	255.52	250.79	247.83	245.49	243.98	242.69	242.25
	GAP(%)	3.1	0.5	0.1	-0.3	-1.0	-0.7	-1.2	0.0
	Melhor	350.98	314.71	310.26	308.08	306.80	304.21	304.09	299.88
45	**Aleatório**	360.90	316.76	310.77	307.01	304.09	302.10	300.39	299.88
	GAP(%)	2.8	0.7	0.2	-0.3	-0.9	-0.7	-1.2	0.0
	Melhor	424.34	382.70	376.35	373.28	371.96	369.54	369.28	363.93
50	**Aleatório**	435.43	383.76	376.51	372.60	369.38	366.85	364.67	363.93
	GAP(%)	2.6	0.3	0.0	-0.2	-0.7	-0.7	-1.2	0.0
	Melhor	501.53	453.72	446.70	443.35	441.59	437.16	436.89	430.58
55	**Aleatório**	516.98	455.23	446.51	441.02	436.83	433.98	431.44	430.58
	GAP(%)	3.1	0.3	0.0	-0.5	-1.1	-0.7	-1.2	0.0
	Melhor	587.58	532.85	524.25	519.82	518.13	513.73	513.52	505.42
60	**Aleatório**	605.56	534.58	524.54	518.52	513.54	509.67	506.54	505.42
	GAP(%)	3.1	0.3	0.1	-0.3	-0.9	-0.8	-1.4	0.0
	Melhor	675.36	615.00	605.84	600.89	598.82	593.93	593.68	586.10
65	**Aleatório**	700.14	618.81	607.20	600.30	594.81	590.62	587.25	586.10
	GAP(%)	3.7	0.6	0.2	-0.1	-0.7	-0.6	-1.1	0.0
	Melhor	773.66	701.95	692.73	687.27	684.39	679.37	679.05	669.85
70	**Aleatório**	797.07	707.77	694.17	686.42	680.31	675.15	671.32	669.85
	GAP(%)	3.0	0.8	0.2	-0.1	-0.6	-0.6	-1.1	0.0
	Melhor	881.99	801.10	788.40	782.73	779.03	773.32	773.07	762.86
75	**Aleatório**	906.76	805.27	789.96	780.79	774.06	768.66	764.48	762.86
	GAP(%)	2.8	0.5	0.2	-0.2	-0.6	-0.6	-1.1	0.0
	Melhor	986.52	896.44	883.12	876.91	873.43	867.15	866.73	853.22
80	**Aleatório**	1015.99	902.43	885.17	874.98	866.89	860.05	855.08	853.22
	GAP(%)	3.0	0.7	0.2	-0.2	-0.7	-0.8	-1.3	0.0
	Melhor	1097.30	998.91	985.87	978.57	974.83	967.42	967.09	952.10
85	**Aleatório**	1130.78	1005.56	987.04	976.09	967.23	959.80	954.21	952.10
	GAP(%)	3.1	0.7	0.1	-0.3	-0.8	-0.8	-1.3	0.0
	Melhor	1214.29	1108.54	1093.73	1086.29	1080.77	1074.38	1074.19	1057.77
90	**Aleatório**	1253.30	1114.86	1094.42	1082.70	1073.54	1065.75	1059.97	1057.77
	GAP(%)	3.2	0.6	0.1	-0.3	-0.7	-0.8	-1.3	0.0
	Melhor	1336.75	1217.16	1200.62	1191.71	1186.91	1179.68	1179.14	1162.92
95	**Aleatório**	1377.68	1227.12	1204.08	1190.83	1180.77	1171.74	1165.52	1162.92
	GAP(%)	3.1	0.8	0.3	-0.1	-0.5	-0.7	-1.2	0.0
	Melhor	1464.52	1342.42	1324.34	1314.49	1308.80	1299.08	1298.53	1279.69
100	**Aleatório**	1508.17	1350.26	1326.11	1311.45	1300.03	1289.97	1282.57	1279.69
	GAP(%)	3.0	0.6	0.1	-0.2	-0.7	-0.7	-1.2	0.0

Tabela 2.3: Custo médio quando é empregado um dado número de configurações. `Melhor` identifica os casos onde o dado número de melhores configurações foi empregado. `Aleatório` identifica os casos onde as configurações foram escolhidas aleatoriamente. `Gap` mostra a distância entre os grupos de configurações `Melhor` e `Aleatório`.

abordagem `Melhor` é usada e *random* o custo médio quando a abordagem `Aleatório` é utilizado, `Gap` é definido como $gap = \frac{100 \times (random - best)}{best}\%$. Portanto, `Gap` é positivo quando `Melhor` fornece melhores resultados que `Aleatório`, e é negativo caso contrário. Quanto mais próximo de zero for o valor de `Gap`, mais parecidos serão os resultados de `Melhor` e `Aleatório`.

`Gap` é obviamente positivo quando somente uma configuração de parâmetros é escolhida aleatoriamente. Isto é esperado porque a melhor configuração sempre é melhor ou igual a qualquer configuração de parâmetros escolhida aleatoriamente. Também notamos que a melhor configuração é entre 2.5% e 4% melhor do que uma configuração aleatória, como pode ser visto na coluna `Gap`.

Por outro lado, quando são escolhidos mais configurações de parâmetros, notamos que a maior parte dos valores de `Gaps` passam a ser negativos, o que significa que configurações aleatórias passaram a ser as melhores.

Este fato nos levou a conclusão de que empregar diversidade na escolha de configurações de parâmetros tem impacto positivo na qualidade dos resultados. As melhores configurações geralmente fornecem o mesmo valor para as mesmas classes de permutação. As outras configurações geralmente retornam bons resultados para algumas classes de permutações, mas resultados ruins para outras classes. Quando a escolha de parâmetros é feita de forma aleatória, são obtidos resultados mais diversificados, o que aumenta a chance de que a combinação destes resultados seja melhor.

2.4 Publicação

Os resultados deste capítulo foram apresentados na conferência ACM BCB'2013 (Conference on Bioinformatics, Computational Biology and Biomedical Informatics), realizada em Washington (EUA), em setembro de 2013, sob o título "Heuristics for the Sorting by Length-Weighted Inversion Problem" (Thiago Silva Arruda, Ulisses Dias e Zanoni Dias) [2].

Capítulo 3

Meta-Heurística GRASP

No Capítulo 2, nós mostramos um método heurístico que desenvolvemos para o problema de Ordenação de Permutações por Reversões Ponderadas. Neste capítulo, nós apresentamos uma nova meta-heurística para o problema, que além de fornecer soluções para permutações sem sinal, também pode ser aplicada para permutações com sinal.

A estrutura desta meta-heurística é baseada no modelo *GRASP*, que foi apresentado originalmente por Feo e Resende [22]. As etapas empregadas são as mesmas para ambas as variações do problema (com sinal e sem sinal), com diferenças em alguns dos algoritmos que tratam características específicas de cada variação.

Dada uma solução inicial, a meta-heurística busca iterativamente melhorar esta solução. De forma que a cada iteração é realizada uma busca local por uma solução melhor no conjunto vizinhança da solução corrente. O conjunto vizinhança $NG(s)$ de uma solução s é composto por soluções resultantes da aplicação de operações de modificação em s.

Na Seção 3.3, nós mostramos em detalhes a estrutura da meta-heurística e a seguir nós listamos os principais pontos do nosso método.

1. Uma solução inicial é gerada utilizando um algoritmo específico para cada variação do problema (com sinal ou sem sinal).

2. Nós definimos analiticamente o conjunto vizinhança $N(s)$ de uma solução s, que especifica as operações para transformar s em s' tal que $s' \in N(s)$. A nossa definição de vizinhança é apresentada na Seção 3.2.

3. Desenvolvemos algoritmos heurísticos para construir uma nova solução $s' \in N(s)$. Estes algoritmos são descritos na Seção 3.4.

As duas últimas etapas apresentadas na lista acima são as mesmas para ambas as versões do problema de ordenação por reversões ponderadas. De fato, estas etapas poderiam ser utilizadas na maioria dos problemas do campo de rerranjo de genomas, sem nenhuma modificação nos conceitos apresentados nas seções 3.2 e 3.3, sendo somente necessário modicações para tratar as operações específicas de cada problema.

Na Seção 3.4, são apresentados os algoritmos que nós desenvolvemos especificamente para o problema que abordamos neste trabalho. Além disto, apresentamos na Seção 3.1 um histórico da pesquisa, compreendendo algumas das abordagens intermediárias que exploramos até desenvolvermos o método GRASP que apresentamos neste capítulo.

Por fim, na Seção 3.5 nós apresentamos os resultados experimentais obtidos com a nossa meta-heurística.

3.1 Histórico da pesquisa

Nesta seção, nós mostramos alguns dos experimentos e análises que realizamos antes de atingirmos os resultados finais deste capítulo.

Primeiramente, na Seção 3.1.1, nós realizamos a avaliação do método paramétrico já desenvolvido para permutações com sinal.

Na Seção 3.1.2 apresentamos nossos experimentos com uma meta-heurística que busca otimizar uma sequência de reversões executando busca local em segmentos da sequência. Denominamos âncoras as divisões entre estes segmentos.

3.1.1 Adaptação do método paramétrico

Como mostramos no Capítulo 2, nós desenvolvemos um algoritmo guloso paramétrico para permutações sem sinal, o qual resultou em uma publicação (Arruda *et al.* [2]). Com a continuidade da pesquisa, decidimos direcionar o foco para desenvolver um método para permutações com sinal.

Desta forma, nós decidimos analisar o método paramétrico para permutações com sinal. Contudo, como já foi explicado no Capítulo 2, este método possui alto custo computacional e devido a isto nós decidimos realizar uma adaptação dos resultados já computados para permutações sem sinal para permutações com sinal. Isto foi feito seguindo as seguintes etapas:

- Geramos uma base de permutações com sinal a partir da base de permutações sem sinal já existente. Para isto, realizamos a atribuição de sinais de forma aleatória aos elementos de cada permutação.

- Considerando que temos os resultados para as configurações de parâmetros $A = [0..10], B = [0..10], C = 0$, o que totaliza 121 resultados para cada permutação, nós utilizamos estas sequências de reversões para ordenar a permutação com sinal correspondente.

- Como esperado, as sequências de reversões que ordenam uma permutação sem sinal é suficiente para posicionar todos os elementos corretamente na permutação respectiva na permutação com sinal. Contudo, alguns elementos podem permanecer com sinais negativos, para tratar estes casos nós realizamos um pós-processamento que aplica reversões unitárias para corrigir sinais.

- Por fim, nós selecionamos a solução com menor custo dentre as 121 soluções computadas para cada permutação.

Após a realização da computação deste método, nós iniciamos a análise destes resultados. Para comparar estes resultados, nós utilizamos os resultados do `GRIMM`, que é uma

implementação do algoritmo para a ordenação por reversões com sinal de custos unitários por Tesler e Glenn [44].

Os resultados deste método são apresentados na Figura 3.1, que mostra o custo médio, e na Figura 3.2, que mostra o número de reversões. Em ambos gráficos, há uma comparação entre os resultados gerados conforme o método descrito anteriormente e os resultados fornecidos pelo `GRIMM`. Nestes gráficos, o rótulo `PARAMETRICO` representa o método que descrevemos, enquanto o rótulo `GRIMM` representa os resultados fornecidos pelo algoritmo GRIMM [44].

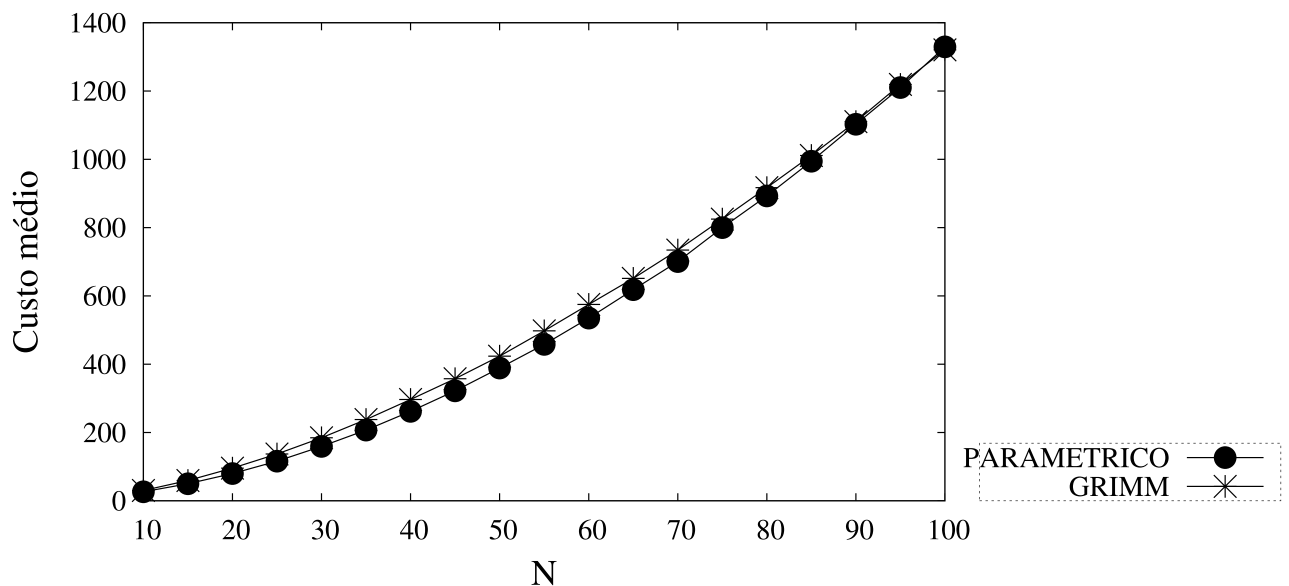

Figura 3.1: Este gráfico mostra uma análise comparativa do custo médio entre os resultados para permutações com sinal. A curva `PARAMETRICO` refere-se aos resultados gerados para permutações com sinal a partir do método paramétrico apresentado no Capítulo 2. A curva `GRIMM` refere-se aos resultados fornecidos pelo algoritmo GRIMM [44].

