COPPE
UFRJ

DISTÂNCIA DE TRANSPOSIÇÃO ATRAVÉS DA TRANSFORMAÇÃO EM PERMUTAÇÃO SIMPLES

Marcelo Pereira Lopes

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Celina Miraglia Herrera de Figueiredo
Luis Antonio Brasil Kowada

Rio de Janeiro
Fevereiro de 2011

DISTÂNCIA DE TRANSPOSIÇÃO ATRAVÉS DA TRANSFORMAÇÃO EM PERMUTAÇÃO SIMPLES

Marcelo Pereira Lopes

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Celina Miraglia Herrera de Figueiredo, D.Sc.

Prof. Luis Antonio Brasil Kowada, D.Sc.

Prof. Marta Lima de Queirós Mattoso, D.Sc.

Prof. Helena Cristina da Gama Leitão, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
FEVEREIRO DE 2011

Lopes, Marcelo Pereira

Distância de Transposição Através da Transformação em Permutação Simples/Marcelo Pereira Lopes. – Rio de Janeiro: UFRJ/COPPE, 2011.

XII, 62 p.: il.; 29, 7cm.

Orientadores: Celina Miraglia Herrera de Figueiredo
Luis Antonio Brasil Kowada

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2011.

Referências Bibliográficas: p. 59 – 62.

1. Biologia Computacional. 2. Rearranjo de Genomas. 3. Permutação. 4. Transposição. 5. Algoritmos Aproximativos. 6. Complexidade de Tempo. I. Figueiredo, Celina Miraglia Herrera de *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

À minha família.

Agradecimentos

A meus pais e meu irmão, por todo apoio durante todos os momentos da minha vida, os meus mais sinceros agradecimentos.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo suporte financeiro. Aos meus orientadores, Celina e Luis Antonio, por tudo que aprendi com eles. Aprendizado que transcendeu os limites da pesquisa científica.

Ao Programa de Engenharia de Sistemas e Computação (PESC), por sua excêlencia acadêmica e por todos os benefícios que me foram concedidos tais como auxílio a viagens e palestras.

Ao Grupo de Grafos e Algoritmos da UFRJ, pelo apoio incondicional prestado a minha pessoa. Aos meus amigos da UFRJ, com quem aprendi muito. Em especial aos meus amigos Hélio Macêdo e Rodrigo Hausen pela grande ajuda durante este trabalho. Agradeço também a Marcela Borges, Bruno Cesar Barbosa, Marcelino Campos de Oliveira, André Ribeiro, Caroline Reis, Diana Sasaki, Hebert Coelho e Hugo Nobrega pela ajuda durante esta caminhada.

Aos membros da banca desta dissertação: Marta Mattoso e Helena Leitão por terem aceitado participar da defesa desta dissertação.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

DISTÂNCIA DE TRANSPOSIÇÃO ATRAVÉS DA TRANSFORMAÇÃO EM PERMUTAÇÃO SIMPLES

Marcelo Pereira Lopes

Fevereiro/2011

Orientadores: Celina Miraglia Herrera de Figueiredo
Luis Antonio Brasil Kowada

Programa: Engenharia de Sistemas e Computação

Biologia Computacional é uma área da Ciência da Computação que tem por objetivo o estudo e aplicação de técnicas e ferramentas computacionais aos problemas da Biologia Molecular. Dentre os problemas pesquisados, encontra-se o de evolução molecular, onde são estudados métodos para comparar sequencias de espécies distintas, baseados em eventos mutacionais. Estes métodos geram medidas de distância, que podem ser empregadas para verificar o relacionamento em termos evolutivos entre dois organismos. Uma técnica de computar distância é comparar blocos, formados por um ou mais genes, de genomas de dois organismos. Neste trabalho implementamos uma estrutura de dados proposta por Feng e Zhu, chamada Permutation Tree, que melhora o tempo e complexidade de execução para se realizar transposições em uma permutação. O algoritmo 1,5-aproximativo de Hartman e Shamir para ordenação de uma permutação por transposições possui complexidade de tempo $O(n^{3/2}\sqrt{\log n})$. Implementaremos o algoritmo com complexidade de tempo $O(n \log n)$, utilizando a estrutura Permutation Tree.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

TRANSPOSITION DISTANCE THROUGH TRANSFORMATION INTO SIMPLE PERMUTATION

Marcelo Pereira Lopes

February/2011

Advisors: Celina Miraglia Herrera de Figueiredo
Luis Antonio Brasil Kowada

Department: Systems Engineering and Computer Science

Computational Biology is an area that aims to study and to apply techniques and computational tools to problems of molecular biology. One of these problems is molecular evolution, in which methods are proposed for comparing sequences of distinct species, based on mutational events. These methods generate distance measures that could be employed to verify the evolutionary relationship between two organisms. A technique to compute distance is to compare blocks, composed by one ore more genes, of the genomes of two organisms. In this work we propose a new implementation of a recent data structure, called the permutation tree, to improve the running time of sorting permutations by transpositions. The existing 1.5-approximation algorithm for sorting permutations by transpositions has time complexity $O(n^{3/2}\sqrt{\log n})$. Using the permutation tree, we implement this algorithm in time complexity $O(n \log n)$.

Sumário

Lista de Figuras

Lista de Algoritmos

Capítulo 1

Introdução

Desde a descoberta da estrutura do DNA em 1953 por Watson e Crick [30], grande quantidade de informações sobre sequenciamento de macromoléculas (ácidos nucléicos e proteínas) de muitos organismos foram geradas. Isso nos permite encontrar as sequências de genomas completos de diversos organismos [18], exigindo técnicas complexas de análise destes dados. Neste contexto, surgiu uma área de pesquisa relativamente recente denominada **Bioinformática**. Esta é uma área onde são usados métodos e técnicas de Matemática e Ciência de Computação para resolver problemas da Biologia Molecular [25]. O termo bioinformática foi criado por Paulien and Ben Hesper em 1978 para estudantes de informática interessados em processar dados biológicos [18]. Seu estudo compreende o campo da genômica e da genética, principalmente da área genômica, envolvendo sequenciamento de DNA em larga escala. São utilizados algoritmos, banco de dados e técnicas computacionais para analisar os dados. O cerne da bioinformática compreende o estudo dos processos de replicação, transcrição e tradução do material genético e a regulação desses processos. Em 1975, F. Sanger, S. Nicklen e A. R. Coulson desenvolveram o primeiro método de sequenciamento rápido do DNA, chamado *chain-termination method*. Em 1976-1977, A. Maxan e W. Gilbert desenvolveram um método menos tóxico utilizando modificação química.

Dentro deste contexto estudaremos um método de comparação das cadeias de DNA (ou RNA) de espécies distintas, a partir do qual é possível identificar suas diferenças e similaridades, o que pode ser usado na reconstrução de filogenias. A análise das diferenças entre espécies provocadas por eventos de rearranjo em genomas

é a técnica mais adequada quando ocorrerem nos genomas eventos de mutação de grandes blocos de bases, ao contrário de operações pontuais sobre as bases [23, 24].

Na última década com o advento de técnicas de sequenciamento rápido de genomas, se investiu muito na área de Rearranjo de Genomas, tendo como objetivo extrair informações biológicas relevantes. Um modo de estruturar essas informações é através da comparação de genomas.

Rearranjo de Genomas, é uma área que basicamente visa a computar uma distância entre genomas de dois organismos de espécies distintas, comparando blocos formados por um ou mais genes. De forma genérica, o problema de distância de rearranjo é encontrar a menor sequência de eventos de rearranjo (mutações) necessários para transformar um genoma em outro.

Esta dissertação contribui para o estudo do problema da distância de transposições cuja complexidade foi mostrada ser NP-Completo por Bulteau, Fertin e Rusu [5]. No decorrer do capítulo definimos rearranjo de genomas por transposições e contextualizamos o estudo realizado. Descrevemos sobre os problemas da ordenação por transposição, dando maior ênfase ao problema da distância de transposição. No Capítulo 2 definimos o grafo de realidade e desejo proposto por Bafna e Pevzner [2] para simplificar o estudo de rearranjo. Definimos o que é uma permutação simples. Este tipo de permutação é utilizado para implementar o algoritmo $1,5$-aproximativo de Hartman e Shamir [17] para o problema de ordenar uma permutação por transposições. Iremos neste capítulo estudar o algoritmo e posteriormente implementá-lo. Por último, introduzimos a estrutura de dados árvore de permutação proposta por Feng e Zhu [11]. No Capítulo 3 implementamos o algoritmo de Hartman e Shamir [17] utilizando a estrutura proposta por Feng e Zhu [11] e propomos uma otimização no algoritmo implementado. O Capítulo 4 conclui a dissertação enfatizando as contribuições e a originalidade deste trabalho.

1.1 Rearranjo de Genomas

Genoma é o conjunto completo de cromossomos dentro de uma célula. Sendo sua representaÃ§Ã£o destacada pela figura 1.1. O número de cromossomos em um genoma é característica de uma espécie. Por exemplo, cada célula humana possui 46

cromossomos organizados em 23 pares de cromossomos. O genoma é toda a informação hereditária de um organismo que está codificada em seu DNA. O DNA, ácido desoxirribonucleico, é um composto orgânico cujas moléculas contêm as instruções genéticas que coordenam o desenvolvimento e funcionamento de todos os seres vivos e alguns vírus. O DNA é uma cadeia composta por duas fitas, cada fita é uma sequência de unidades básicas compostas por um açúcar (desoxirribose), um fosfato e uma base nitrogenada: adenina (A), guanina (G), citosina (C) e timina (T). A estrutura dessas unidades básicas induz uma orientação na cadeia de DNA. As duas fitas do DNA, que obedecem a uma estrutura de dupla hélice, são mantidas juntas pelas conexões (pontes de hidrogênio) entre os pares de bases: A - T e C - G. As bases A e T, bem como C e G, são chamadas bases complementares entre si. Um gene é um segmento, não necessariamente contínuo, em uma das fitas do DNA que codifica informação para a construção de uma proteína.

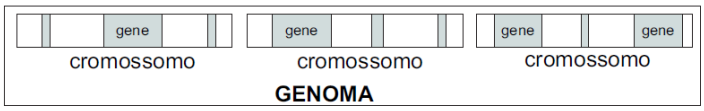

Figura 1.1: Visão esquemática dos genes dentro dos cromossomos e dos cromossomos dentro do genoma. A região hachurada indica um gene.

O problema em rearranjo de genomas pode ser formulado da seguinte forma: dados dois genomas, encontrar o número de eventos evolucionários que transformam um no outro. Este número de mutações, significa a distância entre as duas espécies.

Um gene ou bloco de genes ao evoluir muda a sua posição no cromossomo de diversas maneiras: O modo comum é a reversão ou inversão, logo depois vem a transposição, que é um tipo especial de movimentação de blocos contíguos. Uma reversão inverte a ordem e a orientação de um bloco de genes em um cromossomo do genoma. Uma transposição remove um bloco de genes de um cromossomo e insere-o em uma nova posição no mesmo cromossomo [13].

Foram realizados diversos trabalhos procurando determinar distâncias de rearranjo de genomas [10, 14, 20, 29]. Para simplificar o problema, estudaram os rearranjos em separado, uns optaram por estudar o rearranjo por reversões e outros por transposições, que são respectivamente o menor número de reversões e transposi-

ções de um genoma para o outro. Até os dias de hoje não se conhece uma maneira exata de calcular a distância de transposição [1, 9]. Mesmo a estratégia de se adicionar informação sobre orientação aos elementos de uma permutação [28], como feito no problema de ordenação por reversões, não deu origem ainda a nenhum algoritmo exato. Além disso, ainda há uma grande diferença entre os limites superior e inferior demonstrados para a distância de transposição [19].

Rearranjo de genomas são modelados por meio de permutações de blocos de genes. Usando o conceito de reversão e transposição, que são ambos problemas de ordenação por meio de operações (reversão e transposição) nos cromossomos, conseguiremos a ordenação. É importante notar que sempre é possível transformar um cromossomo em outro com o mesmo conjunto de genes por meio de reversões. Isso também vale para transposições caso os genes possuam a mesma orientação em ambos os cromossomos [13].

O problema de distância de reversão já foi muito bem explorado. Se desconsiderarmos a orientação dos genes, o problema foi provado NP-difícil por Caprara [6] e surgiram muitos algoritmos aproximativos [1, 4, 15]. Por outro lado, considerando a orientação dos genes, o problema pode ser resolvido em tempo polinomial pelo algoritmo de Hannenhalli e Pevzner [15]. Trabalhos posteriores melhoraram o tempo de execução do algoritmo e simplificaram a teoria [3, 10].

1.2 Contextualizando o estudo de rearranjo por transposição

Os primeiros a estudarem e realizarem trabalhos sobre rearranjo por transposições foram Bafna e Pevzner [2]. Eles propuseram uma estrutura denominada grafo-de-ciclos, esta estrutura é gerada a partir de permutações. É criado um grafo com várias propriedades interessantes, podendo visualizar através de suas arestas tanto a permutação original como a permutação desejada. Devido a este motivo a estrutura também é conhecida como diagrama de realidade e desejo. Para resolver o problema de ordenação por transposições, Bafna e Pevzner estudaram propriedades de entrelaçamento de ciclos no grafo-de-ciclos e propuseram algoritmos aproximativos com razões 2,0, 1,75 e 1,5. A complexidade de tempo do algoritmo de razão 1,5 é

$O(n^2)$, onde n é o tamanho da permutação.

Christie [7], utilizando da mesma estrutura proposta por Bafna e Pevzner (grafo-de-ciclos) propôs um algoritmo aproximativo, com razão de aproximação 1,5. Este algoritmo proposto tem a complexidade de tempo $O(n^4)$, sendo n o tamanho da permutação.

Walter, Dias e Meidanis [26] contribuíram com o problema apresentando uma estrutura simples, o diagrama de pontos-de-quebra e fizeram um algoritmo com fator de aproximação 2,25 baseado na estrutura. Walter, Dias e Meidanis comentaram que embora existam algoritmos com fator de aproximação melhores, estes são baseados em estruturas mais complexas de dados. A complexidade de tempo deste algoritmo é $O(b(\pi)^2)$, onde $b(\pi)$ é o número de pontos-de-quebra no diagrama.

Walter, Curado e Oliveira [27] estudaram o algoritmo proposto por Christie [7] e observaram que os passos do algoritmo poderiam ser executados em $O(n^2)$, observaram que Christie ao calcular triplas fortemente orientadas em um ciclo utilizava na execução ordem de $O(n^3)$. Portanto, a complexidade de tempo do algoritmo de Christie era determinada pelo tempo de obter triplas fortemente orientadas. Walter, Curado e Oliveira reduziram a complexidade de tempo do algoritmo de Christie para $O(n^2)$ propondo um novo algoritmo para estas triplas.

Dias e Meidanis [8] trabalharam com permutações prefixadas, esta operação move o bloco formado pelos primeiros genes de um genoma linear para qualquer outra posição do genoma, e baseado neste estudo propuseram um algoritmo com complexidade de tempo $O(n^2)$. Dias e Meidanis [21] relacionaram a estrutura do grafo-de-ciclos de Bafna e Pevzner com álgebra.

Hartman [16] observou que o problema de rearranjo para ordenar permutações circulares por transposições é equivalente ao de ordenar permutações lineares. Observou que todos os algoritmos para ordenar permutações lineares podem ser usados para ordenar as permutações circulares. Seu principal resultado foi um novo algoritmo com complexidade de tempo $O(n^{3/2}\sqrt{\log n})$, com fator de aproximação 1,5 e este é considerado mais fácil de compreender e implementar. Posteriormente, Elias e Hartman [9] propuseram um algoritmo com razão de aproximação 1,375 tendo complexidade de tempo $O(n^2)$, utilizando permutações simples.

1.3 Rearranjo de Genomas por transposições

Todo o estudo realizado em Rearranjo de Genomas visa resolver um quebra-cabeça combinatorial para encontrar uma sequência mínima de eventos de rearranjo (mutações), afetando porções grandes de genoma, que transformam um genoma no outro. Assim, os estudos de evolução baseados em rearranjos levam ao problema de distância de rearranjo, que de forma genérica, é encontrar a menor sequência de eventos de rearranjo necessários para transformar um genoma em outro.

Embora existam diversos tipos de eventos, trataremos apenas do evento de transposição que ocorre em um cromossomo: a transposição, move uma porção de uma região para outra dentro do cromossomo. Este evento de rearranjo leva a um problema combinatorial, sendo que sua complexidade ainda não é conhecida.

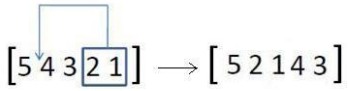

Figura 1.2: Uma transposição move um bloco de genes (representado por números) do cromossomo indicado pela marcação para o local indicado pela seta.

O formalismo clássico criado por Hannenhalli e Pevzner [4] modela um genoma enfatizando a visão do genoma como um conjunto de cromossomos e cada cromossomo como uma sequência de genes. Um gene é identificado por um inteiro. Os n genes encontrados em um genoma são representados pelo conjunto de inteiros $\{1,\ 2,\ ...\ n\}$. Para um cromossomo π, o gene encontrado na posição i é denotado por π_i.

Em problemas de rearranjo de genomas, trataremos genomas como sendo formados por um único cromossomo ou cromossomos encadeados, sem genes repetidos, sendo então representados por permutações. Uma permutação é qualquer ordenação dos elementos de um conjunto finito. Uma permutação dos elementos de um conjunto de tamanho n é chamada de permutação de comprimento n. O i-ésimo elemento de uma permutação π será representado por π_i. Permutações serão representadas por seus elementos entre colchetes e separados por um espaço em branco. Como exemplo podemos tomar π sendo uma permutação de comprimento n, logo $\pi = [\pi_1\ \pi_2\ \pi_3\ ...\ \pi_{n-1}\ \pi_n]$.

Uma permutação de n elementos é uma função bijetiva π tal que cada elemento π_i é um inteiro no intervalo $1 \leq \pi_i \leq n$, e todos os elementos são distintos: $\pi_i \neq \pi_j$ se $i \neq j$.

Duas permutações, em particular, são importantes no estudo de rearranjos, uma é a permutação identidade que leva todos os elementos neles próprios, sendo denotada por $\iota_{[n]} = [1\ 2 \cdots n]$. E a outra é a permutação reversa que é uma permutação que leva o elemento i no elemento $n - 1 + i$, sendo denotada por $\rho_{[n]} = [n\ n-1 \cdots 2\ 1]$.

Definimos também a inversa de uma permutação π, esta sendo denotada por π^{-1}, é uma permutação que mapeia cada elemento π_i de π em sua posição $\pi^{-1}(\pi) = i$, ou seja:

$$\pi^{-1} = [\pi_1^{-1}\ \pi_2^{-1}\ ...\ \pi_n^{-1}],$$

onde π_i^{-1} é a posição do elemento i em π. Exemplo: Seja $\pi = [3\ 5\ 2\ 1\ 4]$. A sua inversa π^{-1} deve ser tal que $\pi^{-1}(3) = 1, \pi^{-1}(5) = 2, \pi^{-1}(2) = 3, \pi^{-1}(1) = 4, \pi^{-1}(4) = 5$. Portanto, $\pi^{-1} = [4\ 3\ 1\ 5\ 2]$.

1.4 Transposições

A operação de transposição consiste em "cortar" certos blocos de genes do cromossomo e "colar" estes blocos de genes em um outro local dentro do mesmo cromossomo. Tal operação pode ser vista como a troca de posições entre dois blocos adjacentes. Matematicamente, definimos uma transposição na permutação π como sendo: Para toda permutação π, a transposição $t(i, j, k),\ 1 \leq i < j \leq n + 1,\ 1 \leq k \leq n + i,\ k \notin [i, j]$, aplicada em π, corta os elementos π_i a π_{j-1} e os cola entre π_{k-1} e π_k. Seja:

$$\pi = [\pi_1\ ...\ \pi_{i-1}\ \pi_i\ \pi_{i+1}\ ...\ \pi_{j-1}\ \pi_j\ \pi_{j+1}\ ...\ \pi_{k-1}\ \pi_k\ \pi_{k+1}\ ...\ \pi_n]$$

então:

$$\pi\ .\ t(i, j, k) = [\pi_1\ ...\ \pi_{i-1}\ \pi_j\ \pi_{j+1}\ ...\ \pi_{k-1}\ \pi_i\ \pi_{i+1}\ ...\ \pi_{j-1}\ \pi_k\ \pi_{k+1}\ ...\ \pi_n]$$

Exemplo: Dada a permutação [7 6 5 2 1 4 3], aplicando a transposição $t(1, 4, 6)$ obtemos [2 1 7 6 5 4 3].

A definição da operação de transposição é motivada pelo evento de mutação observado na comparação entre os genomas de organismos distintos [12]. Note que

desfazer uma transposição também é uma transposição. Por se tratar de uma reordenação, o resultado da aplicação de uma transposição $t(i, j, k)$ a uma permutação π será representado por $\pi \ . \ t(i, j, k)$.

1.4.1 Problema da ordenação por transposições

No problema da ordenação de genomas, queremos medir quão modificada uma permutação ficou em relação à outra devido aos eventos de rearranjos. Ordenar uma permutação por transposição é um problema em aberto, a complexidade do problema ainda não é conhecida.

Sendo π, σ duas permutações lineares de mesmo tamanho n, sempre será possivel transformar a primeira na segunda e vice-versa utilizando, no máximo, $n-1$ transposições, como descrito a seguir: para cada elemento $\sigma_x = [\sigma_1, \ ..., \sigma_n]$ que está na posição $\sigma^{-1}(\sigma_x) = x$ na permutação σ, de tal forma que a sua posição na permutação π seja diferente de sua posição em σ, ou seja, $\pi^{-1}(\sigma_x) \neq x$ é possivel, se $x < \pi^{-1}(\sigma_x)$ aplicar a transposição $t(x, \pi^{-1}(\sigma_x), \pi^{-1}(\sigma_x) + 1)$ a π, de forma que

$$\begin{aligned} \sigma &= [\ \sigma_1 \ ... \ \sigma_x \ ... \ ... \ ... \ \sigma_n \ ] \\ \pi &= [\ \pi_1 \ ... \ \pi_x \ ... \ \sigma_x \ ... \ \sigma_n \ ] \\ \pi \ . \ t(x, \pi^{-1}(\sigma_x), \pi^{-1}(\sigma_x) + 1) &= [\ \pi_1 \ ... \ \sigma_x \ \pi_x \ ... \ ... \ \pi_n \ ], \end{aligned}$$

ou se $x > \pi^{-1}(\sigma_x)$ aplicar a transposição $t(\pi^{-1}(\sigma_x), \pi^{-1}(\sigma_x) + 1, x + 1)$ a π, obtendo-se

$$\begin{aligned} \sigma &= [\ \sigma_1 \ ... \ ... \ ... \ \sigma_x \ ... \ \sigma_n \ ] \\ \pi &= [\ \pi_1 \ ... \ \sigma_x \ ... \ \pi_x \ ... \ \pi_n \ ] \\ \pi \ . \ t(\pi^{-1}(\sigma_x), \pi^{-1}(\sigma_x) + 1, x + 1) &= [\ \pi_1 \ ... \ ... \ \pi_x \ \sigma_x \ ... \ \pi_n], \end{aligned}$$

Em ambos os casos a aplicação da transposição a π resulta em uma permutação que tem o elemento σ_x na x-ésima posição, que é a sua posição em σ. Efetuando-se esta operação para cada elemento σ que está em posição diferente em π, teremos no pior caso $n-1$ transposições para tranformar σ em π. Esta sequência de transposições apresentada não é ótima, a menos de casos degenerados.

1.4.2 Distância de transposição

Dadas duas permutações π e σ, definimos $d_t(\pi, \sigma)$ como sendo a distância de transposição entre elas. Esta distância representa um número mínimo q de transposições, tal que $((\pi \ . \ t_1) \ . \ t_2) \ . \ ...t_{q-1}) \ . \ t_q) = \sigma$. É importante notar que essa distância será sempre finita se ambas as permutações forem de um mesmo conjunto de elementos. O diâmetro de transposição $D(n)$ é a maior distância de transposição possível entre duas permutações de comprimento n sobre um mesmo conjunto de elementos.

Como definimos anteriormente, se π e σ são permutações com o mesmo número de elementos, no pior caso com no máximo $n-1$ transposições é possivel encontrar uma sequência de transposições que transforma π em σ, logo a distância de transposição está limitada superiormente por este número $n-1$.

Exemplo. Seja $\pi = (3\ 2\ 1\ 6\ 5\ 4)$ e $\sigma = (1\ 2\ 3\ 4\ 5\ 6)$. Existe uma sequência de 3 transposições que transforma π em σ; Isto implica que $d_t(\pi, \sigma) \leq 3$, como vemos a seguir:

$$\begin{aligned}
\pi &= [3\ 2\ 1\ 6\ 5\ 4] \\
&\downarrow t(1,\ 3,\ 5) \\
&[1\ 6\ 3\ 2\ 5\ 4] \\
&\downarrow t(2,\ 4,\ 6) \\
&[1\ 2\ 5\ 6\ 3\ 4] \\
&\downarrow t(3,\ 5,\ 7) \\
\sigma &= [1\ 2\ 3\ 4\ 5\ 6]
\end{aligned}$$

Duas transposições são idênticas se não precisamos de nenhuma transposição para transformar uma na outra, logo temos que $\pi = \sigma$. Tendo que $t_1,\ t_2,\ ...,\ t_q$ é uma sequência mínima que transforma π em σ, temos por outro lado que o mesmo número de transposições utilizado para ordenar π em relação a σ pode ordenar σ em relação a π. Isto fica claro observando a sequência $t_q^{-1},\ t_{q-1}^{-1},\ ...,\ t_1^{-1}$, onde a inversa t_l^{-1} de uma transposição t_l é também uma transposição, transforma σ em π, e não

é possivel encontrar outra sequência com um menor número de transposições, pois se encontrarmos tal sequência reduzida poderiamos invertê-la e transformar π em σ com um menor número de transposições do que o possível, uma contradição.

A distância de transposição obedece a desigualdade triangular: Dadas as permutações π, σ, α com mesmo número de elementos, tomando como sequência mínima $t_1,\ ..., t_q$ para transformar π em σ e a outra sequência mínima $t'_1,\ ...,\ t'_s$ que transforma σ em α. Assim, teremos a sequência $t_1,\ ..., t_q,\ t'_1,\ ...,\ t'_s$, com $q + s$ transposições que transforma π em α, não sendo necessariamente o menor número de operações (transposições).