Podemos notar no gráfico da Figura 3.1 que para permutações de tamanho $n \leq 90$ o método `PARAMETRICO` apresenta resultados ligeiramente melhores do que o `GRIMM`. Contudo para permutações com tamanho $n > 90$ os resultados se tornam inferiores aos do `GRIMM`.

Como o `GRIMM` não é um algoritmo para o problema de ordenação por reversões ponderadas, nós consideramos estes resultados não satisfatórios, o que nos levou a explorar novas abordagens, como mostrado na próxima seção.

3.1.2 Método GRASP com âncoras

A próxima abordagem que utilizamos consistiu em aplicar ideias do trabalho desenvolvido por Dias et al. [20], que apresenta uma meta-heurística *GRASP* (a Seção 1.2 apresenta conceitos sobre *GRASP*) originalmente para o problema de ordenação por reversões quase-simétricas. A seguir é apresentado um resumo deste método.

1. Dada uma solução inicial, que é uma sequência de k reversões, são escolhidos alguns pontos ao longo desta sequência, estes pontos são denominados âncoras.

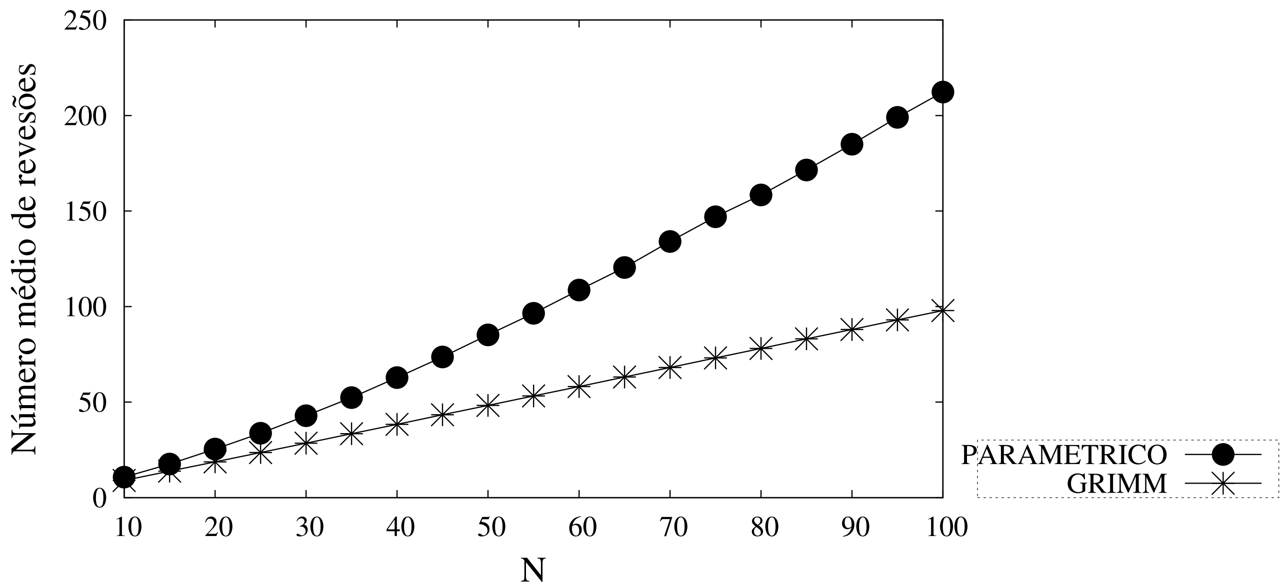

Figura 3.2: Este gráfico mostra uma análise comparativa do número de reversões entre os resultados para permutações com sinal. A curva `PARAMETRICO` refere-se aos resultados gerados para permutações com sinal a partir do método paramétrico apresentado no Capítulo 2. A curva `PARAMETRICO` refere-se aos resultados fornecidos pelo algoritmo GRIMM [44].

2. Os segmentos da sequência entre as âncoras são tratados como subproblemas. Por exemplo, sejam i e j, $1 \leq i < j \leq k$ duas âncoras consecutivas, temos o subproblema de transformar a permutação π_i na permutação π_j, o que é equivalente ao problema de ordenar a permutação $\sigma = \pi_j^{-1}\pi_i$.

3. Para buscar melhorias nas soluções destes subproblemas, é utilizado um algoritmo *GRASP* para construir uma nova solução. Um algoritmo *GRASP* é caracterizado por realizar escolhas aleatórias em um conjunto de elementos, utilizando uma função gulosa para determinar a probabilidade de escolha dos elementos.

Nós realizamos experimentos com este método para o problema de ordenação por reversões ponderadas. Nestes experimentos, utilizamos 5 âncoras, que foram escolhidas aleatoriamente a cada rodada. Nós executamos 100 rodadas.

Para a construção de uma nova solução, utilizamos uma variação do algoritmo da Bergeron [11], de forma que a cada iteração é construído o conjunto de reversões seguras e dentre estas reversões são selecionadas as 5 com menor custo. Em seguida uma reversão é escolhida utilizando o Algoritmo 5, em que a probabilidade de uma reversão ser escolhida é inversamente proporcional ao custo.

Os gráficos nas Figuras 3.3 e 3.4 mostram os resultados destas heurísticas em relação ao custo médio e ao número de reversões, respectivamente. Nós observamos que a nossa implementação da heurística, denotada por `ANCORAS`, apresentou melhores resultados do que os métodos `GRIMM`, que representa o algoritmo para o problema de ordenação de permutações com custos unitários, e o método `PARAMETRICO`, que representa o método apresentado anteriormente na Seção 3.1.1.

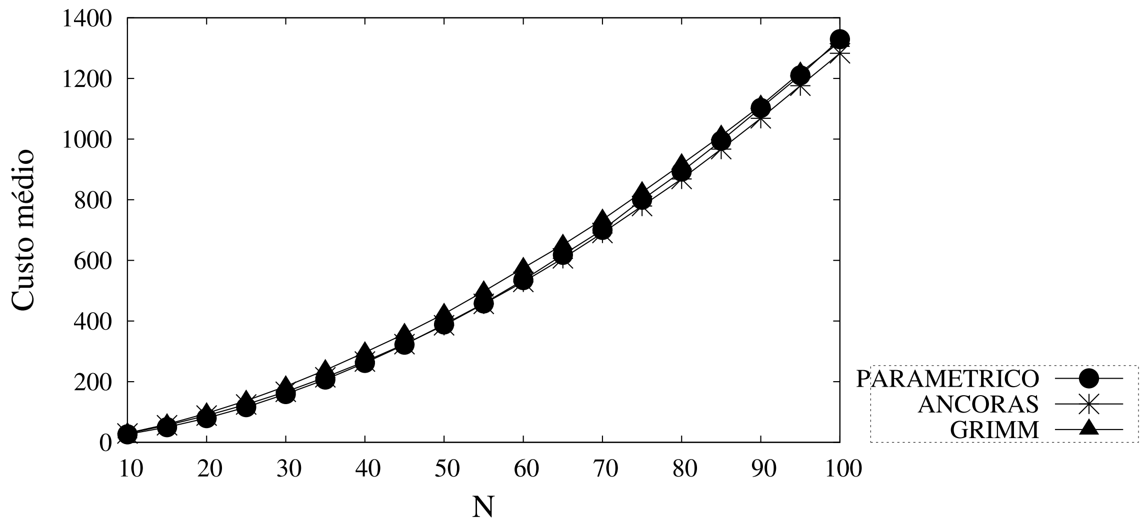

Figura 3.3: Este gráfico mostra uma análise comparativa do custo médio entre os resultados para permutações com sinal. A curva `GRIMM` representa o algoritmo para o problema de ordenação de permutações com custos unitários, a curva `PARAMETRICO` representa o método apresentado anteriormente na Seção 3.1.1, a curva `ANCORAS` representa o método *GRASP* com âncoras, que apresentamos nesta seção.

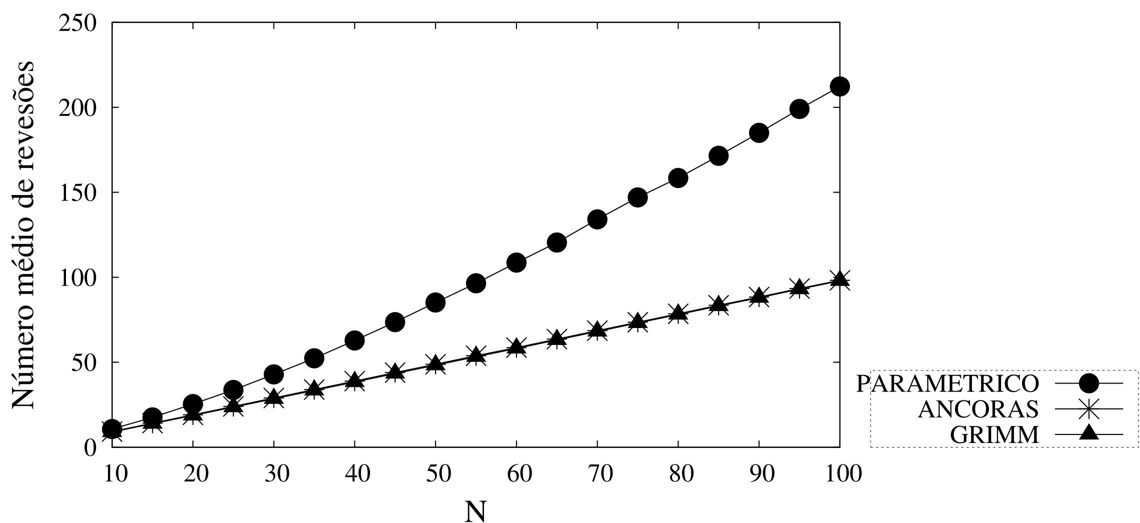

Figura 3.4: Este gráfico mostra uma análise comparativa do número de reversões entre os resultados para permutações com sinal. A curva `GRIMM` representa o algoritmo para o problema de ordenação de permutações com custos unitários, a curva `PARAMETRICO` representa o método nós apresentamos anteriormente na Seção 3.1.1, a curva `ANCORAS` representa o método *GRASP* com âncoras que apresentamos nesta seção.

A partir deste resultado nós concluímos que a meta-heurística *GRASP* possui um bom potencial a ser explorado, o que nos levou a extensão deste trabalho, o qual é mostrado em mais detalhes no decorrer deste capítulo.

3.2 Vizinhança

Primeiramente destacamos que adotamos nesta Seção a representação de uma solução como uma sequência de permutações s, diferentemente de uma sequência de reversões ϱ que utilizamos anteriormente.

Seja s uma solução arbitrária, a vizinhança $N(s)$ é um conjunto de soluções que possuem um nível de similaridade a s. As definições 17 e 18 estabelecem quando uma solução s' está em $N(s)$.

Definição 17. *Seja* $s =< s_0, s_1, \ldots, s_m >$ *uma solução em* S *(conjunto de soluções), então* $sub(s, p, q)$ *é definida como a subsequência* $< s_p, s_{p+1}, \ldots, s_q >$*, para* $0 \leq p < q \leq m$*.*

Definição 18. *A solução* $s' =< s'_0, s'_1, \ldots, s'_{m'} >$ *está em* $N(s)$ *se e somente se* $sub(s, 0, p) = sub(s', 0, p)$ *e* $sub(s, q, m) = sub(s', q', m')$*, para* $0 \leq p < q \leq m$ *e* $p < q' \leq m'$*.*

Observamos que $N(s)$ pode ter um grande número de elementos. De fato, por esta definição, qualquer solução s' está em $N(s)$ desde que seja possível fazer as atribuições $p = 0$ e $q = m$. Desta forma, seria necessário um algoritmo de complexidade exponencial para gerar tal conjunto $N(s)$, o que torna computacionalmente impraticável.

Nossa abordagem para obter um algoritmo polinomial consiste em restringir os valores que podem ser atribuídos a p e q. O que resulta na definição $N_f(s)$ (Definição 19), onde f representa a distância entre p e q. O Exemplo 3 ilustra a definição de $N_f(s)$.

Definição 19. *Seja* f *um número natural, a solução* $s' = < s'_0, s'_1, \ldots, s'_{m'} >$ *está na vizinhança* $N_f(s)$ *se e somente se* $sub(s, 0, p) = sub(s', 0, p)$*,* $sub(s, q, m) = sub(s', q', m')$ *e* $q - p + 1 = f$*, para* $0 \leq p$*,* $p < q \leq m$ *e* $p < q' \leq m'$*. Desta forma, a subsequência* $sub(s, p, q)$ *é uma janela de tamanho* f*.*

Exemplo 3. *Abaixo são esboçadas duas soluções que diferem por uma sequência contínua de permutações. Regiões diferentes são representados por caixas cinzas. Neste exemplo,* s *tem* x *permutações e* s' *tem* y *permutações nas respectivas caixas cinzas.*

$$s =< (+2\ \ +1\ \ -5\ \ \cdots\ \ -3\ \ +16\ \ +17),$$
$$(+2\ \ +3\ \ +4\ \ \cdots\ \ -1\ \ +16\ \ +17),$$
$$(-5\ \ -4\ \ -3\ \ \cdots\ \ -1\ \ +16\ \ +17),$$

x permutations

$$(-5\ \ -4\ \ -3\ \ \ldots\ \ +15\ \ +16\ \ +17),$$
$$(+1\ \ +2\ \ +3\ \ \cdots\ \ +15\ \ +16\ \ +17)>$$

$$s' =< (+2\ \ +1\ \ -5\ \ \cdots\ \ -3\ \ +16\ \ +17),$$
$$(+2\ \ +3\ \ +4\ \ \cdots\ \ -1\ \ +16\ \ +17),$$
$$(-5\ \ -4\ \ -3\ \ \cdots\ \ -1\ \ +16\ \ +17),$$

y permutations

$$(-5\ \ -4\ \ -3\ \ \ldots\ \ +15\ \ +16\ \ +17),$$
$$(+1\ \ +2\ \ +3\ \ \cdots\ \ +15\ \ +16\ \ +17)>$$

Portanto, $sub(s,0,p) = sub(s',0,p)$ e $sub(s,q,m) = sub(s',q',m')$. A janela $sub(s,p,q)$ tem tamanho $f = x+2$, pois também incluímos as permutações $s_p = s'_{p'}$ e $s_q = s'_{q'}$. Desta forma, é dito que $s' \in N_{x+2}(s)$. Um raciocínio similar leva a conclusão de que $s \in N_{y+2}(s')$.