1.5 Objetivos

O objetivo desta dissertação é implementar o algoritmo de razão 1,5-aproximativo (Hartman e Shamir) para o problema da ordenação de permutações(genes) por transposições, utilizando uma estrutura de dados proposta por Feng e Zhu [11]. Iremos após implementar o algoritmo propor uma modificação na maneira de ordenar as permutações com o objetivo de conseguirmos realizar a ordenação de uma permutação com um menor número de transposições.

Esta dissertação contribui para o estudo do problema da distância de transposições cuja complexidade foi mostrada ser NP-completo por Bulteau, Fertin e Rusu [5].

Capítulo 2

Ordenação por transposição atráves de permutações simples

Neste capítulo apresentaremos o algoritmo 1,5-aproximativo proposto por Hartman e Shamir [17] para o problema de ordenação de uma permutação por transposições e uma estrutura de dados proposta por Sheng e Zhu [11] para otimizar o algoritmo. Inicialmente definimos alguns conceitos importantes para trabalhar com permutações (pontos de quebra e diagrama de realidade e desejo). Estas estruturas nos auxiliam no rearranjo de uma permutação, sendo indispensáveis para posteriormente entendermos os algoritmos de ordenação.

2.1 Pontos de quebra

Estudando rearranjo de genomas, sempre queremos comparar as estruturas genéticas e saber o quanto são diferentes. Uma medida para calcular a desordenação de uma em relação a outra seria o número de pontos de quebra. Dadas duas permutações π e σ, um par de elementos subsequentes em π e não em σ caracteriza um ponto de quebra de π em relação a σ. Esses elementos subsequentes em π e σ são ditos adjacentes. Para exemplificarmos pontos de quebra iremos primeiro definir permutação estendida. Dada $\pi = [\pi_1\ \pi_2\ ...\ \pi_n]$, para obtermos sua estendida basta acrescentarmos na permutação dois elementos fixos $\pi_0 = 0$ e $\pi_{n+1} = n + 1$, obtendo a estendida $\pi = [0\ \pi_1\ \pi_2\ ...\ \pi_n\ n + 1]$.

Exemplo: Sejam $\pi = [1\ 4\ 5\ 2\ 3\ 6\ 7]$ e a permutação identidade $\iota = [1\ 2\ 3\ 4\ 5\ 6\ 7]$.

Utilizaremos a permutação estendida de π para calcularmos os pontos de quebra de π em relação a ι,

$$[0\ \underline{1\ 4}\ \underline{5\ 2}\ \underline{3\ 6}\ 7\ 8]$$

$$[0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8]$$

Temos que os pares $\underline{1\ 4}$, $\underline{5\ 2}$ e $\underline{3\ 6}$ são pontos de quebra em relação a permutação identidade, definimos $b(\pi, \iota)$ como sendo o número de pontos de quebra de π em relação a ι, logo $b(\pi, \iota) = 3$.

Temos $n+1$ pares de elementos $0\pi\ \pi\pi_1\ \pi_1\pi_2\ ...\ \pi_{n-1}\pi_n\ \pi_n\pi_{n+1}$ então $0 \leq b(\pi, \iota) \leq n+1$. Um exemplo interessante é a permutação reserva, esta contém $n+1$ pontos de quebra, pois nenhum par é adjacente em relação a identidade: $b(\rho, \iota) = n+1$, o limite superior é atingido pela permutação reversa.

Teorema 2.1 (Bafna e Pevzner)**.** *Para todo par de permutações π e σ de n elementos temos*

$$d(\pi, \sigma) \geq b(\pi, \sigma)/3.$$

Demonstração. Sendo $\pi' = \pi\ .\ t(i, j, k)$, podemos ver que $b(\pi, \sigma)\ -\ b(\pi', \sigma) \leq 3$ mostrando que uma transposição troca um bloco da permutação π de lugar, isso ocorre cortando três posições adjacentes na permutação: $\pi_{i-1}\pi_i$, $\pi_{j-1}\pi_j$ e $\pi_{k-1}\pi_k$. Essa operação pode alinhar no máximo três elementos não adjacentes em relação a π. Abaixo mostramos uma permutação original π com os pontos de quebra destacados em relação a permutação π'.

$$\pi = [\pi_1\ ...\ \overbrace{\pi_{i-1}\ \pi_i}\ \pi_{i+1}\ ...\ \overbrace{\pi_{j-1}\ \pi_j}\ \pi_{j+1}\ ...\ \overbrace{\pi_{k-1}\ \pi_k}\ \pi_{k+1}\ ...\ \pi_n]$$

$$\downarrow t(i\ j\ k)$$

$$\pi.t(i, j, k) = [\pi_1\ ...\ \overbrace{\pi_{i-1}\ \pi_j}\ \pi_{j+1}\ ...\ \overbrace{\pi_{k-1}\ \pi_i}\ \pi_{i+1}\ ...\ \overbrace{\pi_{j-1}\ \pi_k}\ \pi_{k+1}\ ...\ \pi_n]$$

A cada transposição com o objetivo de ordenar a permutação π em relação a σ, podemos no máximo reduzir em três o número de pontos de quebra, logo obtemos a cota inferior $d(\pi, \sigma) \geq b(\pi, \sigma)/3$. □

2.2 Diagrama de Realidade e Desejo

Motivados pelo problema de encontrar a menor sequência de transposições que transforma uma permutação π qualquer de n elementos em uma outra permutação σ de n elementos, Bafna e Pevzner [2] desenvolveram uma estrutura chamada diagrama de ciclos. A estrutura também foi estudada por Meidanis, Walter e Dias [22] que a denominaram de diagrama de realidade e desejo. Esta estrutura é um grafo onde seus vértices obedecem uma certa ordem, este grafo se mostrou muito útil para o estudo do problema da distância de transposição. Nesta dissertação, sempre ordenaremos as permutações em relação a identidade.

Podemos definir o diagrama de realidade e desejo para a permutação π como sendo um grafo com representação $G(\pi)$, definido por um conjunto de vértices e um conjunto de arestas para simbolizar o estado da permutação. Cada elemento π_i, $1 \leq i \leq n$, dá origem a dois vértices $-\pi_i$ e $+\pi_i$, sendo que $\pi_0 = 0$ gera apenas o vértice $+0$ e $\pi_{n+1} = n + 1$ gera apenas o vértice $-(n + 1)$. Cada vértice possui duas arestas ligadas a ele, uma contínua simbolizando a aresta de realidade e outra pontilhada simbolizando a aresta de desejo. As arestas de realidade ligam os elementos adjacentes na permutação, por exemplo: $+\pi_i$ com $-\pi_{i+1}$, sendo $1 \leq i \leq n + 1$ e as arestas de desejo ligam os elementos $+i$ e $-(i + 1)$, com $0 \leq i \leq n$. Pela definição $G(\pi)$ é 2-regular, existe exatamente uma aresta de realidade e uma aresta de desejo incidentes no vértice. Assim podemos particionar o grafo de maneira única em ciclos alternantes e disjuntos em vértices entre si. A figura 2.1 mostra um exemplo do diagrama de realidade e desejo para a permutação $\pi = [3\ 2\ 1\ 6\ 5\ 4]$.

Figura 2.1: Diagrama de realidade e desejo para a permutação $\pi = [3\ 2\ 1\ 6\ 5\ 4]$

Ao percorrermos as arestas de realidade (contínuas) encontramos a configuração atual da permutação π e percorrendo as arestas de desejo (pontilhadas) encontramos a configuração desejada ι. Quando os extremos ligantes das arestas de realidade se

conectam aos mesmos vértices dos extremos ligantes das arestas de desejo, podemos deduzir que a permutação é a ι. Aplicando sucessivas transposições a π com intuito de transformar em ι, ocorrerá a quebra dos ciclos iniciais no diagrama de realidade e desejo $G(\pi)$ e a criação de $n+1$ ciclos unitários. Mostraremos a sequência de transposições necessária para ordenar a permutação π em relação a ι. A cada transposição aplicada a π, ocorrerá ou não variação no número de ciclos do grafo de realidade e desejo $G(\pi)$. Quando tivermos $n+1$ ciclos unitários, nossa permutação estará ordenada, por isso é muito importante para cada transposição aumentarmos o número de ciclos, assim iremos estar mais perto do desejo, que será sempre para *nós* a permutação identidade ι.

Na figura 2.2 conseguimos ordenar a permutação com três transposições. Começamos aplicando a π a transposição $t(1,3,5)$, quebrando três arestas de realidade do grafo $G(\pi)$ e criando novas três, gerando um novo grafo de realidade e desejo, agora com o número de ciclos aumentado de dois. Nosso objetivo é ter todos os ciclos de tamanho unitário. Aplicamos a seguir as transposições $t(2,4,6)$ e $t(3,5,7)$, com isso conseguimos com que nosso diagrama de realidade e desejo passe a ter todos seus ciclos unitários, finalizando a ordenação. Podemos concluir que $d(\pi,\iota) \leq 3$, neste exemplo, em todas transposições conseguimos um aumento no número de ciclos, veremos posteriormente que isto nem sempre é possível. Repare que na permutação ordenada todas as arestas de realidade e de desejo dos elementos coincidem.

Representamos o tamanho de cada ciclo em $G(\pi)$ pelo número de arestas de realidade do grafo, por este ser alternante entre arestas de realidade e desejo, também podemos medir o tamanho do ciclo pelo número de arestas de desejo. Um k-ciclo é definido como sendo um ciclo com k arestas de realidade, sendo denominado um k-ciclo longo se $k \geq 3$, e um k-ciclo curto, se $k \leq 2$.

Denotaremos por $c(\pi)$ o número de ciclos do grafo $G(\pi)$. Estudaremos agora a variação no número de ciclos quando ocorre uma transposição em π, isto gera um novo diagrama de realidade e desejo $G(\pi \,.\, t(i,j,k))$.

Teorema 2.2 (Bafna and Pevzner 1998)**.** *Seja π uma permutação e $t(i,j,k)$ uma transposição aplicada a π. Então:*

$$c(\pi \,.\, t(i,j,k)) - c(\pi) \in \{-2,0,2\}.$$

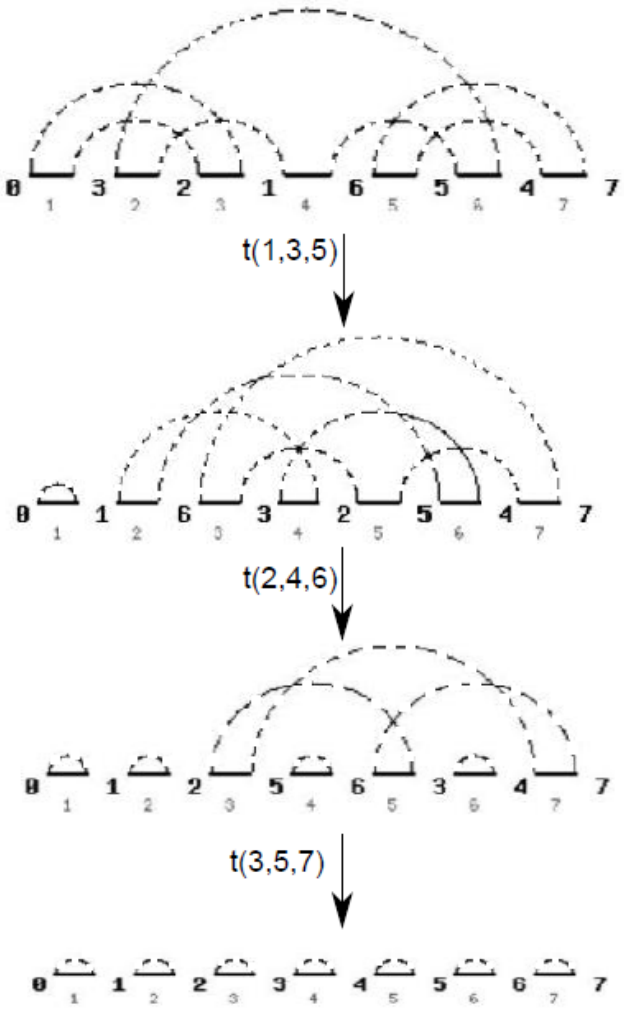

Figura 2.2: sequência de transposições para ordenar π em relação a ι.

Demonstração. Uma transposição aplicada a permutação π irá quebrar três arestas de realidade, (π_i, π_{i-1}), (π_j, π_{j-1}) e (π_k, π_{k-1}) e criar novas três arestas de realidade, (π_j, π_{i-1}),$(-\pi_i, -\pi_{k-1})$ e $(+\pi_k, -\pi_{j-1})$. Essas arestas podem pertencer a um único ciclo no diagrama de realidade e desejo $G(\pi)$, como também podem pertencer cada uma a um ciclo diferente, iremos agora estudar a disposição das arestas removidas na transposição:

Caso 1: As arestas que foram removidas na transposição pertencem a três ciclos diferentes em $G(\pi)$. Após ocorrer a transposição, o número de ciclos varia segundo a fórmula $c(\pi \ . \ t) = c(\pi) - 3 + 1$, como na figura 2.3.