3.3 Estrutura da Meta-heurística

Na Seção 1.2 nós apresentamos o Algoritmo 2, que é o modelo da meta-heurística *GRASP* que utilizamos como base para desenvolver o nosso método. Contudo, gostaríamos de destacar a adaptação que realizamos neste modelo: definimos a etapa de construção como uma sub-etapa da busca local (no modelo original estas eram etapas independentes). A nossa meta-heurística é descrita mais detalhadamente a seguir.

Seja $s =< s_0, s_1, \ldots, s_m >$ uma solução em S e seja $sub(s,p,q)$ uma janela (conforme as definições apresentadas na Seção 3.2), a sequência de reversões que transforma s_p em s_q é a mesma que poderia ser usada para transformar $s_q^{-1}s_p$ em ι, de forma que podemos substituir a sequência de reversões original da janela por $< s_q^{-1}s_p, s_q^{-1}s_{p+1}, \ldots, s_q^{-1}s_q >$, onde $s_q^{-1}s_q = \iota$. Portanto, criar uma nova sequência para a janela $sub(s,p,q)$ é equivalente a ordenar a permutação $\alpha = s_q^{-1}s_p$ por reversões ponderadas. Na Seção 3.4 é mostrado o método para a criação de uma nova janela $sub(s,p,q)$.

Seja $sub(s,p,q)$ uma janela de tamanho $q-p+1 = f$, uma solução encontrada para ordenar a permutação $s_q^{-1}s_p$ pode ser facilmente transformada em uma sequência para transformar s_p em s_q. Se esta nova sequência custa menos do que $sub(s,p,q)$, a janela $sub(s,p,q)$ é substituída pela nova sequência, para criar uma nova solução $s' \in N_f(s)$.

Destacamos que para os casos em que temos o tamanho da sequência de solução s menor do que o temanho da janela, $|s| < f$, utilizamos o tamanho da sequência como o tamanho da janela. Ou seja, na prática o tamanho da jenela em uma solucão s é $f = min(k, |s|)$.

Para uma dada solução s de tamanho m, existem $m - f + 1$ janelas diferentes de tamanho f em s. Experimentalmente verificamos que as janelas com maior custo total apresentam maior probabilidade de serem melhoradas. Contudo, para preservar a diversidade nas soluções, evitando que o algoritmo atinja mínimos locais rapidamente, nós adotamos um mecanismo para escolher aleatoriamente uma janela de forma que a probabilidade de uma janela ser escolhida é proporcional ao respectivo custo. A seguir é mostrado o método para selecionar uma janela a cada iteração.

1. Primeiramente é calculado o custo de cada janela $sub(s, p, q) =< s_p, s_{p+1}, \ldots, s_q >$ somando o custo de cada reversão ρ_k tal que $s_k = s_{k-1}\rho_k$, $p < k \leq q$. Em seguida são selecionadas as janelas de maior custo. O número de janelas selecionadas é definido pelo parâmetro nomeado `limite_janelas`. As janelas não selecionadas nesta etapa são descartadas.

2. A janela é selecionada pelo algoritmo *roulette wheel selection mechanism*, que é bem comum em Algoritmos Genéticos [5], o pseudocódigo é apresentado no Algoritmo 5. Neste algoritmo, a variável `elementos` recebe uma lista de janelas selecionadas no item 1. A pontuação de cada janela é dada por $pontuacoes[i] \leftarrow custo[i] - custo[min] + 1$, onde $custo$ é a sequência de custo de cada janela na variável `elementos` e min é o índice do menor elemento. Em seguida, é gerado um número aleatório $R \in [0..T]$, onde T é a soma de todos os custos em $pontuacoes$. Finalmente é selecionada a primeira janela na sequência tal que, quando todos as pontuações prévias são somadas, é obtido um valor menor ou igual a R.

Após o intervalo da janela ter sido selecionado, nós empregamos o algoritmo descrito na seção a seguir (Seção 3.4) para gerar uma nova sequência de reversões para a janela selecionada. Caso possua menor custo, esta sequência substituirá a sequência corrente. Este processo é repetido por um número de iterações definido previamente por um parâmetro. Como esperado, quanto maior o número de iterações, melhor a resposta gerada. Um estudo da relação de custo-benefício entre qualidade da solução e tempo computacional é necessário para determinar o número de iterações adequadas para uma determinada aplicação.

No nosso algoritmo de busca local, a primeira solução que melhora a solucão corrente é adotada como a nova solução corrente. Esta estratégia segue o modelo *first-improving*, que foi discutido na Seção 1.2.

3.4 Construção de soluções

A etapa de construção de soluções consiste, a cada iteração, em escolher uma reversão de um conjunto de reversões disponíveis, até que a solução esteja completa.

Um método com este propósito pode ter variações tanto em relação ao algoritmo de geração do conjunto de reversões candidatas quanto ao algoritmo para escolher uma reversão para compor a solução. Ao longo da pesquisa realizamos experimentos e analisamos diversos métodos.

Nesta seção, apresentamos duas abordagens para a construção de soluções. Essas duas abordagens empregam estratégia *GRASP* para a escolha de reversões, mas diferenciam-se em relação ao método para gerar o conjunto de reversões e também nas heurísticas utilizadas na estratégia *GRASP* para escolher reversões.

A **primeira abordagem** foi desenvolvida a partir do foco em estender o nosso trabalho em ordenação por reversões ponderadas também para permutações com sinal. Para a geração do conjunto de reversões utilizamos uma versão adaptada do algoritmo desenvolvido inicialmente por Bergeron [11].

Na nossa **segunda abordagem** passamos a utilizar uma implementação do algoritmo *GRIMM*, de Tesler e Glenn [44], para obter soluções para o problema com sinal, o qual considera um conjunto de reversões maior a cada iteração. Também expandimos nosso algoritmo para a variação com permutações sem sinal. Para isto, desenvolvemos um algoritmo para construir um conjunto de reversões candidatas para este problema, o que resultou na segunda abordagem.

Nós apresentamos a seguir as duas abordagens em detalhes.

Primeira abordagem

Desenvolvemos esta abordagem para a variação com sinal do problema. Neste método, para uma permutação π, uma reversão é selecionada do conjunto de reversões candidatas para estender a solução parcial a cada iteração. Para cada reversão candidata, é atribuída uma pontuação baseada na respectiva probabilidade de produzir uma solução com reversões curtas. Em outras palavras, nós estimamos o benefício de cada reversão. Nós analisamos o benefício baseado em dois aspectos da permutação: número de pares orientados ($nop(\pi)$) [11] e entropia ($ent(\pi)$).

O conceito de par orientado foi apresentado inicialmente por Bergeron [11] para ser usado em um algoritmo para o problema de ordenação por reversões com custos unitários. Na Definição 10 é apresentado o conceito de reversão induzida por um par orientado.

O algoritmo apresentado por Bergeron [11] utiliza reversões induzidas para maximizar o número de pares orientados na próxima permutação da sequência que forma a solução, o que somente é possível quando π tem pelo menos um elemento negativo. Se π tem somente elementos positivos, então nós temos um "hurdle" como definido por Hannenhalli e Pevzner [29], o qual não possui pares orientados, $nop(\pi) = 0$. Nestes casos, podemos utilizar as operações "hurdle cutting" e "hurdle merging" como mostrado por Bergeron [11].

Desta forma, nós buscamos selecionar reversões que diminuam o nível de entropia e ao mesmo tempo nós buscamos também selecionar reversões que aumentem o número de pares orientados, seguindo a linha de raciocíno proposta por Bergeron. Considerando estas duas métricas, nós desenvolvemos a nova métrica *enop*, conforme mostrada na Definição 20.

Definição 20. *Dada uma permutação π, nós definimos a métrica enop como uma relação entre as métricas entropia $ent(\pi)$ (Definição 9) e o número de pares orientados $nop(\pi)$ (Definição 11).*
$enop(\pi) = ent(\pi)/nop(\pi)$

Em nossos experimentos, nós utilizamos tanto a *enop* quanto somente a entropia como métricas para computação do benefício de reversões. Isto resultou nas duas funções para computação de benefício mostradas na Definição 21.

Definição 21. *Seja ρ uma reversão aplicável a uma permutação π, nós definimos o seu benefício em duas formas diferentes:*
$\delta_1(\pi, \rho) = \frac{ent(\pi) - ent(\pi\rho)}{cost(\rho)}$
$\delta_2(\pi, \rho) = \frac{enop(\pi) - enop(\pi\rho)}{cost(\rho)}$

Ambas as funções de benefício tem a desejável propriedade que, se continuarmos aplicando reversões com benefício positivo em π, vamos eventualmente atingir a permutação identidade. Contudo, como já foi explicado no capítulo anterior, encontramos algumas permutações para as quais não há reversões com benefício positivo. Devido a isto, decidimos considerar somente reversões que são induzidas por pares orientados quando π tem elementos negativos. Desta forma, garantimos que pelo menos um *breakpoint* será removido. Quando π não tem elementos negativos, como explicado anteriormente, as operações "hurdle cutting" e "hurdle merging" [11] são usadas.

Quando nós temos reversões induzidas por pares orientadas, a reversão que será usada para estender a solução inicial é escolhida seguindo as duas etapas a seguir.

1. Calculamos o benefício de cada reversão ρ induzida por um par orientado, então selecionamos um número limitado de reversões com os maiores benefícios. O número exato de reversões que serão movidas para a segunda fase é definida como um parâmetro nomeado `numero_reversoes`. As reversões que não foram selecionadas serão descartadas. Desta forma temos uma lista restrita de elementos (RCL), como apresentado na Seção 1.2.

2. As reversões que não foram descartadas no item 1 devem ser selecionadas pelo algoritmo *roulette wheel mechanism* descrito no Algoritmo 5. Cada permutação tem probabilidade de ser selecionada proporcional ao respectivo benefício. A variável `elementos` recebe uma lista de reversões selecionadas na etapa anterior e a variável `pontuacoes` recebe o benefício de cada reversão.

Considerando que temos duas funções para calcular o benefício de uma inversão (δ_1 e δ_2), executamos todo o algoritmo duas vezes, utilizando uma função diferente em cada uma das execuções. Em seguida, escolhemos como a solução final aquela que possui o menor custo.

Segunda abordagem

Nesta nossa segunda abordagem, novamente desenvolvemos um algoritmo para construir uma solução para uma permutação π. Nesta abordagem somente é utilizado a entropia ($ent(\pi)$) como métrica para computar o benefício de reversões.

É notado que $ent(\pi) = 0$ se e somente se $\pi = \iota$. Portanto, buscamos selecionar reversões que diminuam a entropia. Como um critério adicional, deve-se selecionar reversões que não custem muito. Portanto, a seguinte definição junta ambos os conceitos em uma função gulosa. Notamos que no Capítulo 1 nós apresentamos duas definições para entropia, uma para a variação sem sinal (Definição 8) e outra para a variação com sinal (Definição 9).

Definição 22. *Seja π uma permutação (com sinal ou sem sinal) e seja ρ uma reversão, é definido o benefício da função como* $ben(\pi, \rho) = \frac{ent(\pi) - ent(\pi\rho)}{cost(\rho)}$.

Esta definição de benefício é bastante similar à que foi utilizada no algoritmo do capítulo anterior, que era restrito a permutações sem sinal. De modo análogo, foram encontradas algumas permutações para as quais não há reversões com benefício positivo que possam ser aplicadas. Para garantir que uma solução será criada, o conjunto de reversões considerado deve ser limitado. A forma que o conjunto de reversões é limitado difere quando são consideradas permutações sem sinal ou com sinal.

Seja π uma permutação com sinal, Hannenhalli e Pevzner [29] apresentaram um algoritmo polinomial para ordenar π usando o número mínimo de reversões. Este algoritmo pode gerar muitas soluções ótimas diferentes, pois em cada etapa é construído um conjunto de reversões seguras (reversões que levam a uma solução ótima). Este conjunto de reversões seguras é um superconjunto das reversões consideradas no algoritmo de Bergeron [11], pois não é limitado às reversões induzidas por pares orientados. Também realizamos experimentos com o algoritmo apresentado por Siepel [39], que gera todas as soluções ótimas para o problema. Entretanto, na prática esse método não nos levou a encontrar soluções melhores.

Portanto, utilizando o conjunto de reversões fornecido por este algoritmo em cada etapa no nosso método, garantimos que eventualmente atingiremos a permutação identidade.