Caso 2: As arestas removidas pertencem a dois ciclos diferentes, neste caso após ocorrer a transposição, o número de ciclos irá variar segundo a fórmula $c(\pi.t) = c(\pi) - 2 + 2$, os dois ciclos deram origem a outros dois ciclos, como na figura 2.4.

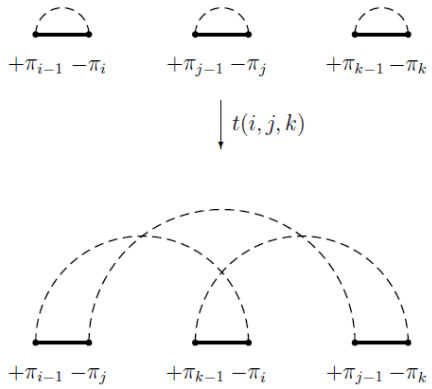

Figura 2.3: $G(\pi)$ contendo três ciclos e após a transposição $t(i,j,k)$, temos $G(\pi.t)$ com apenas um ciclo. Logo temos um -2-transposição.

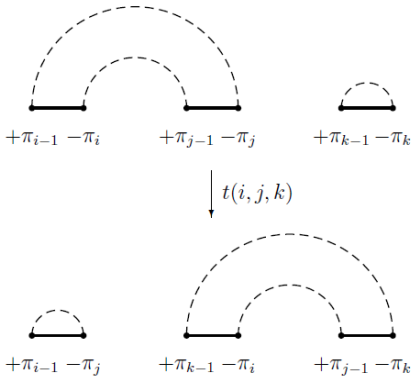

Figura 2.4: $G(\pi)$ contendo dois ciclos e após a transposição $t(i,j,k)$, o número de ciclos permanece constante. Logo temos um 0-transposição.

Caso 3: As arestas removidas pertencem a um único ciclo, iremos dividir este caso em dois subcasos. No primeiro subcaso o número de ciclos permanece constante, após a transposição t aplicada a $G(\pi)$, o número de ciclos varia segundo a formula $c(\pi)+1-1$, teremos um 0-transposição como na figura 2.5. No segundo subcaso, o número de ciclos varia segundo a fórmula $c(\pi.t) = c(\pi)-1+3$, o número de ciclos em $G(\pi.t)$ aumenta de duas unidades em relação a $G(\pi)$, teremos um $+2$-transposição como ilustrado na figura 2.6. □

Definimos Δc como sendo a variação no número de ciclos do diagrama de realidade e desejo de uma permutação causado por uma transposição. Da mesma forma definimos Δc_{imp} como sendo a variação no número de ciclos ímpares do diagrama,

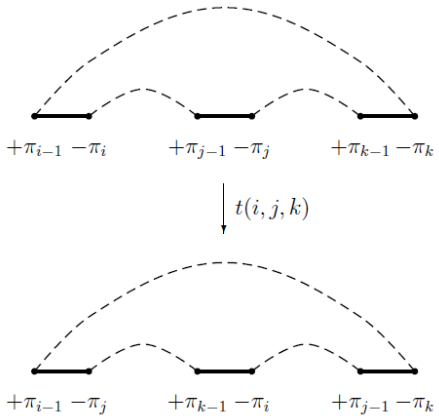

Figura 2.5: $G(\pi)$ contendo um único ciclo e após a transposição $t(i,j,k)$, o número de ciclos de $G(\pi.t)$ permanece constante. Logo temos um 0-transposição.

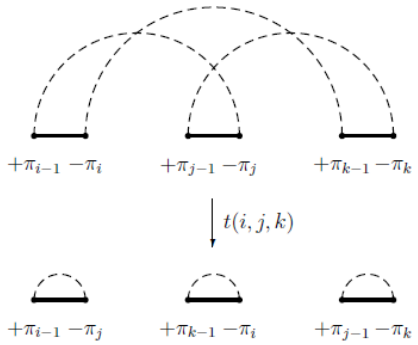

Figura 2.6: $G(\pi)$ contendo um único ciclo e após a transposição $t(i,j,k)$, o número de ciclos de $G(\pi.t)$ passa a ser três. Logo temos um +2-transposição.

após ocorrer uma transposição. Sendo π uma permutação e $t(i,j,k)$ uma transposição temos que: $\Delta c(\pi\ .\ t(i,j,k)) = c(\pi\ .\ t(i,j,k)) - c(\pi)$ e $\Delta c_{imp}(\pi\ .\ t(i,j,k)) = c_{imp}(\pi\ .\ t(i,j,k)) - c_{imp}(\pi)$.

Com base no diagrama de realidade e desejo, Bafna e Pevzner provaram os seguintes resultados:

Lema 2.3. *Dada uma permutação π e uma transposição $t(i,j,k)$, temos*

$$\Delta c_{imp}(\pi, t(i,j,k)) = \{-2, 0, +2\}.$$

Lema 2.4. *A distância de transposição entre duas permutações π e σ obedece ao seguinte limite inferior*

$$d(\pi) \geq \frac{n+1-c_{imp}(\pi)}{2}.$$

Lema 2.5. *Dada uma permutação π de comprimento n, que não seja a identidade, sempre é possível obter ou uma transposição $t(i,j,k)$ tal que $\Delta c(\pi, t(i,j,k)) = 2$ ou três transposições $t_1(i,j,k)$, $t_2(i',j',k')$ e $t_3(i'',j'',k'')$ tais que $\Delta c(\pi, t_1(i,j,k)) = 0$, $\Delta c(\pi \ . \ t_1(i,j,k), t_2(i',j',k')) = 2$ e $\Delta c(\pi \ . \ t_1(i,j,k) \ . \ t_2(i',j',k'), t_3(i'',j'',k'')) = 2$.*

Teorema 2.6. *Qualquer algoritmo que produza as transposições como indicadas pelo Lema 2.5 é um algoritmo de aproximação com fator 1,5 para o problema da distância de transposição.*

2.3 Permutação Simples

O grafo de realidade e desejo, bem como a permutação que o define, é chamado *simples* quando possui apenas ciclos curtos (tamanho máximo três). Hartman e Shamir [17], com a expectativa de simplificar a estrutura do grafo de realidade e desejo, isto é, as relações de entrelaçamento e cruzamento entre os ciclos e arestas, utilizaram uma transformação de permutações arbitrárias em permutações simples semelhante às transformações equivalentes utilizadas por Hannenhalli e Pevzner [15] no diagrama de realidade e desejo.

Dado o grafo $G(\pi)$, usaremos $b = (v_b, w_b)$ para denotar uma aresta de realidade e $g = (v_g, w_g)$ para denotar a de desejo, ambas pertencentes ao ciclo

$$C \ = \ (..., v_b, \ w_b, \ ..., \ v_g, \ wg, \ ...),$$

de $G(\pi)$. Um (g, b)-split de $G(\pi)$ consiste na seguinte série de operações sobre $G(\pi)$:

- Remoção das arestas b e g de $G(\pi)$.
- Adição de dois novos vértices v e w a $G(\pi)$.
- Adição de duas novas arestas de realidade (v_b, v) e (w, w_b).
- Adição de duas novas arestas de desejo (w_g, w) e (v, v_g).

O grafo resultante da aplicação de um (g, b)-split em $G(\pi)$ é denotado por $\hat{G}(\pi)$. Um (g, b)-split divide o ciclo C em dois novos ciclos em $\hat{G}(\pi)$, como na figura 2.7. Hannenhalli e Pevzer [15] demonstraram que existe uma permutação $\hat{\pi}$ de $n + 1$ elementos que define o grafo $\hat{G}(\pi)$ e portanto $\hat{G}(\pi) \ = \ G(\hat{\pi})$, nesse caso as permutações $\hat{\pi}$ e π são chamadas *equivalentes*. Seja $n(\pi)$ o número de arestas de realidade de π. Um (g, b)-split é chamado *seguro* quando essa operação preserva o parâmetro $c(\pi) \ - \ c_{imp}(\pi)$, ou em outros termos, temos $c(\pi) \ - \ c_{imp}(\pi) \ = \ c(\hat{\pi}) \ - \ c_{imp}(\hat{\pi})$.

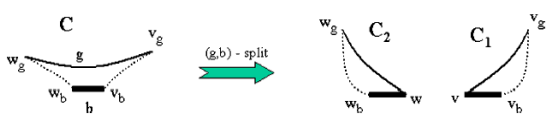

Figura 2.7: Os ciclos C_1 e C_2 são resultantes de um (g, b)-split aplicado ao ciclo C

Lema 2.7. *(Lin e Xue [20]) Qualquer permutação pode ser transformada em uma permutação simples por meio de splits seguros.*

De fato, os splits seguros preservam o limitante inferior para a distância de transposição, devido a garantia de que $c(\pi) - c_{imp}(\pi) = c(\hat{\pi}) - c_{imp}(\hat{\pi})$. Além disso, uma sequência de transposições que ordena π é relacionada a uma sequência de transposições que ordena $\hat{\pi}$ por meio do lema seguinte de Hannenhalli e Pevzner [15]:

Lema 2.8. *(Hannenhalli e Pevzner [15]) Dada uma permutação π, seja $\hat{\pi}$ uma permutação simples equivalente a π. Uma sequência de transposições que ordena $\hat{\pi}$ corresponde a uma sequência de transposições que ordena π com o mesmo número de operações.*

A importância deste lema se deve a possibilidade de transformar uma permutação cujo grafo de realidade e desejo é composto de ciclos longos e com estruturas complexas de entrelaçamento e cruzamento que exigiam a análise de diversos casos na teoria Bafna e Pevzner [2] em uma permutação simples envolvendo apenas ciclos curtos com um número reduzido de relacionamentos entre os ciclos.

2.4 Algoritmo 1,5-aproximativo de Hartman e Shamir

Apresentamos a definição de transposição segundo o trabalho de Hartman e Shamir [17]. Seja $\pi = [\pi_1 \ ... \ \pi_n]$ uma permutação de n elementos. Definimos como segmento A os elementos consecutivos $\pi_i, \ ... \ \pi_k$ $(k \geq i)$. Dois segmentos $A = \pi_i \ ... \ \pi_k$ e $B = \pi_j \ ... \ \pi_l$ são contíguos caso $j = k + 1$ ou $i = l + 1$. A transposição $t(i, j, k)$ aplicada a π troca dois blocos contíguos, veja a figura 2.8.

Em permutações circulares, pode-se definir uma transposição analogamente como a troca de dois segmentos contíguos. Note que aqui os índices são cíclicos, i.e. $\pi(\pi_i) = \pi_{i+1}$ para $1 \leq i \leq n-1$ e $\pi(\pi_n) = \pi_1$. A transposição quebra a permutação circular em três segmentos, ao contrário de no máximo quatro em uma permutação linear, veja a figura 2.8. Note que para permutações circulares, podemos assumir que todos os três segmentos são não vazios, pois caso contrário a permutação original e transformada seriam as mesmas. Existem apenas duas ordens cíclicas (sentido horário e anti-horário) e três segmentos, sabemos que cada dois dos três segmentos são contíguos, logo a transposição pode ser representada por qualquer troca de dois deles.

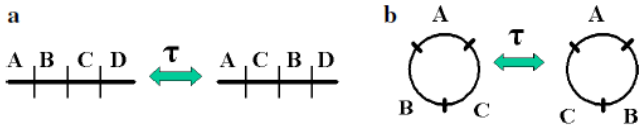

Figura 2.8: A transposição t aplicada a permutação linear π troca os segmentos B e C. A transposição t aplicada sobre uma permutação circular, pode ser vista como a troca de A e B, ou B e C, ou A e C.

Hartman e Shamir mostram que os problemas de ordenação de permutações circulares e lineares são equivalentes pois existe uma sequência de transposições que ordena uma permutação linear π e sua equivalente permutação circular denotada π_c [17].

Teorema 2.9 (Hartman e Shamir)**.** *O problema de ordenar permutações lineares por transposições é equivalente ao problema de ordenar permutações circulares por transposições.*

Demonstração. Dada uma permutação linear π de n elementos, para circularizar esta permutação basta adicionar o elemento $n+1$ sendo $\pi_{n+1} = n+1$, e fechar o ciclo. Iremos denotar a permutação circularizada por π_c. Qualquer transposição aplicada a π_c pode ser representada por dois segmentos que não incluem $n+1$. Assim, existe uma sequência ótima de transposições que ordena π_c, e nenhuma delas envolve $n+1$. A mesma sequência pode ser vista como uma sequência de transposições sobre a permutação linear π, ignorando $n+1$. Isto implica que $d(\pi) \leq d(\pi_c)$. Por outro lado, qualquer sequência de transposições sobre π também é uma sequência de transposições sobre π_c, logo $d(\pi_c) \leq d(\pi)$. Assim, $d(\pi_c) = d(\pi)$. Além disso, uma sequência ótima para π fornece uma sequência ótima para π_c.

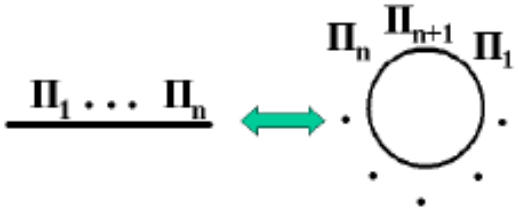

Figura 2.9: Na permutação circular, um novo elemento, $n+1$, é introduzido.