Seja π uma reversão sem sinal, Caprara [16] provou que encontrar uma solução ótima para o problema de ordenação por reversões é NP-Completo. Contudo, pode ser empregado o número de *breakpoints* (veja Definição 5) para identificar reversões que levam a uma permutação mais próxima da identidade. Para a compreensão deste conceito nós utilizaremos a Definição 7 sobre *strip*.

Além disto, Kececioglu e Sankoff [31] provaram que toda permutação que tem pelo menos uma *strip* decrescente tem pelo menos uma reversão que remove pelo menos um *breakpoint* (como explicado no Teorema 7 e Lema 8). Se todas as *strips* em π forem crescentes, podemos reverter qualquer *strip* crescente para se obter uma *strip* decrescente. Isto garante que pelo menos um *breakpoint* é removido a cada duas reversões aplicadas.

Como uma reversão pode remover no máximo 2 *breakpoints*, esta abordagem permite a construção de um algoritmo aproximado de fator 4 para o problema ordenação por reversões de custos unitárias para permutações sem sinal, o qual pode ser implementado com complexidade de tempo $O(n^2)$.

Definição 23. *Dada uma permutação π e uma reversão ρ, a diferença de benefício entre*

π e $\pi \cdot \rho$ é definido a seguir:

$$\Delta_{ben(\pi,\rho)} = ben(\pi \cdot \rho) - ben(\pi)$$

Teorema 7. *Se o elemento k pertence a uma strip decrescente e o elemento $k-1$ pertence a uma strip crescente, então existe uma reversão ρ tal que $\Delta_{b(\pi,\rho)} < 0$ (Definição 3.4).*

Lema 8. *Seja π uma permutação com ao menos uma strip decrescente. Então, existe uma reversão ρ tal que $\Delta_{b(\pi,\rho)} < 0$.*

Definição 24. *Conforme o Teorema 7 e Lema 8, nós podemos definir reversões que removem breakpoints. Seja $[w..k]$ uma strip decrescente em uma permutação π, até duas reversões podem ser aplicadas para diminuir o número de breakpoints.*

- *Caso exista uma strip crescente $[l..k-1]$:*
  - $\rho_1 = (\pi_k^{-1} + 1, \pi_{k-1}^{-1})$, *caso* $\pi_k^{-1} < \pi_{k-1}^{-1}$
  - $\rho_1 = (\pi_k^{-1} + 1, \pi_{k-1}^{-1})$, *caso* $\pi_k^{-1} > \pi_{k-1}^{-1}$
- *Caso exista uma strip crescente $[w+1..l]$:*
  - $\rho_2 = (\pi_w^{-1}, \pi_{w+1}^{-1} - 1)$, *caso* $\pi_w^{-1} < \pi_{w+1}^{-1}$
  - $\rho_2 = (\pi_{w+1}^{-1}, \pi_w^{-1} - 1)$, *caso* $\pi_w^{-1} > \pi_{w+1}^{-1}$

Estando ciente do conjunto de reversões consideradas para as versões com e sem sinal do problema podemos descrever o método geral para construção de soluções nesta abordagem, o qual é composto das etapas a seguir.

1. Construimos um conjunto de reversões distintas que removem pelo menos um *breakpoint*, tais reversões devem satisfazer a Definição 24. Caso não exista nenhuma reversão desse tipo (quando a permutação não tem *strips* decrescentes), nós construímos, alternativamente, um conjunto formado pelas reversões que invertem uma *strip* crescente.

2. Calculamos o benefício de cada reversão ρ que satisfaz às restrições previamente descritas, então é selecionado um número limitado de reversões que têm o máximo benefício. O número exato de reversões que serão consideradas na segunda etapa é definido como um parâmetro nomeado `numero_reversoes`. As reversões que não foram selecionadas nesta etapa serão descartadas.

3. Utilizando o algoritmo *roulette wheel mechanism*, descrito no Algoritmo 5, selecionamos uma reversão do conjunto obtido na etapa anterior. Cada permutação tem uma probabilidade de ser selecionada proporcional ao respectivo benefício. A variável `elementos` recebe uma lista de reversões selecionadas na etapa anterior e a variável `pontuacoes` recebe o benefício de cada reversão.

3.5 Resultados

Nesta Seção, nós descrevemos e analisamos os resultados obtidos dos experimentos que realizamos nos algoritmos apresentados neste capítulo. Para realizar estes experimentos nós implementamos nossos algoritmos em `C++`. O código fonte desta implementação está disponível para acesso.[1]

Os resultados são divididos em dois grupos, os quais são diferenciados em relação à abordagem utilizada para a **construção de soluções**, conforme tratado na Seção 3.4, configuração de parâmetros e configuração de execução.

A primeira abordagem abrange somente permutações com sinal, enquanto a segunda abordagem também trata de permutações sem sinal.

Além disto, foram utilizados conjunto de instâncias distintos para os experimentos realizados para cada abordagem.

3.5.1 Primeira abordagem

Nesta seção, nós mostramos os resultados de experimentos realizados com a nossa meta-heurística quando é utilizada a estrutura de Bergeron [11] para construção de soluções. A seguir nós apresentamos um resumo dos parâmetros e configuração de execução utilizados nesta abordagem.

1. Para o tamanho de janela f adotamos uma sequência de tamanhos de janelas. Em nossos experimentos obtivemos os melhores resultados com a sequência de tamanhos de janelas $< 14, 12, 10, 8, 6, 4 >$, de forma que executamos 150 iterações para cada tamanho de janela. No total nosso algoritmo executa 900 iterações.

2. Na Seção 3.3 nós mencionamos o parâmetro `limite_janelas`. Este parâmetro é definido como a porcentagem das janelas com maior custo que serão selecionados para serem utilizadas na etapa seguinte. Realizamos experimentos com vários valores de parâmetro, como 25%, 50% e 75%. Nossa conclusão final é que o valor 75% nos leva aos melhores resultados.

3. Na Seção 3.4, mencionamos o parâmetro `numero_reversoes`. Este parâmetro é fixado independentemente do número de reversões disponíveis. Nós realizamos experimentos com vários valores para `numero_reversoes`, como 3, 5, 10, 15 e 20. Nossa conclusão final é que 5 é o valor que leva aos melhores resultados.

Utilizando estes parâmetros foram obtidos os tempos de execução mostrados na Tabela 3.1.

Para encontrar a complexidade de tempo da nossa implementação, mostramos a seguir a análise de cada parte.

1. Utilizamos `GRIMM` para obter uma solução inicial, o qual possui tempo de execução $O(n^4)$.

[1] Disponível em `https://github.com/xth1/lwr`.

Tamanho	Tempo
10	0.6
15	1.5
20	3.1
25	5.1
30	7.3
35	9.9
40	12.2
45	14.5
50	16.6
55	21.7
60	25.1
65	27.7
70	32.1
75	32.1
80	27.5
85	29.3
90	31.2
95	32.8
100	34.4

Tabela 3.1: Tempo médio (em segundos) para processar cada permutação de um dado tamanho. Para cada tamanho, nós geramos 1000 permutações aleatórias.

2. O número de iterações é um parâmetro que denominamos por l.

3. Para selecionar uma janela, precisamos de tempo $O(m \log m)$, onde m é o número de permutações na solução inicial. Relembre que a primeira etapa para selecionar uma janela consiste de escolher um número limitado de janelas de alto custo, o que podemos fazer ordenando a lista de janelas. Portanto obtemos a complexidade $O(n \log n)$, pois $m = O(n)$.

4. Nós precisamos de $O(fn^2)$ para construir uma nova solução para substituir a janela. A cada etapa construímos um conjunto de reversões induzidas por pares orientados de tamanho $O(n^2)$ e precisamos de tempo $O(n)$ para computar o benefício de cada janela. Além disso, a nova janela terá tamanho proporcional a f.

A complexidade final deste algoritmo é $O(Solucao_Inicial + Iteracoes \times (Buscar_Janela + Melhorar_Janela))$, o que resulta em $O(n^4 + l(n \log n + fn^2))$. Depois de algumas simplificações, concluímos que nossa implementação executa em $O(n^4 + fln^2)$. Esta complexidade pode ser reduzida para $O(fln^2)$ se usarmos o algoritmo proposto por Tannier e Sagot [43] para obter a solução inicial, que executa em tempo $O(n\sqrt{n \log n})$.

A principal medida de qualidade usada em nossos experimentos é a diferença em custo entre a sequência produzida por nossa implementação e a solução inicial produzida pelo `GRIMM`.

Para n no intervalo $\{10, 15, \ldots, 100\}$, nós geramos 1000 permutações aleatórias e as utilizamos como entrada para a nossa implementação. A Figura 3.5 mostra com qual

frequência nossa abordagem melhora a solução inicial. Conseguimos melhorar a solução inicial em 94.0% dos casos de teste. Quando consideramos somente permutações grandes, tais como $n \geq 50$, nós notamos que 99.3% das soluções iniciais foram melhoradas.

Os padrões de preenchimento das barras na Figura 3.5 representam tamanhos de janelas e, portanto, as barras nos mostram quais tamanhos foram responsáveis pelo primeiro melhoramento na solução inicial. Como um exemplo, para $n = 100$ nós conseguimos melhorar 99.8% das soluções iniciais. Em 94.4% dos casos o primeiro melhoramento ocorre quando $f = 14$ e em 4.4% dos casos o primeiro melhoramento ocorre quando $f = 12$. Notamos que para o tamanho $N = 10$, nas iterações para $f = 12$ e $f = 14$ foi utilizado $f = 10$ na prática.

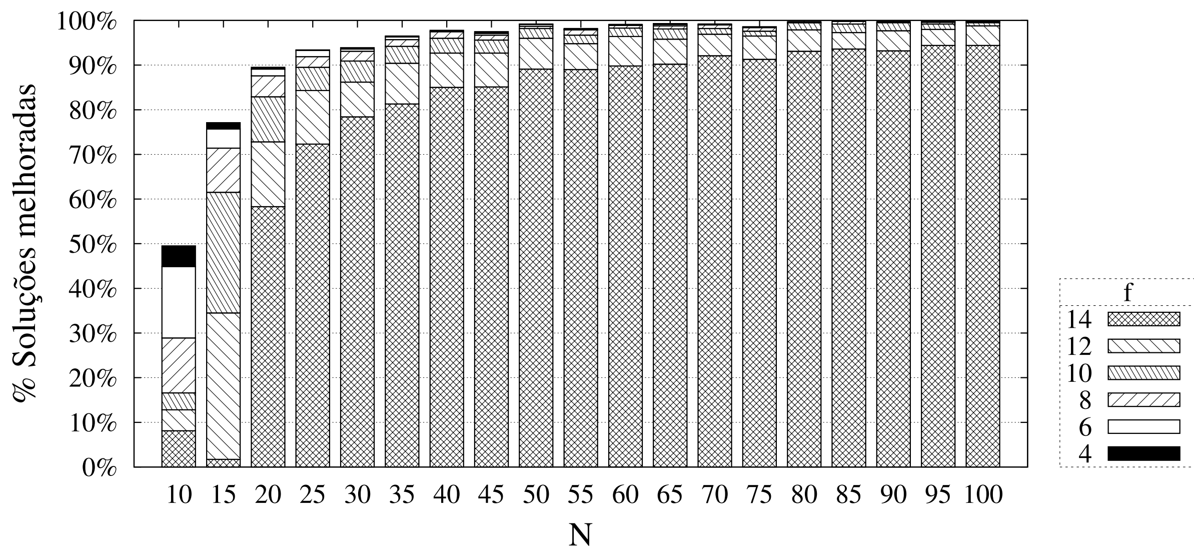

Figura 3.5: Este gráfico mostra a porcentagem de vezes que nosso algoritmo melhora a solução inicial. Em geral, a solução inicial foi melhorada em 94.0% dos casos. Se nós considerarmos somente permutações grandes tais que $n \geq 50$, nós observamos que 99.3% da soluções iniciais foram melhoradas. Nesse gráfico, os padrões de preenchimento das barras representam tamanhos de janelas e barras nos mostram quais tamanhos de janelas foram responsáveis pela primeira melhoria na solução inicial.Notamos que para o tamanho $N = 10$, nas iterações para $f = 12$ e $f = 14$ foi utilizado $f = 10$ na prática.

A Figura 3.6 mostra a porcentagem de melhoramento em média obtida usando nosso algoritmo. Seja $S_{inicial}$ a solução inicial e S_{final} a solução final produzida pelo nosso algoritmo *GRASP*. Definimos melhoramento como a diferença em custo entre $S_{inicial}$ e S_{final} dividido pelo custo de $S_{inicial}$, em outras palavras, $\%_melhoramento = 100 \times \frac{custo(S_{inicial}) - custo(S_{final})}{custo(S_{inicial})}$. Observamos que nossas soluções custam em torno de 12.6% menos que a solução inicial.

Continuamos nossa análide comparando nossos resultados contra o algoritmo anterior para o problema, desenvolvido por Swidan *et al.* [40]. Eles criaram um modelo de ordenação por reversões em que o custo para reverter uma subsequência de tamanho l da permutação é dado pela função $f(l) = l^{\alpha}$, para $\alpha \geq 0$. Eles conseguiram garantir a taxa de aproximação $O(\lg n)$ quando $\alpha = 1$, que é a mesma função de custo que usamos neste trabalho.