Seguindo a outra direção, começamos com uma permutação circular π_c. Podemos linearizá-la removendo um elemento arbitrário, que desempenha o papel de $n+1$, veja a figura 2.9. Os mesmos argumentos implicam que uma solução ótima para permutação linear é uma solução ótima para a circular. □

Hartman e Shamir adotaram a representação de genomas por meio de permutações circulares. Dada a permutação π sobre $[n]$ defina $f(\pi)$ como a permutação sobre $[2n]$ tal que cada elemento i de π é substituído pelos elementos $2i-1$ e $2i$, nessa ordem em $f(\pi)$. Dadas a permutação circular $\pi = (\pi \ ... \ \pi_n)$ e $f(\pi) = \pi' = (\pi'_1 \ ... \ \pi'_{2n})$, o grafo de realidade e desejo $G(\pi)$ é um diagrama que possui como conjunto de vértices os elementos $1, 2, \ ... \ 2n$ e π'_{2i} é ligado a π_{2i+1} por uma aresta de realidade, chamada b_i e o elemento $2i$ é ligado a $2i+1$ por meio de uma aresta de desejo, para $1 \leq i \leq n$. Uma vez que cada vértice possui duas arestas incidentes, o diagrama possui uma única decomposição em ciclos. Todos os resultados provados utilizando o diagrama de realidade e desejo para permutações lineares, são válidos para permutações circulares. As definições anteriores sobre número de

ciclos, variação de número de ciclos e o limite inferior mostrado por Bafna e Pevzner [1] também são válidas para as circulares. Bafna e Pevzner trabalham com permutações simples, logo o diagrama de realidade e desejo $G(\pi)$ passa por um pré-processamento e em sua nova configuração $G(\pi')$ haverá apenas ciclos de tamanho máximo três. Há apenas dois tipos possíveis de configurações de 3-ciclos. A Figura abaixo mostra os dois tipos de configurações de um 3-ciclo. Uma transposição

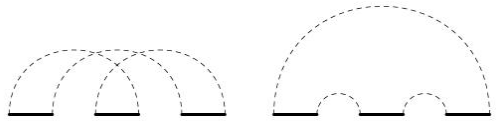

Figura 2.10: 3-ciclo orientado e 3-ciclo desorientado.

$t(i,j,k)$ atuando em π quebrará três arestas de realidade do diagrama de realidade e desejo $(r(\pi_{i-1}), l(\pi_i))$,$(r(\pi_{j-1}), l(\pi_j)))$ e $(r(\pi_{k-1}), l(\pi_k))$, e criará novas três arestas $(r(\pi_{i-1}), l(\pi_j)))$,$(r(\pi_{k-1}), l(\pi_i)))$ e $(r(\pi_{j-1}), l(\pi_k))$ não alterando as arestas de desejo. Usaremos b_i para denotar a aresta $(r(\pi_i), l(\pi_{i+1}))$, $0 \leq i \leq n$. Nós dizemos que um par de arestas $< b_i, b_j >$ **intercepta** outro par de arestas $< b_k, b_l >$ se $i < k < j < l$ ou $k < i < l < j$. Seja C um k-ciclo. Sejam as arestas de C listadas $b_{i_1}, b_{i_2}, ..., b_{i_k}$, tal que b_{i_1} a aresta mais a esquerda de C na permutação π, e o vértice direito de b_{i_j} está conectado ao vértice esquerdo de $b_{i_{j+1}}$ e a aresta de desejo $(r(\pi_{i_{j+1}}), l(\pi_{i_{j+1}}))$, $1 \leq j \leq k$, $b_{i_{k+1}} = b_{i_1}$. Se $i_1, i_2, ..., i_k$, temos que C é um ciclo **desorientado**, senão C é um ciclo **orientado**.

Cada transposição realizada irá modificar nosso diagrama de realidade e desejo $G(\pi)$. Uma transposição sobre π pode variar nosso número de ciclos ímpares em +2, 0 ou -2 [Bafna e Pevzner 1998]. Uma transposição é chamada k-transposicão se esta incrementa de k o número de ciclos ímpares no diagrama de realidade e desejo $G(\pi)$. Descrevemos $(0,2,2)$-sequência como sendo uma sequência de 3 transposições em π, sendo que a primeira transposição é uma 0-transposição, não variando o número de ciclos ímpares e as demais são 2-transposições aumentando o número de ciclos ímpares em duas unidades cada uma.

Sejam $C = b_i, b_j, b_k$ e $D = b_{i'}, b_{j'}, b_{k'}$ dois 3-ciclos desorientados, onde b_i, b_j, b_k são arestas de realidade do ciclo C e $b_{i'}, b_{j'}, b_{k'}$ são arestas de realidade do ciclo D, se $i < i' < j < j' < k < k'$ ou $i' < i < j' < j < k' < k$, dizemos que C está

entrelaçado com D. Sejam C, D e E 3-ciclos desorientados. Se todo par de arestas de realidade em E intercepta um par de arestas de realidade em C ou intercepta um par de arestas de realidade em D, dizemos que o ciclo E está **saturado** por C e D.

Teorema 2.10 (Christie 1999)**.** *Se π é uma permutação que contém 2-ciclos, então existe uma 2-transposição para π.*

Teorema 2.11 (Bafna and Pevzner 1998)**.** *Se $G(\pi)$ contém 3-ciclos orientados, então existe uma 2-transposição para π.*

Teorema 2.12 (Hartman and Shamir 2006)**.** *Se $G(\pi)$ contém dois 3-ciclos desorientados e entrelaçados, então existe uma $(0, 2, 2)$-sequência de transposições para π.*

Teorema 2.13 (Hartman and Shamir 2006)**.** *Se $G(\pi)$ contém três 3-ciclos desorientados C, D e E sendo que E é saturado por C e D, então existe uma $(0, 2, 2)$-sequência para π.*

Os teoremas 2.10, 2.11, 2.12 e 2.13 garantem que para qualquer permutação simples π sempre irá existir uma 2-transposição ou $(0, 2, 2)$-sequência de transposições para π. Hartman and Shamir [2006] propuseram o algoritmo 1 para ordenar qualquer permutação.

Algoritmo 1: Sort

Entrada: π

Saída: π ordenado

1 **início**

2 **1.** Transformar a permutação π em uma permutação simples $\hat{\pi}$

3 **2.** Chamar o algoritmo SimpleSort($\hat{\pi}$) para computar a sequência de transposições $t_1,\ t_2,\ ...,\ t_m$ para ordenar $\hat{\pi}$

4 **3.** As transposições encontradas $t_1,\ t_2,\ ...,\ t_m$ ordenam a permutação simples $\hat{\pi}$. Como π e $\hat{\pi}$ são equivalentes, ordenamos π com base no lema 2.8;

5 **fim**

Algoritmo 2: SimpleSort

Entrada: π

Saída: π ordenado

```
1  início
2      enquanto G = G(π̂) possuir 2-ciclos faça
3          Aplicar 2-transposição.
4      enquanto G possuir 3-ciclos faça
5          Tomar um 3-ciclo C = b_{i1}, b_{i2}, b_{i3}
6          se C for orientado então
7              Aplicar 2-transposição
8          senão
9              Tomar outro ciclo D com um par arestas de realidade que
               intersecta < b_{i1}, b_{i2} >
10             se C entrelaçados com D então
11                 Aplicar (0, 2, 2)-sequência
12             senão
13                 Tomar o ciclo E o qual tem um par de arestas de realidade que
                   intersecta com < b_{i2}, b_{i3} >
14                 se E entrelaçado com C ou D então
15                     Aplicar (0, 2, 2)-sequência
16                 senão
17                     Ciclo C é saturados pelos ciclos D e E, aplicar
                       (0, 2, 2)-sequência para C, D e E.
18 fim
```

O algoritmo 1 para o problema de ordenação de permutações por transposições possui complexidade de tempo $O(n^{3/2}\sqrt{n \log n})$. Iremos a seguir estudar uma estrutura de dados que melhora a complexidade do algoritmo 1.

2.5 Estrutura de Dados Proposta por Feng e Zhu

A estrutura de dados proposta por Feng e Zhu [11] para a implementação do algoritmo com fator de aproximação 1,5 proposto por Hartman e Shamir [17] é uma árvore binária. A estrutura de dados melhora o tempo de execução para ordenação da permutação por transposição. O algoritmo aproximativo para ordenar permutações com transposições tem complexidade de tempo $O(n^{3/2}\sqrt{\log n})$. Utilizando a estrutura árvore de permutação, podemos melhorar esse algoritmo, realizando a ordenação com complexidade de tempo $O(n \log n)$. Os elementos da permutação serão os *nós* da árvore, trabalharemos com árvores binárias balanceadas, cada *nó* folha representará um elemento da permutação, o conjunto das folhas da árvore representa a permutação. Essa estrutura é chamada árvore de permutação. Usando a estrutura árvore de permutação, podemos fazer o rearranjo de uma permutação com a complexidade de tempo $O(n \log n)$, sendo n o número de elementos da permutação.

2.5.1 Árvore de Permutação

A árvore de permutação é uma árvore binária balanceada T com o *nó* raiz r, onde cada *nó* interno tem dois filhos. Sendo s um *nó* da árvore T, o *nó* filho a esquerda de s seria representado por $L(s)$ e o *nó* filho a direita de s seria representado por $R(s)$. A altura da folha é zero e dos demais *nós* serão $H(s) = max\{H(L(s)), H(R(s))\}+1$. Nossa árvore por ser balanceada, para cada *nó* s teremos, $H(L(s)) - H(R(s)) \leq 1$. O tamanho da árvore é definido pela altura do *nó* raiz, $H(T) = H(r)$.

Sendo uma permutação π de comprimento n, $\pi = [\pi_1\ \pi_2\ \pi_3\ ...\ \pi_{n-1}\ \pi_n]$, a árvore de permutacao será composta por um número n de *nós* folha, e o valor de cada folha será $\pi_1, \pi_2, ..., \pi_n$ respectivamente. A árvore é construída de maneira bottom-up, ou seja, esta é construída a partir das folhas em direção a raiz. Sendo s um *nó* interno da árvore T, seu valor é calculado da seguinte forma: compara-se o valor de $L(s)$ e $R(s)$, pega-se o maior valor e atribui a s. Construindo a árvore desta maneira garantimos o valor da raiz de T como sendo o maior valor da permutação. Como a árvore é balanceada, temos um limite superior $O(\log n)$ para a altura da estrutura.

Teorema 2.14. *A altura da árvore de permutação correspondente a* $\pi = [\pi_1\ \pi_2\ ...\ \pi_n]$ *é limitada superiormente por* $O(\log n)$.

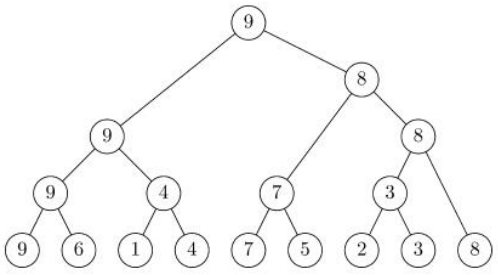

Figura 2.11: A árvore de permutação para $\pi = 9, 6, 1, 4, 7, 5, 2, 3, 8$.

Para realizarmos as transposições na árvore de permutação não é necessário inserirmos ou removermos elementos da estrutura. Feng e Zhu [11], utilizam três operações bem definidas: A primeira é chamada de *Build*, esta operação constroi a árvore de permutação. As outras duas operações são *Join* e *Split* que fazem respectivamente a união de duas árvores disjuntas e quebra de uma árvore em duas correspondentes.

A operação de *Build* da árvore de permutação é feita camada por camada, começando das folhas e subindo em direção a raiz, sendo um caminho bottom-up. A primeira camada é formada pelas folhas, cada *nó* folha recebe um elemento da permutação, sendo que a adjacência de elementos na permutação implica que os elementos sejam adjacentes na estrutura. Para construirmos as camadas superiores, percorremos da esquerda para a direita pares de *nós* adjacentes, como a seguir: Dado um par de *nós*, sabendo que cada *nó* deve ser utilizado uma única vez no processo, selecionamos deste par o *nó* com maior valor e este é replicado na camada superior como pai do par de *nós* adjacêntes. O algoritmo 3 formaliza a construção.

Algoritmo 3: Build

Entrada: π

Saída: U

início
  Criar os *nós* folha u_i para π_i, $1 \leq i \leq n$
  $U \leftarrow [u_i | 1 \leq i \leq n]$
  $k \leftarrow n$
  para $k > 1$ **faça**
    Criar v_i, setar $L(v_i) = u_{2i-1}$, $R(v_i) = u_{2i}$ para $1 \leq i \leq \lfloor k/2 \rfloor$
    se k *for par* **então**
      $U \leftarrow [v_i | 1 \leq i \leq k/2]$
    se k *for ímpar* **então**
      Criar v, $L(v) = v_{\lfloor k/2 \rfloor}$, $R(v) = u_k$
      $U \leftarrow [v_i | 1 \leq i < \lfloor k/2 \rfloor] \cup [v]$
    $k \leftarrow \lfloor k/2 \rfloor$
fim

Teorema 2.15. *O algoritmo Build sempre irá criar uma árvore correspondente a permutação dada. A complexidade de tempo é* $O(n)$.

Em cada rodada, as alturas dos *nós* em U são idênticas, exceto para a última posição, onde pode ocorrer da altura ser uma unidade maior. Isso garante que durante a construção a árvore se mantém balanceada. A complexidade de tempo é limitada pelo número de *nós* criados, que é $O(n)$. Seja T_1 árvore de permutação correspondente a $[\sigma_1 = \pi_1 \ \cdots \pi_m]$ e T_2 outra árvore correspondente a $\pi_{m+1}...\pi_n$, então o $Join(T_1, T_2)$ retorna uma árvore de permutação correspondente a permutação π_1 ... π_m π_{m+1} ... π_n. O algoritmo 4 formaliza a operação.

Algoritmo 4: Join

Entrada: T_1, T_2

Saída: T

```
1  início
2      se H(T1) − H(T2) ≤ 1 então
3          Criar um novo nó s, sendo que L(s) = T1 e R(s) = T2
4      se H(T1) − H(T2) = 2 então
5          se H(L(T1)) ≤ H(R(T1)) então
6              Criar 2 novos nó s' e s
7              Atribuir L(s') = R(T1), R(s') = T2, L(s) = L(T1) e R(s) = s'
8          se H(L(T1)) < H(R(T1)) então
9              Criar 3 novos nó s'', s' e s
10             Atribuir L(s') = L(T1), R(s') = L(R(T1)), L(s'') = R(R(T1)),
               R(s'') = T2,L(s) = s', R(s) = s''
11     se H(T1) − H(T2) > 2 então
12         s = Join(L(T1), Join(R(T1), T2))
13 fim
```

Teorema 2.16. *Sejam T_1 e T_2 as árvores correspondentes às sequências disjuntas $\pi_1, ..., \pi_m$ e $\pi_{m+1}, ..., \pi_n$, respectivamente. O método Join retorna a árvore de permutação correspondente à sequência π_1, ..., π_m, π_{m+1}, ..., π_n. A complexidade de tempo de $Join(T_1, T_2)$ é $O(H(T_1) - H(T_2))$.*

O algoritmo apresentado recebe como parâmetro duas árvores binárias balanceadas T_1, T_2 e retorna uma árvore T também binária e balanceada, o algoritmo é subdividido de acordo com a diferença de altura entre T_1 e T_2:

Caso $H(T_1) - H(T_2) \leq 1$, temos $L(T) = T_1$, $R(T) = T_2$, assegurando $|H(L(T)) - H(R(T))| \leq 1$. A figura 2.12 representa a situação. Esta etapa leva tempo $O(1)$.

Caso $H(T_1) - H(T_2) = 2$, Se $H(L(T_1)) \geq H(R(T_1)$ então de acordo com a linha 4 do algoritmo 4, $|H(L(s')) - H(R(s'))| = |H(R(T_1)) - H(R(T_2))| \leq 1$, e $|H(L(s)) - H(R(s))| = |H(L(T_1)) - H(s')| = |H(L(T_1)) - H(R(T_1)) - 1| \leq 1$. As figuras 2.13 e 2.14 representam a situação.

Figura 2.12: A operação de $Join(T_1, T_2)$ para o caso $|H(L(T_1)) - H(R(T_2))| \leq 1$, $H(T_1) = k$ e $H(T_2) = k$ ou $k - 1$.

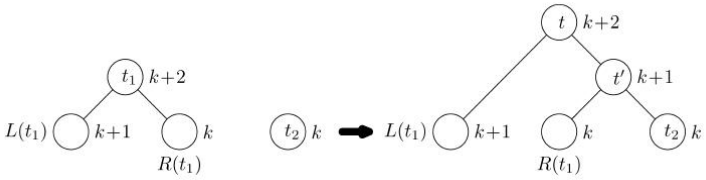

Figura 2.13: A operação de $Join(T_1, T_2)$, sendo $H(T_1) = k + 2$, $H(T_2) = k$ e $H(L(T_1)) > H(R(T_1))$.

Se $H(L(t_1) < H(R(t_1)))$, de acordo com a linha 8 do algoritmo 4, $|H(L(s')) - H(R(s'))| = |H(L(T_1)) - H(L(R(T_1)))| \leq 1$, $H(L(T_1)) \geq H(L(R(T_1)))$, $|H(L(s")) - H(R(s"))| = |H(R(R(T_1))) - H(T_2)| \leq 1, H(R(R(T_1))) \leq H(T_2)$. Assim $|H(L(s)) - H(R(s))| = H(L(T_1)) - H(T_2) \leq 1$. A figura 2.15 representa a situação. Este passo leva tempo $O(1)$.

Caso $H(T_1) - H(T_2) > 2$. A chamada recursiva para os casos anteriores simplifica o problema, terminando quando encontrarmos um *nó* $u = R(v)$ que satisfaz $1 \leq H(u) - H(T_2) \leq 2$ pela primeira vez. De acordo com as etapas passadas, juntando-se u e T_2 levamos tempo constante para produzir uma árvore W tal que $H(W) - H(u) \leq 1$. Assim $|H(W) - H(L(v))| \leq 2$. Isso implica que juntando $L(v)$ e W teremos também custos de tempo constante. Pelo mesmo argumento, podemos mostrar que, para cada momento de calcular $Join(L(T_1), Join(R(T_1), T_2))$, a diferença de altura entre $L(T_1)$ e associação $(R(T_1), T_2)$ não é mais do que dois. Assim a operação sempre produz em tempo constante uma árvore de permutação correspondente a um intervalo da permutação.

Seja a permutação $\pi = [\pi_1, \ ..., \ \pi_{m-1}, \pi_m, \ ..., \ \pi_n]$ o método $Split(T, m)$ desmembra a permutação π nas permutações $\pi_l = [\pi_1, ..., \pi_{m-1}]$ e $\pi_r = [\pi_m, ..., \pi_n]$. Sendo T a árvore de permutação que corresponde a $\pi = [\pi_1, \ ... \ \pi_{m-1}, \ \pi_m, \ ..., \ \pi_n]$, representamos o caminho da folha π_m até a raiz da árvore T por $P =< v_0, v_1, ..., v_k >$, tal que

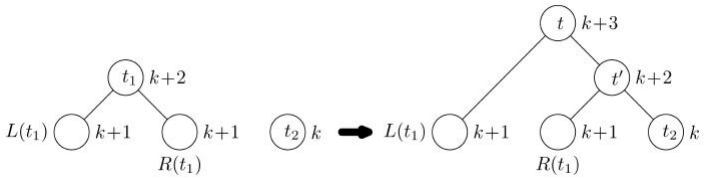

Figura 2.14: A operação de $Join(T_1, T_2)$, sendo $H(T_1) = k + 2$, $H(T_2) = k$ e $H(L(T_1)) = H(R(T_1))$.

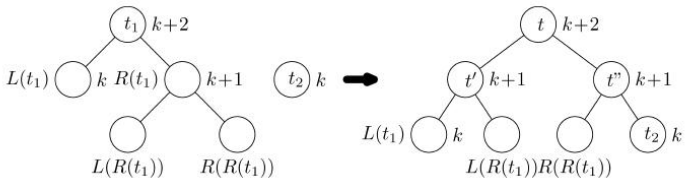

Figura 2.15: A operação de $Join(T_1, T_2)$, sendo $H(T_1) = k + 2$, $H(T_2) = k$ e $H(L(T_1)) < H(R(T_1))$.

$v_0 = \pi_m$ e $v_k = r$, sendo r a raiz da árvore T. Se $U_l = \{L(v_i)|v_{i-1} = R(v_i), 0 < i \leq k\}$ e $U_r = \{v_0\} \cup \{R(v_i)|v_{i-1} = L(v_i), 0 < i \leq k\}$, então concatenando o intervalo correspondente aos *nós* em U_l da esquerda para direita obtemos π_l e concatenando o interválo U_r também na ordem da esquerda para a direita, obtemos π_r. Portanto, temos o algoritmo 5 para *Split*.

Algoritmo 5: Split

Entrada: T, m

Saída: T_r, T_l

```
1 início
2     Encontrar o caminho de π_m até a raiz, sendo r a raiz de T. Assumir que o
      caminho é v_0, v_1, ..., v_k, onde v_0 = π_m e v_k = r. t_r ← v_0, t_l ← null
3     para i = 1 até i = k faça
4         se L(v_i) = v_{i-1} então
5             t_r ← Join(T_r, R(v_i))
6         se R(v_i) = v_{i-1} então
7             t_l ← Join(L(v_i), T_l)
8 fim
```

Teorema 2.17. *Supondo T sendo nossa árvore de permutação correspondente a $\pi = [\pi_1,\ ...,\ \pi_{m-1},\ \pi_m,\ ...,\ \pi_n]$. O algoritmo 5 sempre retorna T_l correspondente a $\pi_l = [\pi_1, ..., \pi_{m-1}]$, e T_r correspondente a $\pi_r = [\pi_m, ..., \pi_n]$. A complexidade de tempo do algoritmo 5 é $O(\log n)$.*

Para utilizarmos nossa estrutura Árvore de Permutação, que representa e manipula uma permutação, temos que guardar alguns atributos importantes nos *nós* da árvore. Por exemplo, caso se precise calcular a posição de um elemento na permutação. Para cada *nó* s, definimos como atributo um número inteiro x para guardar a posição de s na permutação. A posição de π_i pode ser calculada percorrendo o caminho de π_i (folha) até a raiz da árvore e contando o número de elementos a esquerda de π_i que existe na permutação. Este cálculo pode ser feito em $O(\log n)$ para cada elemento.

Capítulo 3

Implementação e Otimização

Neste capítulo, mostraremos nossa implementação do algoritmo 1,5-aproximativo para ordenação de permutações por transposições proposto por Hartman e Shamir [17] utilizando na implementação a estrutura de dados árvore de permutação, proposta por Feng e Zhu [11]. Nossa implementação foi realizada utilizando a linguagem de programação Java. Java é uma linguagem Orientada a Objetos, logo criamos objetos, classes Java, para que esses realizem a ordenação de uma permutação por transposições. A seguir iremos descrever todas as classes Java criadas e a finalidade de sua implementação.

O estudo que fizemos do algoritmo de Hartman e Shamir e da estrutura de dados de Feng e Zhu no capítulo 2 nos fornece apenas um arcabouço para a implementação, ficando o desafio de construirmos algoritmos eficientes para resolver as ideias propostas. Podemos citar como exemplo, no artigo de Hartman e Shamir [17] o procedimento para transformar uma permutação em simples. Este não especifica detalhes da implementação, também não formaliza a resolução do problema com um algoritmo, fornecendo apenas uma breve descrição de como resolver o problema. O artigo de Feng e Zhu [11] fornece algoritmos para trabalharmos com árvores binárias, deixando a nosso cargo implementá-los de maneira inteligente, citando poucos atributos para os *nós* das árvores, omitindo outros importantes para implementar a transposição. Neste trabalho implementamos funcionalidades para calcularmos a distância de transposição entre permutações. Nossa implementação terá ordem de complexidade $O(n \log n)$, partindo de uma permutação qualquer e chegando na identidade.

Dividimos este capítulo nas seguintes partes: Primeiro, descrevemos como implementamos a transformação de uma permutação em simples. Antes de montar nossa árvore de permutação, teremos que nos certificar de que a permutação é simples. Caso contrário fazemos um pré-processamento, pois a estrutura de dados árvore de permutações trabalha com permutações simples. O pré-processamento de uma permutação é tratado na classe *Permutation*. Em seguida descrevemos a classe *Node*, que representa os *nós* de uma árvore binária. Instâncias do objeto *Node* constrói a árvore de permutação. Descrevemos também neste capítulo as classes *PermutationTree* e *Transposition* implementadas. Estas otimizam a estrutura de dados utilizada para realizar as transposições. A classe *PermutationTree* é constituída pelos métodos de criação e manipulação de uma árvore de permutação, já a classe *Transposition* tem como principal função realizar as transposições. Por fim, descrevemos a classe *SortingByTransposition* e a nossa otimização proposta. A classe *SortingByTransposition* é a classe principal do sistema. Esta implementa o algoritmo 1,5-aproximativo proposto por Hartman e Shamir [17] para ordenar uma permutação utilizando a estrutura de Feng e Zhu [11]. Nossa otimização sobre o algoritmo de Hartman e Shamir finaliza o capítulo.

3.1 A classe Permutation

Esta classe realiza a etapa de pré-processamento da permutação. O objetivo do pré-processamento de uma permutação é transformá-la em uma permutação simples (ciclos no diagrama de Realidade e Desejo de tamanho máximo três). A classe recebe como parâmetro uma permutação $\pi = [\pi_1\ \pi_2\ ...\ \pi_n]$ qualquer de n inteiros. Se π é uma permutação que possui ciclos com tamanho maior que três, então é realizado o seu pré-processamento, explicado no decorrer do capítulo.

Utilizaremos a permutação reversa $\rho = [7\ 6\ 5\ 4\ 3\ 2\ 1]$ para exemplificar o pré-processamento. Tomando ρ e obtendo seu grafo de realidade e desejo $G(\rho)$ representado na figura 3.2, temos a configuração dos ciclos de ρ. Cada elemento da permutação esta associado a um ciclo e a quatro arestas ligantes, sendo duas arestas de realidade e outras duas de desejo, exceto o primeiro e último elemento que foram acrescidos na permutação, possuindo apenas duas arestas ligantes cada, sendo uma de realidade e outra de desejo. As arestas de realidade no grafo representam a adjacência dos elementos em ρ, enquanto arestas de desejo representam os elementos ordenados. Percorrendo somente as arestas de desejo do diagrama de realidade e desejo, obtemos os elementos ordenados, $[1\ 2\ 3\ ...\ n]$.