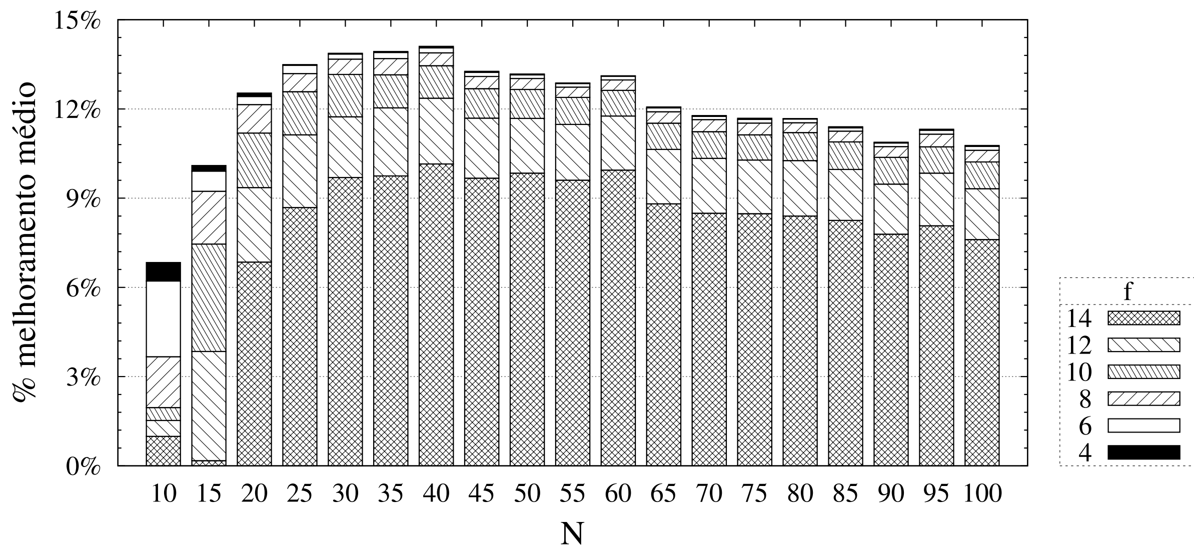

Figura 3.6: Este gráfico mostra a porcentagem de melhoramento obtido por nosso algoritmo. Observamos que nossa solução custa em torno 12.6% menos que a solução inicial. Nesse gráfico, os padrões de preenchimento das barras representam tamanhos de janelas e as barras mostram qual tamanho de janela foi responsável pelo melhoramento em média. Notamos que para o tamanho $N = 10$, nas iterações para $f = 12$ e $f = 14$ foi utilizado $f = 10$ na prática.

Nossa análise mostra que o nosso método encontra soluções que custam menos do que as obtidas pelo algoritmo de Swidan *et al.* em 97.5% dos casos. Quando consideramos somente permutações grandes, tais como $n \geq 50$, notamos que em 99.9% dos casos nossa solução custa menos do que aquelas obtidas por Swidan *et al.*

Na Figura 3.7, o eixo-Y representa a média do custo do conjunto de 1000 permutações e o eixo-X representa tamanhos de permutações. O rótulo `Swidan` representa o algoritmo proposto por Swidan *et al.* [40]. Como podemos ver, `Swidan` é superado tanto por `GRIMM` quanto por nossa abordagem (rotulada como `GRASP`). Em média, nossas soluções custam 23.3% menos do que soluções do algoritmo de `Swidan`.

Na Figura 3.8, mostramos o número de reversões usadas pela nossa abordagem. `Swidan` gera resultados com muitas reversões, o que é rasoável, dado que eles não estão tentando minimizar a quantidade de reversões que estão usando. `GRIMM` representa o valor mínimo para o gráfico, pois trata-se de um algoritmo ótimo para o problema de ordenação por reversões com custos unitários.

Observamos que nossa abordagem (rotulada como `GRASP`) aumenta em somente 1.8% o número de reversões geradas pelo `GRIMM`. Esta é uma propriedade desejável porque nós não queremos que nossa solução esteja muito distante da solução mais parsimoniosa. A mesma análise mostra que soluções fornecidas por `Swidan` tem 97.6% mais reversões do que o `GRIMM`.

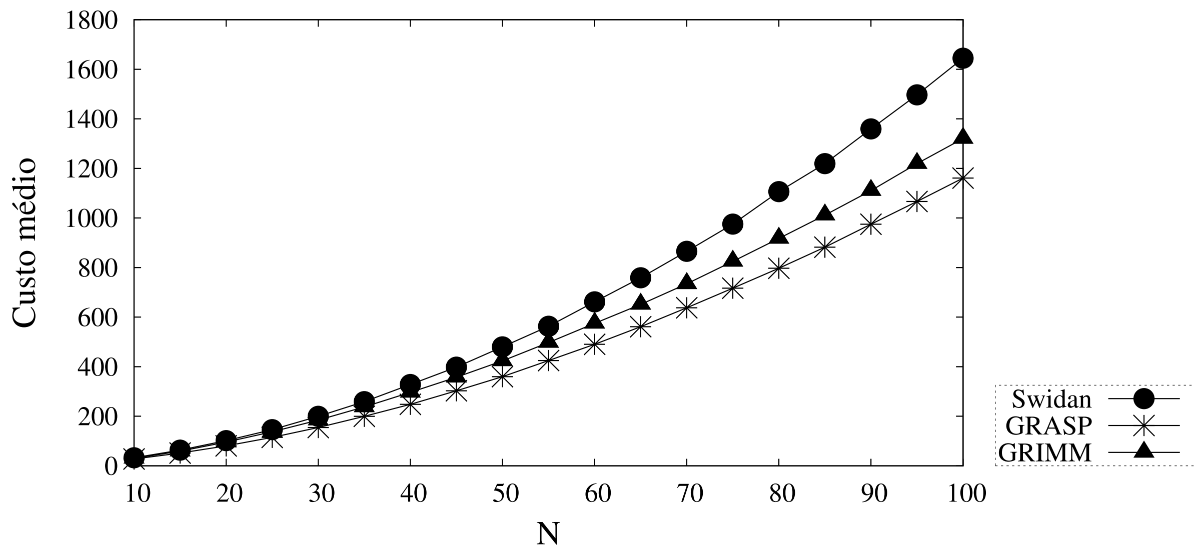

Figura 3.7: Este gráfico mostra uma análise comparativa entre 3 algoritmos. O eixo-Y representa a média de custo e o eixo-X representa o tamanho da permutação. O rótulo `Swidan` representa o algoritmo proposto por Swidan *et al.* [40]. `GRIMM` é uma solução ótima para o problema de ordenação por reversões de custo unitario. `GRASP` representa nossa abordagem.

3.5.2 Segunda abordagem

Usamos o programa `GRIMM` [44] para encontrar a solução ótima para o problema de ordenação por reversões com custos unitários, o qual é usado como solução inicial. A seguir mostramos os valores para os parâmetros que mencionamos na Seção 3.3.

1. Nesta abordagem, utilizamos uma nova estratégia para definir o tamanho de janela. Esta estratégia consiste em selecionar um valor, em um dado intervalo, a cada iteração. Em nossos experimentos, os melhores resultados foram obtidos quando selecionamos o valor aleatório para $f \in \{5, 6, \ldots, 20\}$.

2. Na Seção 3.3 nós mencionamos que, quanto mais iterações melhores seriam os resultados finais. Desta forma, uma relação de custo-benefício entre qualidade e tempo computacional é necessária. Em nossos experimentos, executamos até 2000 iterações. Contudo, observamos que possivelmente não seja necessário executar todas estas iterações para encontrar bons resultados, como será mostrado posteriormente nesta seção.

3. Na Seção 3.3 mencionamos o parâmetro `limite_janelas`. Assim como na primeira abordagem, adotamos o valor 75% para este parâmetro.

4. Na Seção 3.4 mencionamos o parâmetro `numero_reversoes`. Este parâmetro é fixado independentemente do número de reversões disponíveis. Assim como na primeira abordagem, adotamos o valor 5 para este parâmetro.

Para encontrar a complexidade de tempo de nossa implementação, computamos a complexidade de cada parte:

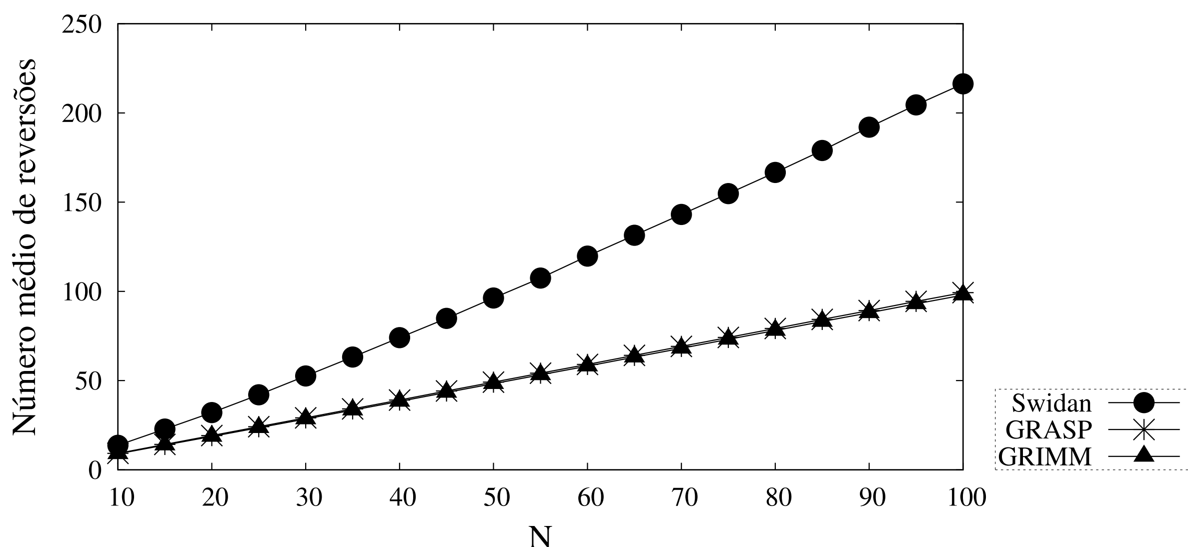

Figura 3.8: Este gráfico mostra a média do número de reversões de soluções obtidas por cada um dos três algoritmos usados na nossa análise. O eixo-Y representa a média no número de reversões e o eixo-X representa o tamanho da permutação. Os rótulos para algoritmos são os mesmo usados na Figura 3.7.

1. Usamos o `GRIMM` para obter uma solução inicial tanto para a versão sem sinal quanto para a com sinal. Para permutações com sinal, `GRIMM` fornece uma solução em tempo $O(n^4)$, esta solução é ótima para o problema de ordenação por reversões com custos unitários com sinal.

2. Para permutações sem sinal, primeiramente realizamos a atribuição de sinais para elementos da permutação. Esta tarefa é realizada por intermédio do `GRIMM`, que gera 1000 permutações com sinal a partir da permutação sem sinal da entrada[2]. `GRIMM` usa sua própria heurística para criar versões com sinal e todo o processo dura menos de um minuto, para permutações com até 100 elementos. Após selecionar a permutação com sinal que possui a menor distância de ordenação [3], usamos `GRIMM` novamente para encontrar uma solução ótima para o problema de ordenação por reversões em permutações com sinal em tempo $O(n^4)$. Seja $s =< s_0, s_1, \ldots, s_m >$ a solução para a permutação com sinal, nós transformamos esta solução de volta para a versão sem sinal:

   (a) Primeiramente removendo os sinais de cada elemento na permutação para criar a solução sem sinal $u =< u_0, u_1, \ldots, u_m >$.

   (b) Finalmente, removemos u_i de u se $u_i = u_{i+1}$, ou seja, se foi aplicada uma reversão cujo único efeito foi a troca do sinal de um elemento de s_i.

   A etapa mais cara para gerar a solução inicial apresenta complexidade $O(n^4)$.

3. O número de iterações é um parâmetro, denotado por l.

[2]Realizamos esta tarefa executando o seguinte comando `grimm -U 1000`

[3]Bader, Moret e Yan [6] mostraram que o número mínimo de reversões pode ser calculado em tempo linear quando não se deseja obter a sequência de reversões.

4. Para selecionar uma janela, precisamos de tempo $O(m \log m)$, onde m é o número de permutações na solução inicial. Relembramos que a primeira etapa para selecionar uma janela consiste em encontrar um número limitado de janelas de alto custo, o que pode ser feito ordenando a lista de janelas. Portanto, obtemos a complexidade final $O(n \log n)$, pois $m = O(n)$.

5. Para o caso com sinal, utilizamos um algoritmo de complexiade $O(fn^3)$ para construir uma nova solução para substituir a janela. Usamos o `GRIMM` para encontrar todas as reversões seguras, o que pode ser feito em tempo $O(n^3)$ e a nova solução terá tamanho proporcional a f. Também realizamos experimentos com o algoritmo apresentado por Siepel [39], que gera todas as soluções ótimas para o problema. Entretanto, na prática esse método não nos levou a encontrar soluções melhores.

6. Para o caso sem sinal, precisamos de tempo $O(fn^2)$ para construir uma nova solução para substituir a janela. Cada etapa seleciona $O(n)$ reversões que removem *breakpoints* e gastamos $O(n^2)$ para encontrar estas reversões. Além disso, precisamos de tempo $O(n)$ para computar o benefício de cada reversão. Além disso, a nova janela terá tamanho proporcional a f. Portanto chegamos à complexidade final quando usamos $O(fn^2 + fn^2) = O(fn^2)$.

A complexidade final para ambas as versões pode ser dada por $O(Solucao_Inicial + Iteracoes \times (Buscar_Janela + Melhorar_Janela))$, o que nós dá $O(n^4 + l(n \log n + fn^3))$ para o caso com sinal e $O(n^4 + l(n \log n + fn^2))$ para o caso sem sinal. Após algumas simplificações, concluímos que nossa implementação executa em tempo $O(n^4 + fln^3)$ e $O(n^4 + fln^2)$ para as versões com sinal e sem sinal, respectivamente. Nós destacamos que estas complexidades podem ser reduzidas substituindo o termo $O(n^4)$ por $O(n\sqrt{n \log n})$, se usarmos o algoritmo proposto por Tannier e Sagot [43] para obter uma solução para o problema de ordenação de reversões em permutações com sinal em vez do `GRIMM`.

A principal medida de qualidade usada em nossos experimentos é a diferença de custo entre a solução produzida por nossa implementação e a solução produzida produzida pelo algoritmo do `GRIMM`.

Para n no conjunto $\{10, 20, \ldots, 100\}$, geramos aleatoriamente 1000 permutações com sinal e 1000 permutações sem sinal. Depois disso, executamos nossa implementação. A Figura 3.9 mostra com qual frequência nossa abordagem melhora a solução inicial em permutações com sinal e a Figura 3.10 mostra o mesmo para permutações sem sinal. Observamos que ambas versões apresentam comportamento semelhante: nós conseguimos melhorar a solução inicial em mais do que 96% dos casos. Desta forma, permutações de tamanho 10 são o único grupo com menos de 96% de melhoria, sendo 75.2% para permutações com sinal e 76.7% para permutações sem sinal.

As Figuras 3.9 e 3.10 também mostram que obtemos melhoramento já nas primeiras iterações. Em 91.2% das permutações com sinal e e 86.2% das permutações sem sinal, o primeiro melhoramento acontece em até 50 iterações. Portanto é muito raro encontrar o primeiro melhoramento após 250 iterações, o que ocorreu em 0.8% e 0.6% dos casos em permutações com sinal e sem sinal respectivamente.

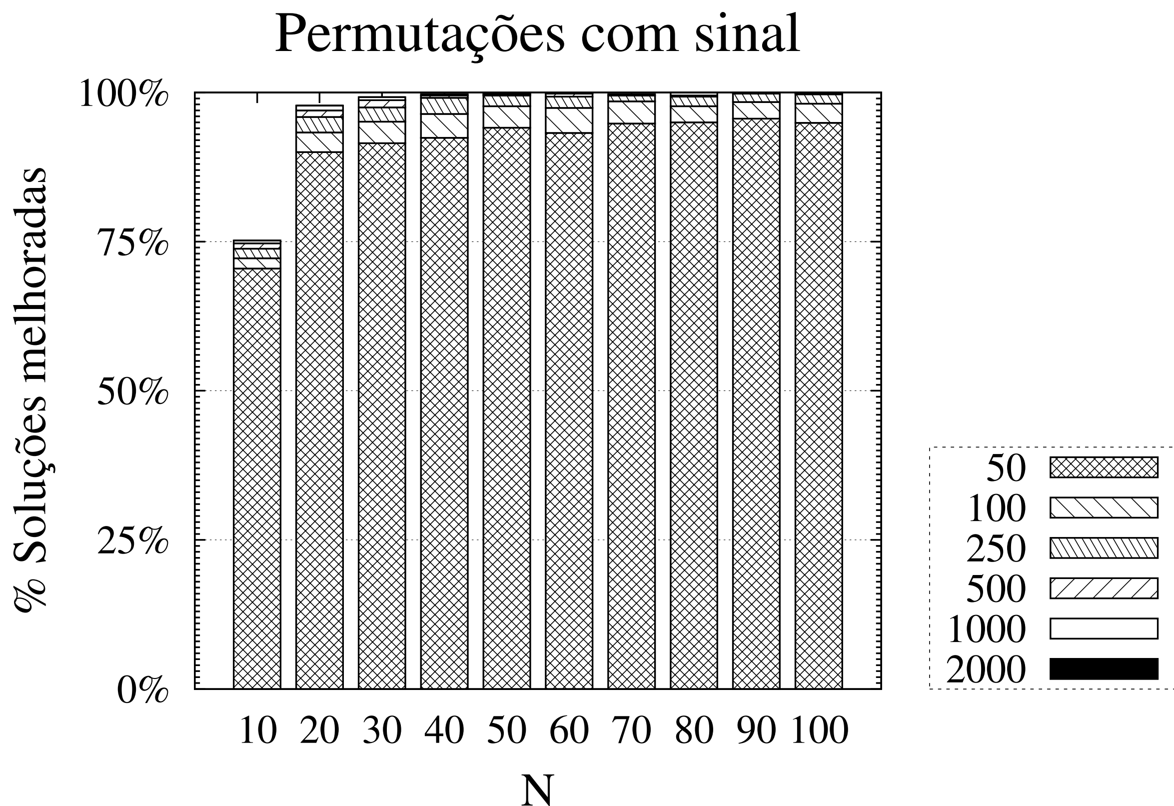

Figura 3.9: Este gráfico mostra a porcentagem de vezes que a nossa heurística melhorou a solução inicial em permutações com sinal. Em geral, a solução inicial foi melhorada em 97.1% dos casos. Se nós considerarmos somente permutações grandes, tal como $n \geq 50$, nós observamos que 99.9% das soluções iniciais são melhoradas. Em nosso gráfico, nós especificamos o intervalo de iterações que geraram o primeiro melhoramento na solução inicial, o que nos permitiu observar que 91.2% dos melhoramentos ocorreram com 50 iterações.

As Figuras 3.11 e 3.12 mostram a porcentagem de melhoramento médio usando nosso método. Seja $S_{inicial}$ a solução inicial e S_{final} a solução final produzida pela nossa abordagem *GRASP*, definimos o melhoramento como a diferença em custo entre $S_{inicial}$ e S_{final} dividido pelo custo de $S_{inicial}$. Em outras palavras, $\%_melhoramento = 100 \times \frac{cost(S_{inicial}) - cost(S_{final})}{cost(S_{inicial})}$.

Observamos que nossas soluções custam em média 17.9% e 16.6% menos do que a solução inicial para permutações com sinal e sem sinal, respectivamente. As figuras podem também ajudar na decisão de quantas iterações podem ser necessárias para encontrar boas soluções. Nossa implementação requer tempo aproximadamente linear em relação ao número de iterações, o que significa que executar 2000 iterações é aproximadamente 40 vezes mais lento do que executar somente 50 iterações.

Desta forma, podemos exemplificar para o caso com sinal, considerando que argumento similar poderia ser feito para o caso sem sinal. Para permutações com 10 elementos, nós obtemos uma solução que é 14.6% melhor do que a solução inicial quando executamos 50 iterações, e este número irá aumentar para 16.2% se nós executarmos 2000 iterações, o

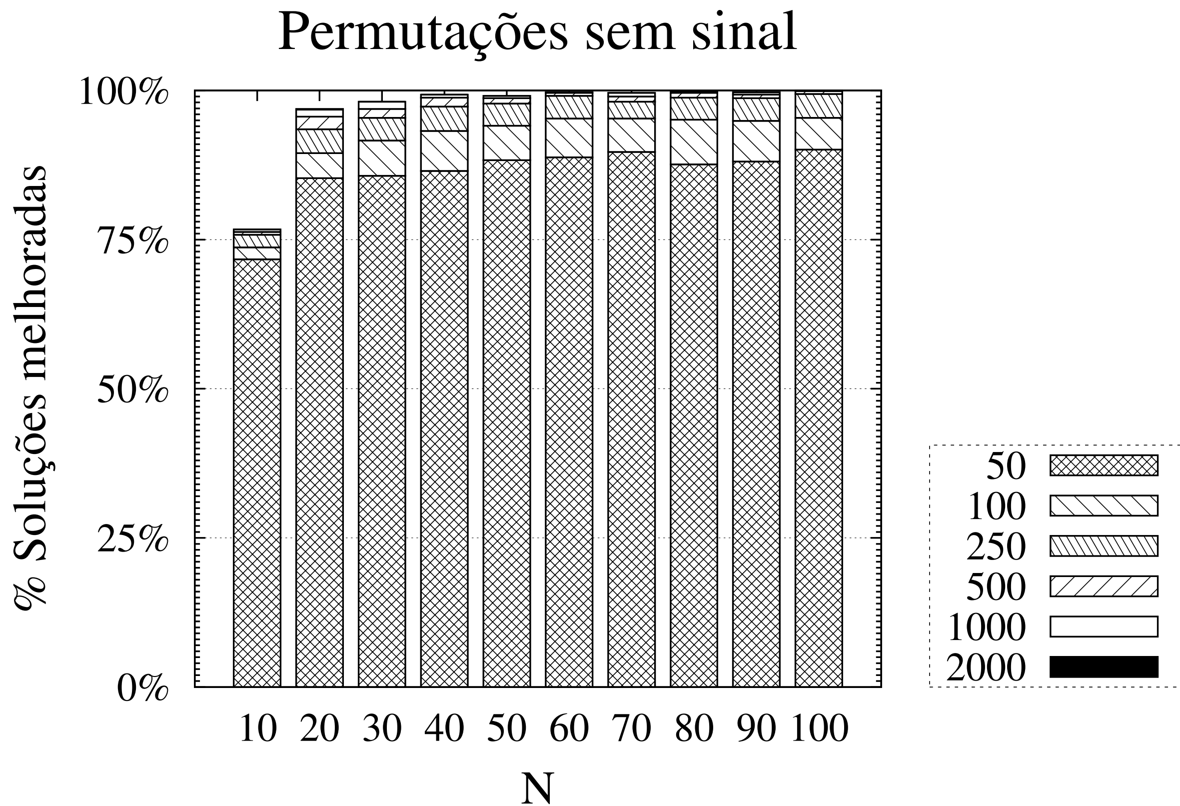

Figura 3.10: Este gráfico mostra a porcentagem de vezes que a nossa heurística melhora a solução inicial em permutações sem sinal. Em geral, a solução inicial foi melhorada em 96.8% dos casos. Se considerarmos somente permutações grandes como $n \geq 50$, observamos que 99.7% das soluções iniciais foram melhoradas. No nosso gráfico nós especificamos o intervalo de iterações que gerou o primeiro melhoramento na solução inicial, o que nos permite obserar que 86.2% dos melhoramentos ocorrem com 50 iterações.

que resulta em uma melhora relativa de 1.6% e um tempo de execução 40× mais lento.

Como esperado, esta análise muda conforme as permutações aumentam em tamanho. Para permutações com 50 elementos, são obtidas soluções 9.5% melhores do que as respectivas soluções iniciais depois de 50 iterações, 11.7% depois de 100 iterações, 14.5% depois de 250 iterações, 17.5% depois de 1000 iterações e 18.5% depois de 2000 iterações. Portanto, depois de 2000 iterações nós obtemos soluções quase duas vezes melhores do que as soluções obtidas quando executadas 50 iterações para permutações de tamanho 50.

Para permutações com 100 elementos, a solução após 2000 iterações é em média 2.3 vezes melhor do que aquela obtidas após 50 iterações pela mesma análise, o que é significante apesar da gradual diminuição da taxa $\frac{\%_of_improvement}{number_of_iterations}$.

Continuamos nossa análise comparando nossos resultados com algoritmos anteriores para o problema de ordenação por reversões ponderadas. No caso com sinal, Swidan *et al.* [40] criou um modelo sensível à ponderação onde o custo para reverter uma subsequência de tamanho l é dado pela função $f(l) = l^{\alpha}$, para $\alpha \geq 0$. Eles conseguiram garantir a taxa de aproximação $O(\lg n)$ quando $\alpha = 1$, o que é o mesma função de custo que usamos neste trabalho.

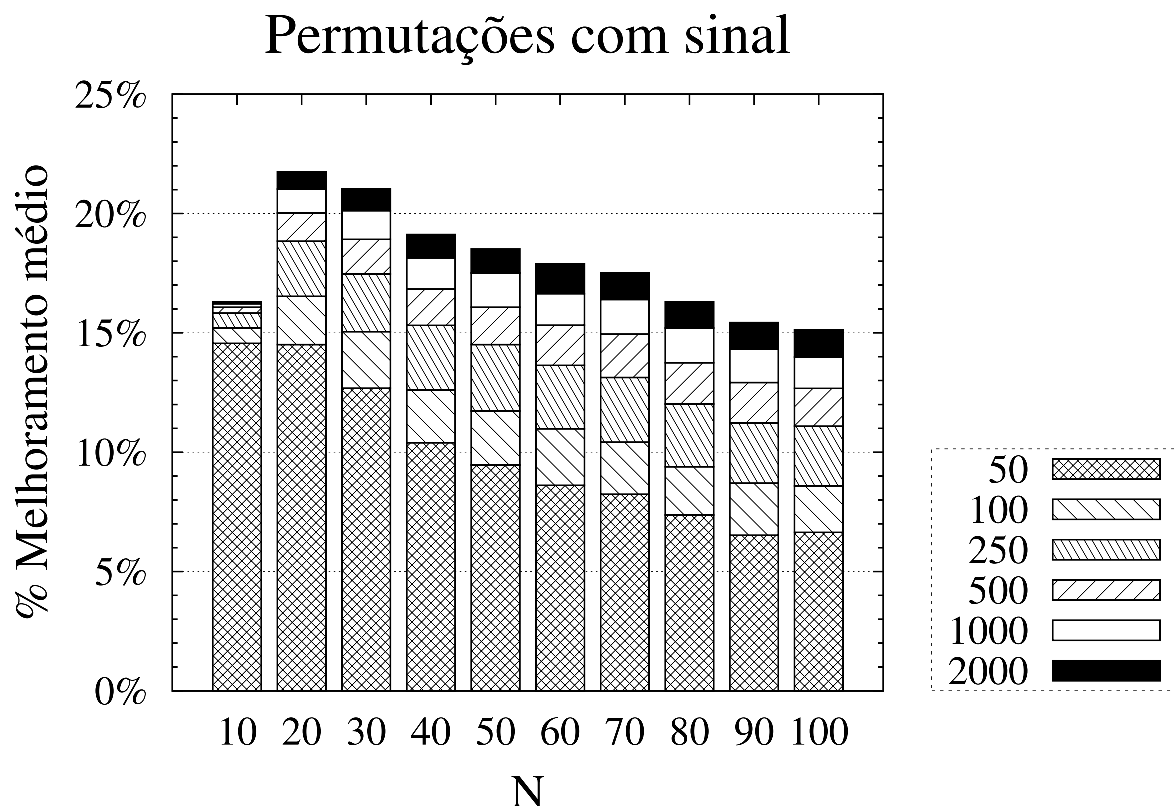

Figura 3.11: Este gráfico mostra a porcentagem de melhoria obtida usando nosso método em permutações com sinal. Observamos que nossas soluções custam por volta de 17.9% menos do que a solução inicial. O histograma também mostra que o melhoramento médio após um dado número de iterações ter sido executado. Observamos que a tendência de que mais iterações devem ser executadas conforme se aumenta o tamanho das permutações.