Para obtermos os ciclos de uma permutação, devemos analisar o diagrama de realidade e desejo desta. Cada elemento da permutação pertence a um dos ciclos do diagrama, caso o ciclo não seje unitário, este possui sucessor no ciclo, sendo este sucessor alcançado após percorrermos um par de arestas (sendo uma de realidade e outra de desejo), figura 3.1. Quando o sucessor no ciclo coincidir com o primeiro elemento do ciclo (elemento visitado no ciclo), significa que fechamos o ciclo, bastando contar o número de arestas de realidade para sabermos seu tamanho. Por exemplo, tomando o elemento 0 na permutação da figura 3.2, temos que seu sucessor no ciclo é o elemento 6, que tem como sucessor o elemento 4, que tem como sucessor o elemento 2. O sucessor do elemento 2 é o elemento 0, de onde partimos, logo não o incluímos novamente e fechamos o ciclo em (0 6 4 2). Para obter os outros ciclos em $G(\rho)$, obtemos um elemento ainda não visitado e percorremos todo seu ciclo, o procedimento continua até passarmos por todos elementos da permutação. Para nosso exemplo obteremos os seguintes ciclos, (7 5 3 1) e (0 6 4 2). Como estes ciclos possuem mais de três elementos teremos que realizar o pré-processamento em

ρ, realizamos este inserindo elementos na permutação.

Figura 3.1: Partindo do elemento 0 e percorrendo respectivamente uma aresta de realidade e outra de desejo, chegaremos ao proximo elemento daquele ciclo, no exemplo da figura é o elemento 6. Retiramos as outras arestas do diagrama de realidade e desejo para melhor entendimento.

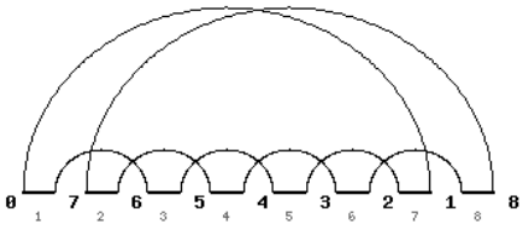

Figura 3.2: Diagrama de realidade e desejo da permutação $\rho = [7\ 6\ 5\ 4\ 3\ 2\ 1]$.

Inserindo elementos na permutação ρ, figura 3.2, quebramos algumas das arestas em $G(\rho)$, modificando o grafo. O objetivo deste procedimento é criar ciclos menores, até obtermos uma permutação simples equivalente.

Para transformarmos uma permutação em simples, nos criamos na classe *Permutation* arrays para guardarem informações sobre os ciclos da permutação. O array F_1, por exemplo, guarda na sua primeira posição o número de ciclos da permutação, as demais posições do array guardam o tamanho de cada ciclo. Em $G(\rho)$ temos, $F_1 = \{2,\ 4,\ 4\}$, sendo dois ciclos de quatro elementos cada. Outro array, F_2, especifica a qual ciclo pertence os elementos de F_1. Definimos F_2 como um array que guarda os ciclos concatenados. Por exemplo, sendo a e b ciclos de uma permutação qualquer, representados por $a = (a_1\ a_2\ ...\ a_n)$ e $b = (b_1\ b_2\ ...,\ b_m)$, a configuração de F_2 será $F_2 = \{a_1\ a_2\ ...\ a_n\ b_1\ b_2\ ...\ b_m\}$. Sabendo onde termina um ciclo e começa outro visualizamos F_1. Em $G(\rho)$ temos $F_2 = \{0,\ 6,\ 4,\ 2,\ 1,\ 7,\ 5,\ 3\}$. Os quatro

primeiros elementos pertencem ao primeiro ciclo e os quatro últimos ao segundo ciclo.

Precisamos calcular a quantidade de elementos a inserir em cada ciclo. Ciclos podem ter tamanho máximo três, logo:

$$ne = \frac{\lfloor tc - 3 \rfloor}{2}$$

onde ne corresponde ao número de elementos a ser inserido no ciclo e tc corresponde ao tamanho do ciclo. Cada elemento inserido deve quebrar o ciclo em duas partes. O processo se repete até que todos ciclos da permutação tenham tamanho menor que três.

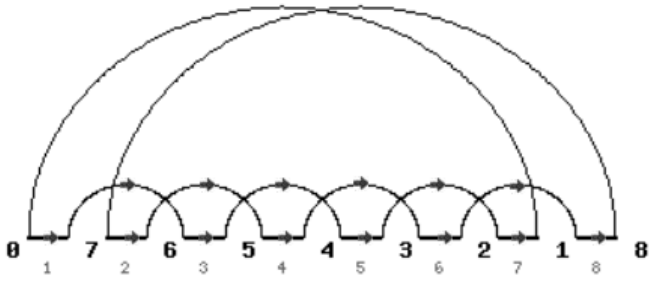

Figura 3.3: Diagrama de realidade e desejo $G(\rho)$ orientado.

A permutação, $\rho = [7\ 6\ 5\ 4\ 3\ 2\ 1]$ possui $c(\rho) = 2$, ambos de tamanho 4. Percorrendo os ciclos em $G(\rho)$, figura 3.3, iremos inserir elementos para obtermos a permutação simples equivalente. Os elementos necessários em cada um dos ciclos não são inseridos de forma aleatória. Por exemplo, pegando o ciclo (0 6 4 2), partindo do elemento 0 deste ciclo, para obtermos o próximo elemento deste ciclo, devemos caminhar no grafo, passando por um par de arestas de realidade e de desejo, alcançando o próximo no ciclo que é o elemento 6. Após o elemento 6 o próximo no ciclo será o 4. Como para ciclos simples não podemos ter mais de três elementos, o próximo no ciclo referente ao elemento 4 terá de apontar para o elemento 0, fechando o ciclo e este virando simples. Analisando o elemento 4, percebemos que o par de arestas direcionadas deste nos leva ao elemento 2, para que isso não ocorrá, iremos inserir um elemento na permutação que irá alterar a configuração do diagrama de realidade e desejo $G(\rho)$, passando o elemento 4 ter como próximo no ciclo o elemento 0, fechando o ciclo em (0 6 4).

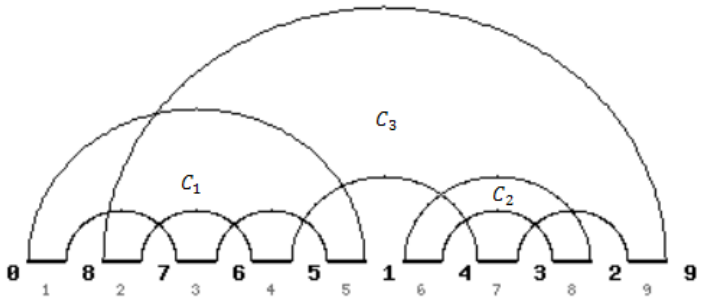

Figura 3.4: Diagrama de realidade e desejo que representa a permutação ρ = [8 7 6 5 1 4 3 2].

Ao inserirmos um elemento que consta na permutação, devemos somar +1 em todos outros elementos maior ou igual a ele, assim a permutação continua com elementos distintos. Após a inserção do primeiro elemento em ρ podemos visualizar na figura 3.4 a nova configuração do digrama de realidade e desejo. Quebramos o ciclo (0 6 4 2) de ρ em um ciclo de tamanho três e outro de tamanho dois. O segundo ciclo de tamanho quatro da permutação original também passa pelo procedimento, no final teremos apenas 1-ciclo, 2-ciclo e 3-ciclo.

Em nossa implementação, todos elementos são inseridos em um mesmo momento, para isso criamos os arrays F_3 e F_4:

- O array F_3 guarda o incremento que cada elemento da permutação deve sofrer para a nova permutação não possuir elementos repetidos. Em $G(\rho)$ temos $F_3 = \{0,\ 1,\ 2,\ 2,\ 2,\ 2,\ 2,\ 2\}$, isso quer dizer que apenas o elemento 0 do primeiro ciclo da permutação não sofre acréscimo.

- O array F_4 representa a soma dos arrays F_2 e F_3, nos dando os ciclos da permutação original, sem elementos inseridos mas com os elementos re-rotulados, $F_4 = \{0,\ 8,\ 6,\ 4,\ 2,\ 9,\ 7,\ 5\}$. Repare que faltam os elementos 1 e 3, justamente os elementos que vamos inserir.

Tomando F_1, temos o número e tamanho de cada ciclo, por F_4 sabemos quais elementos inserir. Percorrendo apenas os ciclos maiores que três em F_4 e inserindo na quarta posição de cada ciclo um elemento, quebramos cada ciclo em dois (devido suas arestas no grafo passar a apontar para outros elementos). Caso a inserção de apenas um elemento no ciclo não seja suficiente, continuamos a inserir elementos

naquele ciclo. A figura 3.5 demonstra o procedimento para o grafo $G(\rho)$. Ao inserir

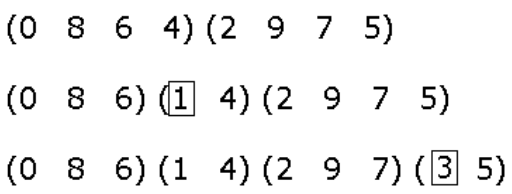

Figura 3.5: Inserindo elementos nos ciclos da permutação.

o primeiro elemento no ciclo re-rotulado (0 8 6 4), transformamos ele nos ciclos (0 8 6) e (1 4), que são ciclos simples. Mas nossa permutação ρ continua não sendo simples, devido a outro ciclo (2 9 7 5) pertencente a $G(\rho)$ ter tamanho superior a três. Neste caso, realizamos o mesmo procedimento no ciclo, inserindo na quarta posição o elemento 3. Com isso o elemento 7 pertencente do ciclo, deixará de ter como próximo no ciclo o elemento 5 e passará a preceder o elemento 2, fechando o ciclo em (2 9 7), criando um novo ciclo com (3, 5). No final do procedimento teremos o array $F_5 = \{0,\ 8,\ 6,\ 1,\ 4,\ 2,\ 9,\ 7,\ 3,\ 5\}$ que corresponde aos ciclos da permutação simples. O número e tamanho dos ciclos da permutação simples criada a partir de ρ é representado pelo array F_1 atualizado, $F_1 = \{4,\ 3,\ 2,\ 3,\ 2\}$, isso significa que temos agora quatro ciclos, dois de tamanho três e dois de tamanho dois. Nosso passo final para realizar o pré-processamento é construir a nova permutação a partir dos novos ciclos, utilizando F_5 e F_1.

Para montarmos nossa permutação simples basta tomar o elemento 0 em F_5, este será o primeiro elemento da permutação. Pegamos o 0 e obtemos seu próximo no ciclo somando +1, o resultado passa a ser o segundo elemento da permutação. Utilizando deste segundo elemento obtemos seu próximo no ciclo e somamos +1, assim teremos o terceiro elemento da permutação. O processo se repete até obtermos a permutação por completo. Conseguimos com o procedimento obter a permutação simples $F_6 = \{9,\ 8,\ 7,\ 3,\ 6,\ 1,\ 5,\ 4,\ 2\}$, representada na figura 3.6.

Agora que obtemos uma permutação simples, mostraremos a implementação da estrutura árvore de permutações proposta por Feng e Zhu [11] para realizarmos as transposições com uma complexidade de tempo melhor que a proposta por Hartman e Shamir [17].

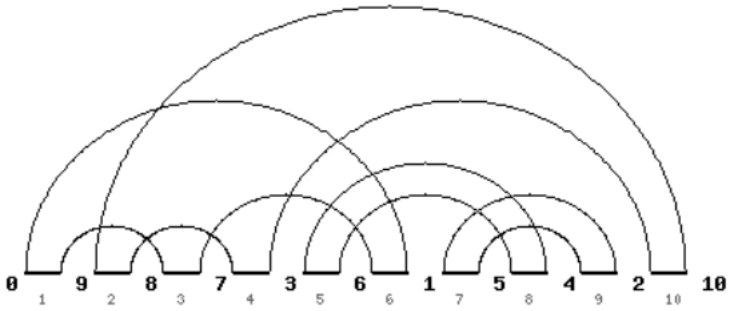

Figura 3.6: Diagrama de realidade e desejo para ρ'

3.2 Árvore de Permutação

A estrutura árvore de permutações proposta por Feng e Zhu [11] é uma árvore binária balanceada. Podemos definir esta estrutura como sendo uma árvore na qual as alturas das duas sub-árvores de cada um dos *nós* nunca diferem em mais de uma unidade. O balanceamento de um *nó* é igual a diferença entre as alturas de suas sub-árvores à esquerda e à direita. Portanto, cada *nó* de uma árvore balanceada tem balanceamento igual a -1, 0 ou 1, dependendo da comparação entre as alturas esquerda e direita. Lembrando que a altura de um *nó* n da árvore é o número de *nós* do maior caminho de n até um de seus descendentes.