Para permutações sem sinal, Bender *et al.* [10] apresentaram um algoritmo com fator de aproximação $O(\lg n)$. Este é o melhor fator de aproximação conhecido. Nós também comparamos com a abordagem gulosa proposta por Arruda *et al.* [2] (descrita no Capítulo 2), que retorna bons resultados em uma abordagem prática. Algoritmos de aproximação para o problema de ordenação por reversões para permutações sem sinal fornecem soluções viáveis para nosso problema. Kececioglu e Sankoff [31], Christie [18] e Berman *et al.* [12] apresentaram algoritmos com fatores de aproximação de 2.0, 1.5 e 1.375, respectivamente. Nós conseguimos implementar os algoritmos propostos pelo primeiro e o segundo autores, o que nos levou a observar que o segundo produz soluções de menor custo. O algoritmo 1.375 proposto por Berman *et al.* é puramente teórico e nenhuma implementação prática é conhecida até o momento.

A Figura 3.13 e a Tabela 3.2 apresentam o custo médio para permutações com sinal. O rótulo `GRIMM` representa nossa solução inicial, `Swidan` representa o algoritmo proposto por Swidan *et al.* [40]. Como podemos observar, `Swidan` é superado por ambos `GRIMM` e a nossa abordagem (rotulada como `50`, `250`, e `2000`, que representa o número de iterações). Em média, nossa soluções depois de 2000 iterações custam 27.9% menos do que as soluções fornecidas por `Swidan`.

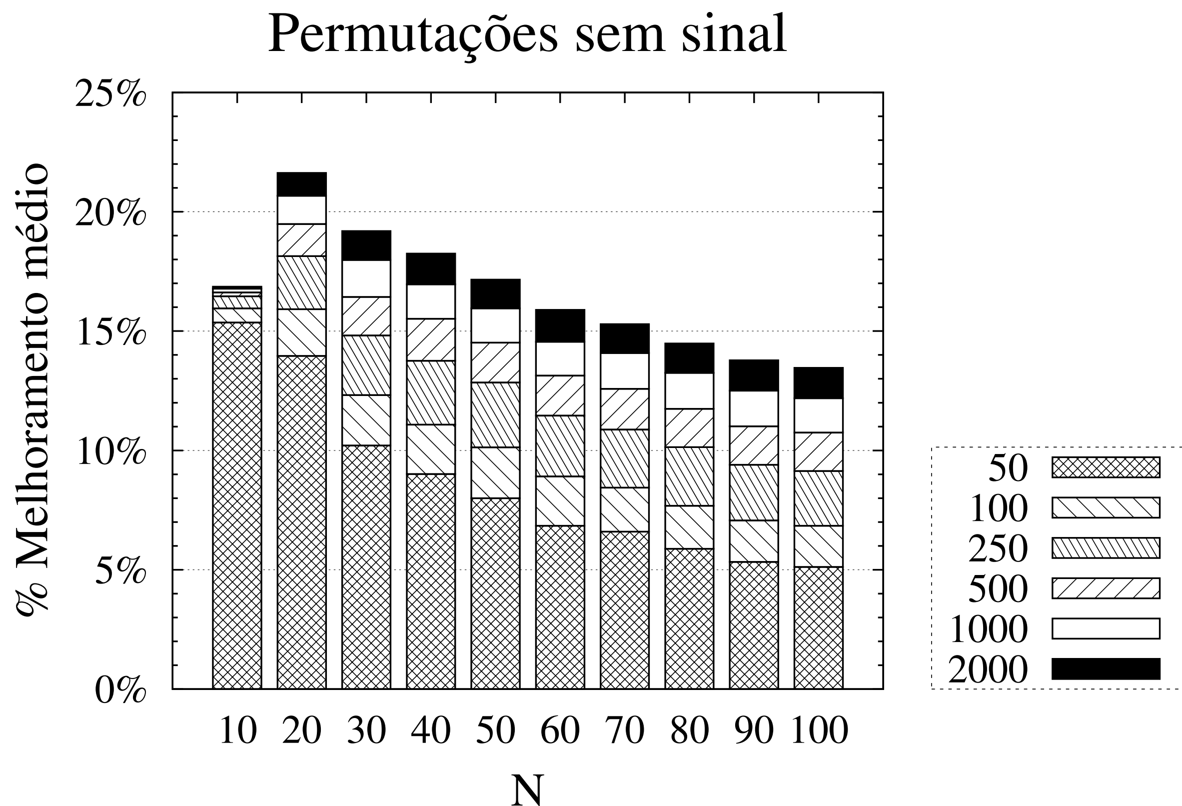

Figura 3.12: Este gráfico mostra a porcentagem de melhoramento obtida usando nosso método em permutações com sinal. Nós observamos que as soluções fornecidas pelo nosso método custam por volta de 16.6% menos do que a solução inicial. Em geral o comportamento mostrado aqui é similar ao caso com sinal.

A Figura 3.14 e a Tabela 3.3 apresentam o custo médio para permutações sem sinal. O rótulo `GRIMM` representa nossa solução inicial, `Bender` representa o algoritmo proposto por Bender *et al.* [10], `Christie` representa o algoritmo proposto por Christie [18] para o problema de ordenação por reversões, o rótulo `Arruda` representa uma abordagem gulosa proposta anteriormente por Arruda *et al.* [2] para o problema de ordenação por reversões ponderadas (descrita no Capítulo 2), e números indicam quantas iterações nossa abordagem executou para obter os respectivos resultados. Para permutações com 50 elementos ou menos, `Arruda` teve sucesso em encontrar resultados que são melhores do que nosso método depois de 50 iterações, mas em média nenhum método é melhor do que o nosso quando nós executamos mais de 250 iterações, independentemente do tamanho da permutação.

3.6 Publicações

O trabalho apresentado nesse capítulo resultou em duas publicações.

Uma publicação referente à primeira abordagem para construção de soluções, conforme descrita na Seção 3.4, foi apresentada na *1st International Conference on Algorithms for Computational Biology* (AlCoB'2014), realizada em Tarragona (Espanha), em julho de

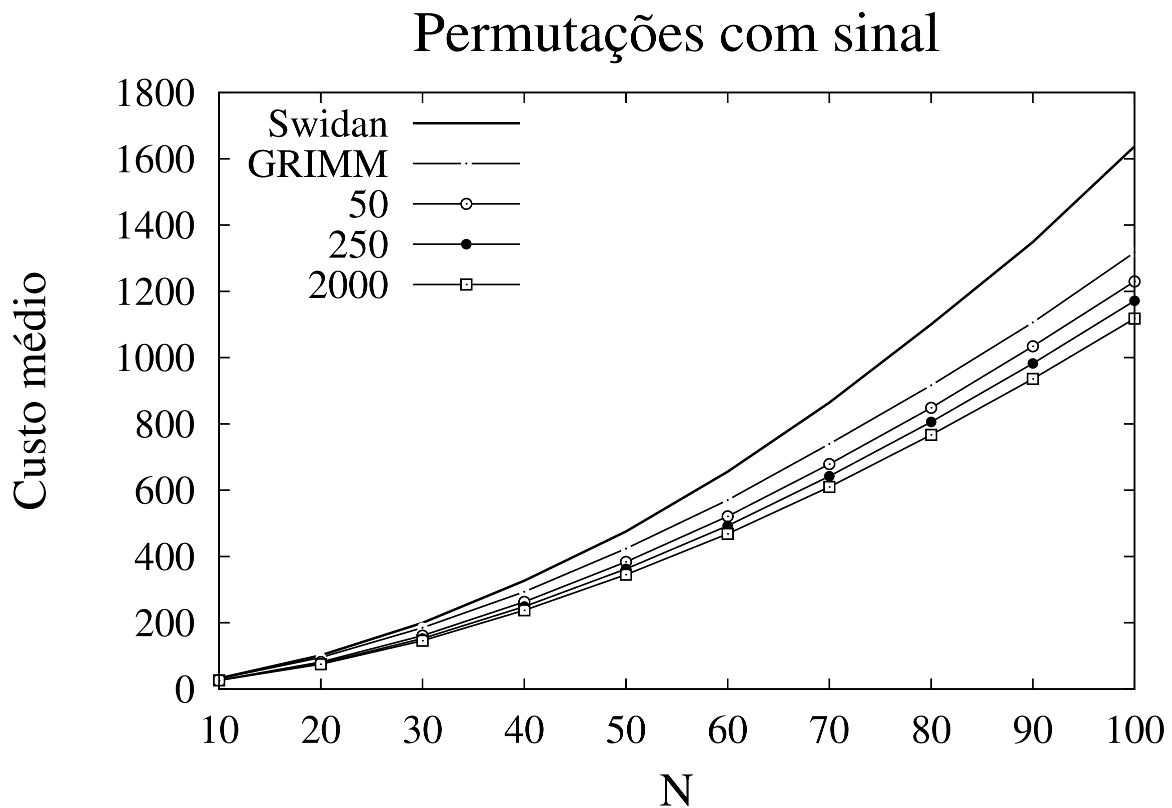

Figura 3.13: Este gráfico mostra uma análise comparativa entre três algoritmos considerando somente permutações com sinal. O eixo-Y representa o custo médio e o eixo-X representa o tamanho da permutação. O rótulo `Swidan` representa o algoritmo proposto por Swidan *et al.* [40]. `GRIMM` [44] fornece soluções ótimas para o problema de ordenação por reversões em que cada reversão possui custos unitários. Curvas rotuladas com números representam nossa abordagem, onde cada número indica o número de iterações que usamos.

2014, com o título 'Heuristics for the Sorting by Length-Weighted Inversions Problem on Signed Permutations"(Thiago Arruda, Ulisses Dias e Zanoni Dias) [3].

A outra publicação é referente à segunda abordagem que utilizamos para a construção de soluções. O trabalho foi publicado no periódico *IEEE/ACM Transactions on Computational Biology and Bioinformatics* com o título "A GRASP-based Heuristic for the Sorting by Length-Weighted Inversions Problem" (Thiago Arruda, Ulisses Dias e Zanoni Dias) [4].

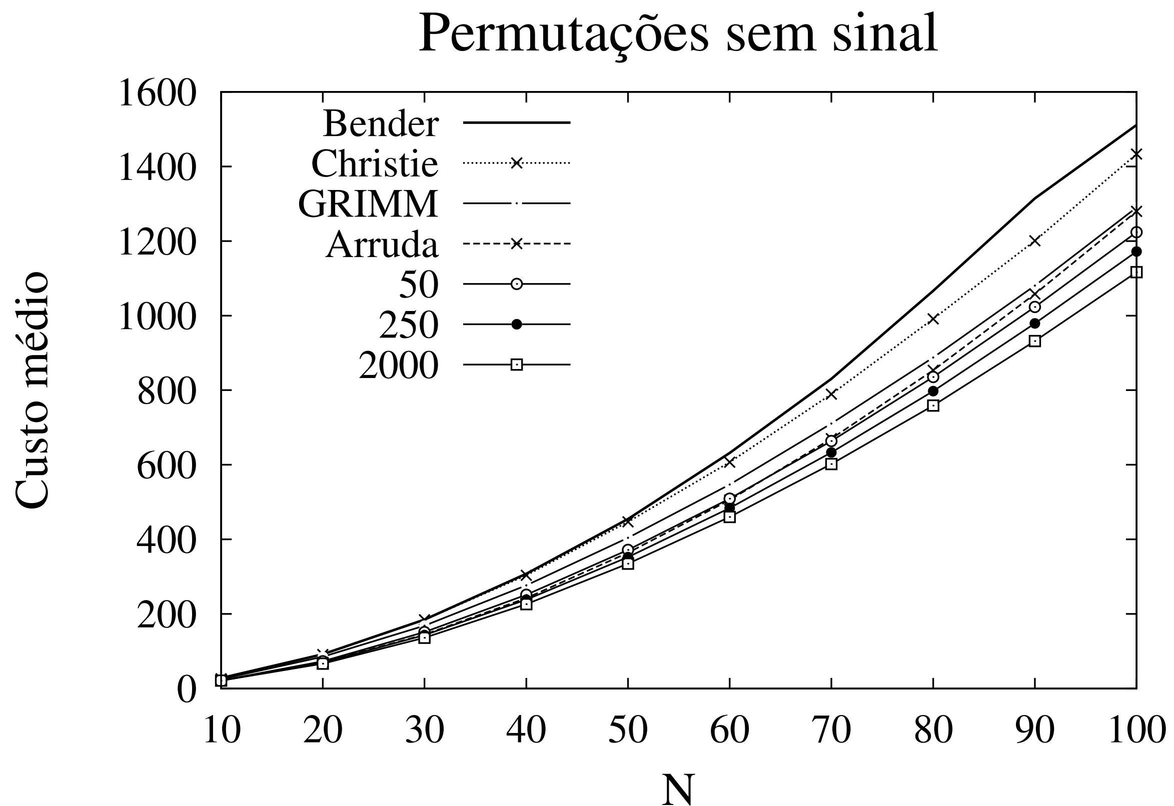

Figura 3.14: Este gráfico mostra uma análise comparativa entre cinco algoritmos. O eixo-Y representa o custo médio e o eixo-X representa o tamanho da permutação. O rótulo `GRIMM` representa a solução inicial, `Bender` representa o algoritmo proposto por Bender *et al.* [10], `Christie` representa o algoritmo proposto por Christie [18] para ordenação de reversões de custos unitários, o rótulo `Arruda` representa o método guloso previamente proposto por Arruda *et al.* [2] para ordenação por reversões ponderadas, e os números indicam quantas iterações foram executas para serem obtidos os resultados apresentados.

N	GRIMM	50	250	2000	Swidan
10	31.19	26.60	26.25	26.10	32.60
20	95.55	81.58	77.63	74.77	101.75
30	184.81	162.06	153.13	146.35	199.58
40	293.73	263.84	248.82	236.01	326.84
50	423.84	385.08	363.27	343.62	475.55
60	570.36	523.73	494.52	467.79	656.07
70	739.57	684.01	647.96	613.07	864.46
80	916.26	856.95	810.57	769.55	1100.70
90	1106.28	1040.08	989.94	939.39	1348.72
100	1317.37	1244.56	1185.41	1124.93	1637.28

Tabela 3.2: Esta tabela mostra uma análise comparativa do custo médio entre algoritmos para a variação do problema para permutações com sinal. Esta tabela mostra resultados para permutações de tamanho 10 até 100. Os números 50, 250 e 2000 indicam quantas iterações do nosso método foram executas para serem obtidos os respectivos resultados. O rótulo `Swidan` representa o algoritmo proposto por Swidan *et al.* [40].

N	GRIMM	50	250	2000	Arruda	Kececioglu	Bender	Christie
10	25.37	21.56	21.17	21.07	21.57	29.93	27.74	26.25
20	85.04	73.20	69.75	66.76	70.09	109.56	91.72	90.84
30	168.46	151.73	143.88	136.33	144.65	238.80	184.82	184.30
40	276.43	251.91	238.79	226.08	242.26	414.84	307.95	303.85
50	404.10	371.81	353.01	335.30	363.93	646.77	454.64	446.75
60	546.78	518.08	490.12	463.52	505.43	918.55	631.61	606.75
70	710.57	665.17	634.84	601.29	669.85	1242.02	830.42	789.47
80	887.51	836.22	798.45	758.61	853.22	1624.55	1066.59	991.08
90	1080.68	1018.84	974.71	929.56	1057.77	2043.78	1314.31	1200.97
100	1290.04	1221.65	1172.61	1118.45	1279.69	2523.35	1511.02	1433.35

Tabela 3.3: Esta tabela mostra uma análise comparativa do custo médio entre algoritmos para a variação do problema para permutações sem sinal. Esta tabela mostra resultados para permutações de tamanho 10 até 100. `GRIMM` é o rótulo para solução inicial. Os números 50, 250 e 2000 indicam quantas iterações do nosso método foram executas para serem obtidos os respectivos resultados. Arruda [2] é o rótulo para os resultados do nosso primeiro trabalho para o problema. O rótulo `Kececioglu` representa o algoritmo 2-aproximado proposto por Kececioglu [31]. O rótulo `Swidan` representa o algoritmo proposto por Swidan *et al.* [40].

Capítulo 4

Conclusões

No Capítulo 2 foi apresentado um novo método para o *problema de ordenação de permutações sem sinal por reversões ponderadas*. Este modelo considera o caso onde a ponderação de reversões é igual ao número de elementos compreendidos no segmento abrangido pela reversão. Nosso método é baseado em uma abordagem gulosa que leva em consideração o número de *breakpoints*, o número de inversões e o nível de entropia. Fizemos um estudo para a escolha de configurações de parâmetros para o algoritmo. Analisamos como estes parâmetros afetam o comportamento do método. Para isto foram escolhidas várias configurações de parâmetros e executadas em todas as possíveis permutações de tamanho pequeno (com tamanho menor ou igual a 10), e também em amostras de permutações grandes (com tamanho entre 10 e 100). Em ambos os casos, é mostrado que o método deste trabalho apresenta soluções que são menos custosas do que métodos previamente existentes na literatura. Mostramos também que é necessário executar apenas um pequeno número de configurações de parâmetros, escolhidos aleatoriamente, para se obter bons resultados.

Já no Capítulo 3, apresentamos uma meta-heurística GRASP para as versões com sinal e sem sinal do problema. Nosso modelo tem alguns parâmetros que podem ser usados e configurados facilmente. Os parâmetros impactam o tempo de processamento e a qualidade das soluções. Construímos um conjunto de testes com mais de 20000 instâncias (considerando permutações sem sinal e com sinal), então nós definimos os parâmetros de forma a fazer a heurística executar rapidamente. Com esta configuração, nós conseguimos obter melhores resultados do que outras abordagens previamente disponíveis na literatura e também conseguimos melhorar as soluções iniciais com uma margem significativa. Conseguimos melhorar a solução inicial em mais de 96% dos casos durante nossos experimentos. Constatamos que em geral, quanto maior o tamanho da permutação, mais provável torna-se obter uma melhoria da solução inicial. Além disso, nossa solução custa 17.9% e 16.6% menos do que a solução inicial para permutações com sinal e sem sinal, respectivamente.

A pesquisa desenvolvida neste trabalho resultou em 3 artigos: dois foram apresentados em conferências e um foi publicado em um periódico. Os resultados do Capítulo 2 foram apresentados na conferência ACM BCB'2013 (Conference on Bioinformatics, Computational Biology and Biomedical Informatics), realizada em Washington (EUA), em setembro de 2013, sob o título "Heuristics for the Sorting by Length-Weighted Inversion

Problem" (Thiago Silva Arruda, Ulisses Dias e Zanoni Dias) [2]. O conteúdo apresentado no Capítulo 3 resultou em duas publicações. A primeira foi apresentada na *1st International Conference on Algorithms for Computational Biology* (AlCoB'2014), realizada em Tarragona (Espanha), em julho de 2014, com o título "Heuristics for the Sorting by Length-Weighted Inversions Problem on Signed Permutations"(Thiago Arruda, Ulisses Dias e Zanoni Dias) [3]. A segunda foi publicada no periódico *IEEE/ACM Transactions on Computational Biology and Bioinformatics* com o título "A GRASP-based Heuristic for the Sorting by Length-Weighted Inversions Problem" (Thiago Arruda, Ulisses Dias e Zanoni Dias) [4].

Referências Bibliográficas

[1] R. M. Aiex, S. Binato, and M. G. C. Resende. Parallel GRASP with path-relinking for job shop scheduling. *Parallel Computing*, 29:2003, 2002.

[2] T. S. Arruda, U. Dias, and Z. Dias. Heuristics for the sorting by length-weighted inversion problem. In *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, pages 498–507. ACM, 2013.

[3] T. S. Arruda, U. Dias, and Z. Dias. Heuristics for the sorting by length-weighted inversions problem on signed permutations. In *Algorithms for Computational Biology -1st International Conference on Algorithms for Computational Biology, AlCoB'2014, Tarragona, Spain*, pages 59–70, 2014.

[4] T. S. Arruda, U. Dias, and Z. Dias. A GRASP-based heuristic for the sorting by length-weighted inversions problem. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, PP(99):1–1, 2015.

[5] T. Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms.* Oxford University Press, Oxford, UK, 1996.

[6] D. A. Bader, B. M. E. Moret, and M. Yan. A Linear-Time Algorithm for Computing Inversion Distance between Signed Permutations with an Experimental Study. *Journal of Computational Biology*, 8:483–491, 2001.

[7] M. Bader and E. Ohlebusch. Sorting by weighted reversals, transpositions, and inverted transpositions. In *Proceedings of the 10th annual international conference on Research in Computational Molecular Biology*, RECOMB2006, pages 563–577. Springer-Verlag, 2006.

[8] V. Bafna and P. A. Pevzner. Genome Rearrangements and Sorting by Reversals. *SIAM Journal of Computing*, 25(2):272–289, February 1996.

[9] M. A. Bender, D. Ge, S. He, H. Hu, R. Y. Haodong, R. Y. Pinter, S. Skiena, , and F. Swidan. Improved bounds on sorting by length-weighted reversals. *Journal of Computer and System Sciences*, 74(5):744–774, 2008.

[10] M. A. Bender, D. Ge, S. He, H. Hu, R. Y. Pinter, S. Skiena, and F. Swidan. Improved bounds on sorting by length-weighted reversals. *Journal of Computer and System Sciences*, 74(5):744 – 774, 2008.

[11] A. Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. *Discrete Applied Mathematics*, 146:134–145, 2005.

[12] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-approximation algorithm for sorting by reversals. In *Proceedings of the 10th European Symposium on Algorithms, ESA'2002*, pages 200–210, Rome, Italy, 2002.

[13] M. Blanchette, T. Kunisawa, and D. Sankoff. Parametric genome rearrangement. *Gene*, 172(1):C11–17, 1996.

[14] R. G. Cano, G. Kunigami, C. C. Souza, and P. J. Rezende. A hybrid GRASP heuristic to construct effective drawings of proportional symbol maps. *Computers and Operations Research*, 40(5):1435 – 1447, 2013.

[15] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the first annual international conference on Computational molecular biology*, RECOMB'1997, pages 75–83, New York, NY, USA, 1997. ACM.

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

[17] 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, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

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

[19] A. E. Darling, I. Miklós, and M. A. Ragan. Dynamics of genome rearrangement in bacterial populations. *PLoS Genetics*, 4(7):e1000128, 2008.

[20] U. Dias, C. Baudet, and Z. Dias. Greedy Randomized Search Procedure to Sort Genomes using Symmetric, Almost-Symmetric and Unitary Inversions. In *Proceedings of the 4th ACM Conference on Bioinformatics, Computational Biology and Biomedical Informatics, ACM BCB'2013*, pages 181–190, Maryland, United States, September 2013.

[21] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[22] T. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[23] T. A. Feo and M. Pardalos. A greedy randomized adaptive search procedure for the 2-partition problem. *Operations Research*, 1994.

[24] P. Festa and M.G.C. Resende. GRASP: basic components and enhancements. *Telecommunication Systems*, 46(3):253–271, 2011.

[25] J. Fischer and S. W. Ginzinger. A 2-approximation algorithm for sorting by prefix reversals. In *Proceedings of the 13th annual European conference on Algorithms*, ESA'2005, pages 415–425, Berlin, Heidelberg, 2005. Springer-Verlag.

[26] W. H. Gates and C. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, pages 47–57, 1979.

[27] S. Hannenhalli, C. Chappey, E. Koonin, and P. Pevzner. Scenarios for genome rearrangements: Herpesvirus evolution as a test case. *Proceedings of 3rd International Conference on Bioinformatics and Complex Genome Analysis*, 1994.

[28] 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.

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

[30] M. H. Heydari and I. Hal Sudborough. On the Diameter of the Pancake Network. *Journal of Algorithms*, 25(1):67–94, 1997.

[31] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement, 1995.

[32] E. B. Knox, S. R. Downie, and J. D. Palmer. Chloroplast genome rearrangements and evolution of giant lobelias from herbaceous ancestors. *Molecular Biology and Evolution*, 10:414–430, 1993.

[33] J. F. Lefebvre, N. El-Mabrouk, E. Tillier, and D. Sankoff. Detection and validation of single gene inversions. *Bioinformatics*, 19 Suppl 1:i190–196, 2003.

[34] R. Y. Pinter and S. Skiena. Genomic sorting with length-weighted reversals. *Genome Informatics*, 13:2002, 2002.

[35] C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A hybrid GRASP with perturbations for the steiner problem in graphs. *INFORMS Journal on Computing*, 14:200–2, 2001.

[36] E. P. C. Rocha. An appraisal of the potential for illegitimate recombination in bacterial genomes and its consequences: From duplications to genome reduction. *SIAM Journal on Computing*, Genome Res., 13(6a):1123–1132, 2003.

[37] D. Sankoff. Short inversions and conserved gene cluster. *Bioinformatics*, 18(10):1305–1308, 2002.

[38] D. Sankoff, J. F. Lefebvre, E. Tillier, A. Maler, and N. El-Mabrouk. The distribution of inversion lengths in bacteria. In *Comparative Genomics*, volume 3388 of *Lecture Notes in Computer Science*, pages 97–108. Springer Berlin Heidelberg, 2005.

[39] A. C. Siepel. An algorithm to enumerate all sorting reversals. In *Proceedings of the Sixth Annual International Conference on Computational Biology*, RECOMB '02, pages 281–290, New York, NY, USA, 2002. ACM.

[40] F. Swidan, M. A. Bender, D. Ge, S. He, H. Hu, and R. Pinter. Sorting by length-weighted reversals: Dealing with signs and circularity. In SuleymanCenk Sahinalp, S. Muthukrishnan, and Ugur Dogrusoz, editors, *Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 32–46. Springer Berlin Heidelberg, 2004.

[41] F. Swidan, M. A. Bender, D. Ge, S. He, H. Hu, and R. Y. Pinter. Sorting by length-weighted reversals: Dealing with signs and circularity. *Lecture Notes in Computer Science*, 3109:32–46, 2008.

[42] E. Tannier, A. Bergeron, and M-F. Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155(6-7):881–888, April 2007.

[43] E. Tannier and M.-F. Sagot. Sorting by reversals in subquadratic time. In *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching, CPM'2004*, pages 1–13, Istambul, Turkey, 2004.

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