Ao trabalharmos com árvores, devemos sempre ter uma boa especificação dos atributos e operações referente aos seus *nós*, pois na árvore de permutação cada *nó* representa um elemento da permutação.

Com a implementação da estrutura de dados podemos construir uma árvore binária, a partir dos elementos de uma permutação qualquer. Essa estrutura é projetada para otimizar as operações de transposições aplicadas à permutação. Trabalhando com árvores, conseguimos diminuir a complexidade de uma transposição, que é a operação fundamental do nosso algoritmo aproximativo de ordenação.

Comentaremos nossa implementação falando inicialmente sobre a classe $Node$ e posteriormente descreveremos como construir e operar com esta árvore binária.

3.2.1 A classe Node

A classe $Node$ é uma classe simples e importante para nossa implementação. Cada elemento π_i, $1 \leq i \leq n$ da permutação π é representado por uma instância da classe $Node$. Esta é composta pelo construtor da classe e métodos de acesso aos atributos. Dado uma árvore T, os atributos de um objeto $Node$ nos permitem navegar e alterar valores de T. A seguir detalhamos alguns atributos existentes na classe $Node$, primeiro especificamos os atributos citados por Feng e Zhu [11] posteriormente especificaremos nossa contribuição para melhorar a estrutura.

- **Valor:** Atributo que guarda um inteiro que representa um elemento na permutação.
- **Altura:** Altura de um *nó* na árvore. Folhas possuem altura nula (zero), e cada vez que subimos na árvore atribuímos mais 1 para o *nó*. Através deste atributo podemos observar se a árvore esta balanceada.
- **NumeroDescendentes:** Guarda o número de descendentes de um dado *nó*. Especifica o número de folhas que o *nó* selecionado tem abaixo na hierarquia. Atribuimos ao *nó* folha o número de descendentes igual a um, correspondendo a ele mesmo.
- **NumeroDeFolhas:** Um inteiro que representa a quantidade de elementos da permutação, usado para dimensionar a permutação.
- **ParaEsquerda e ParaDireita:** Cada *nó* por estar em uma árvore binária pode ter um filho a sua esquerda e outro a sua direita, exceto as folhas que não possuem filhos. Para representar os filhos de um *nó* de uma árvore t, criamos os atributos p_esq e p_dir que representam respectivamente outro objeto $nó'$ a esquerda e outro objeto $nó'$ a direita do *nó*. Podemos visitar qualquer *nó* da árvore partindo da raiz utilizando esses métodos. Atráves deste atributo é possível percorrer a árvore binária da raiz até suas folhas.
- **Pai:** Para podermos percorrer uma árvore binária das folhas em direção à raiz necessitamos saber qual *nó* é pai de um dado *nó*, logo criamos o atributo *pai* para representar a hieraquia de um dado *nó*, sendo o valor deste atributo na raiz nulo. Cada *nó* terá como pai o *nó* logo acima dele.

Definidos os atributos especificados por Feng e Zhu para a classe *Node*, definiremos mais dois atributos que simplificarão nosso trabalho posteriormente na estrutura.

- **PróximoNoCiclo e AnteriorNoCiclo:** Estes valores são atribuídos aos *nós* folha da árvore. Tomando o grafo de realidade e desejo de uma permutação π, representado por $G(\pi)$, podemos visualizar o comportamento dos seus ciclos. Dado que cada *nó* folha representa um elemento da permutação, percorremos todos estes *nós* e atribuir ao objeto o seu próximo no ciclo, do mesmo modo capturaremos seu anterior no ciclo e atribuimos ao objeto, no final teremos listas duplamente encadeadas, formadas por *nós* de mesmo ciclo em π.

Descrevemos a classe *nó* com seus atributos, descrevemos agora nossa árvore binária de permutação, a partir de uma permutação simples qualquer. A construção da árvore é realizada na classe PermutationTree da nossa implementação.

Cabe ressaltar que a árvore pode ser usada para descrever uma permutação qualquer, mesmo que não seja simples.

3.2.2 A classe PermutationTree

A nossa classe PermutationTree tem como arcabouço a estrutura proposta por Feng e Zhu [11] para implementação de uma nova estrutura de dados. Esta melhora o tempo de execução para ordenação de uma permutação por transposições. O algoritmo 1,5-aproximativo de Hartman e Shamir [17] para ordenar uma permutação por transposições é realizado com complexidade de tempo $O(n^{3/2}\sqrt{\log n})$. Implementando a classe PermutationTree realizamos as transposições com complexidade de tempo $O(n \log n)$. Os principais métodos desta classe são: O método *Build*, que constrói uma árvore binária com elementos da permutação, o método *Join* que realiza a união de duas árvores mantendo o balanceamento e o método *Split*, que quebra uma árvore em duas correspondentes.

Método Build

O método *Build* recebe como parâmetro um *ArrayList*, que é uma lista de objetos, sendo que cada objeto possui uma posição relativa na memória, identificado na

estrutura por um número inteiro que se inicia no zero. Este ArrayList guarda todos arrays oriundos do pré-processamento $\{F_1, F_2, F_3, F_4, F_5, F_6\}$. Dado que F_6 guarda uma permutação simples obtida do pré-processamento, construimos nossa árvore de permutação a partir de F_6 implementando o algoritmo 3 do capítulo 2. A construção da árvore a partir de F_6 é feita através da criação dos *nós* camada por camada, de forma bottom-up (partindo das folhas em direção a raiz). Desta forma a primeira camada a ser criada será das folhas (elementos da permutação), depois criaremos as camadas superiores. Antes de construirmos as camadas superiores, trabalhamos com os *nós* folha. Percorremos os *nós* folha e atribuímos a cada objeto seu próximo e anterior no ciclo, isso é possível baseando-se nos arrays F_1 e F_5. Continuando com nossa permutação de exemplo, apresentado neste capítulo $\rho_s = [9\ 8\ 7\ 3\ 6\ 1\ 5\ 4\ 2]$, visualizamos na figura 3.7 a configuração da primeira camada com os atributos próximo e anterior no ciclo.

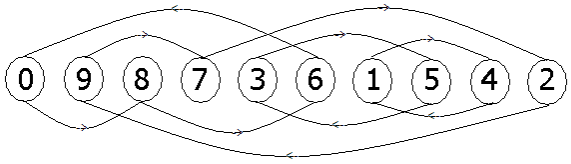

Figura 3.7: Primeira camada da árvore de permutação referente à permutação ρ_s, cada *nó* aponta para seu próximo no ciclo e é apontado pelo seu anterior no ciclo.

Seguindo com a implementação do nosso método *Build*, fizemos outra grande contribuição para a estrutura de Feng e Zhu. Após criarmos a primeira camada da árvore de permutação, colheremos dados desta para implementarmos o que denominamos de estrutura de ciclos.

Nossa estrutura de ciclos será um *ArrayList* contendo três listas encadeadas distintas: Uma lista contendo os ciclos de tamanho par, outra lista com os ciclos de tamanho ímpar maior que um e outra lista com os ciclos de tamanho unitário, ou seja, com elementos que já estão em ordem consecutiva. Como estamos trabalhando com permutações simples, cada lista guarda os ciclos de tamanho 1, 2 ou 3. Utilizamos os atributos das folhas *proximoNoCiclo* e *anteriorNoCiclo* para mapeá-los na

permutação, obtendo informações sobre a quantidade e tamanho de cada ciclo, para alocação. Planejamos esta distribuição devido ao algoritmo para ordenação de uma permutação trabalhar primeiro com ciclos pares e depois com os ímpares maiores que 1.

Sendo a permutação $\pi = [\pi_1\ \pi_2\ \pi_3\ \pi_4\ \pi_5\ \pi_6\ \pi_7\ \pi_8\ \pi_9]$, supondo que seu diagrama $G(\pi)$ possua $c(5)$, sendo $c_1 = (\pi_1, \pi_3)$, $c_2 = (\pi_2, \pi_4)$, $c_3 = (\pi_5, \pi_7, \pi_9)$, $c_4 = (\pi_6)$ e $c_5(\pi_8)$, nossa estrutura de ciclos se comporta conforme, figura 3.8.

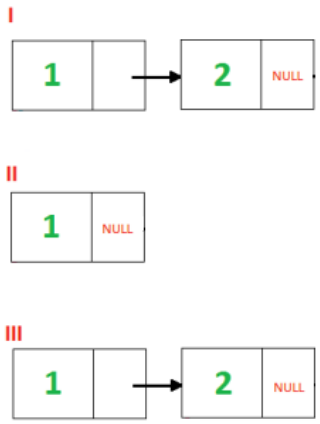

Figura 3.8: Array contendo três listas encadeadas.

Todo ciclo elege um *nó* para representá-lo na estrutura de ciclos, este *nó* passa a fazer parte de uma das listas encadeadas da estrutura. Na primeira lista temos dois *nós* que representam os ciclos pares c_1 e c_2. A segunda lista encadeada possui apenas um *nó*, este é representante do ciclo c_3 (ímpar de tamanho maior que 1). Nossa terceira lista encadeada possui dois *nós* que representam respectivamente os ciclos c_4 e c_5, ciclos unitários.

Após a criação e atribuição de valores à nossa estrutura de ciclos, voltamos nossa atenção para a construção das camadas superiores da árvore de permutação. A nossa implementação volta a seguir os passos do algoritmo 3. Concluindo nossa árvore, o método *Build* retornará o *nó* raiz, a partir dele podemos caminhar por toda a árvore.

Método Join

Seja T_1 uma árvore de permutação correspondente a permutação $\pi_a = [\pi_1,\ ...,\ \pi_m]$ e T_2 uma árvore de permutação correspondente à $\pi_b = [\pi_{m+1},\ ...,\ \pi_n]$. O

método *Join*(T_1, T_2) irá retornar uma árvore T que corresponde à permutação $\pi = [\pi_1, \ ..., \ \pi_m, \ \pi_{m+1}, \ ..., \ \pi_n]$. Implementamos o método *Join* a partir do algoritmo 4 proposto por Feng e Zhu. Apesar do algoritmo 4 proposto descrever apenas o caso onde $H(T_1) \geq H(T_2)$, implementamos o caso onde $H(T_1) < H(T_2)$. Ao montarmos a árvore T, nos preocupamos em atualizar cada atributo dos *nós*. Esta atualização ocorre em relação aos *nós* de T_1 e T_2, que após o *Join* se unem para formar uma única árvore T. Apenas atributos do Objeto *nó* definidos por Feng e Zhu em seu artigo [11] são atualizados ao executar *Join*, como por exemplo: tamanho, número de descendentes, pai e altura. Atributos de nossa contribuição (próximo-anterior no ciclo e estrutura de ciclos) não são modificados por este método e sim em outro momento.

Método Split

Seja T uma árvore de permutação correspondente à permutação $\pi = [\pi_1, \ ..., \ \pi_{m-1}, \ \pi_m, \ ..., \ \pi_n]$. O método $Split(T, m)$ recebe como parâmetro uma árvore T e um inteiro m, que representa onde ocorre a quebra, retornando após a execução duas árvores T_l e T_r, constituídas por *nós* de T, respeitando os respectivos intervalos $[\pi_1, \ ..., \ \pi_{m-1}]$ e $[\pi_m, \ ..., \ \pi_n]$.

Para realizarmos o $Split(T, m)$ precisamos primeiramente obter um caminho das folhas de T até sua raiz, para demarcarmos as partes que constituem as novas árvores. Seja $P = v_0, \ v_1, \ ..., \ v_k$ o caminho do *nó* folha até a raiz r, onde $v_0 = \pi_m$, $v_k = r$. Partindo do *nó* raiz em direção as folhas, se o *nó* alcançado tiver o atributo *NumeroDescendentes* com o mesmo valor de m, descemos na árvore pela direita e adicionamos o *nó* em um array. Caso contrário descemos na árvore para a esquerda e adicionamos o *nó* em um array. Comparamos novamente o atributo *NumeroDescendentes* do *nó* atual com m e repetimos o passo anterior. No final, obtemos o caminho raiz-folha, mas queremos o caminho folha-raiz, logo invertemos o caminho encontrado e teremos v_0, v_1, ... v_r. O algoritmo 4 descrito no capítulo 2 mostra como realizamos o *Split* na árvore partindo do caminho encontrado.

3.3 A classe Transposition

Esta classe contém os métodos que realizam as transposições na permutação desordenada, com objetivo de ordená-la. Os principais métodos da classe são *MakeTransposition* e *Query*. As duas operações foram implementadas com complexidade de tempo $O(nlogn)$.

Dada uma permutação π representada pela árvore binária T, o método MakeTransposition é responsável por realizar a transposição $t(i, j, k)$ sobre a árvore T. Sua implemeção é formalizada pelo algoritmo 6.

As linhas 2,3 e 4 do algoritmo 6 atribuiem as posições i,j e k da transposição aos seus respectivos *nós*. Posteriormente entre as linhas 5 e 10 checamos se os nós são do mesmo ciclo e se este ciclo é orientado. Na linha 11 atualizamos a nossa estrutura de ciclos, devido quando ocorrer mudanças no diagrama de realidade e desejo ao se realizar uma transposição. A atualização da estrutura de ciclos é representada pelo procedimento *updateStructureCycle* algoritmo 7. Após a atualização da estrutura de ciclos, transformamos a árvore T em quatro sub-árvores: T_1 correspondente a $[\pi_1...\pi_{i-1}]$, T_2 correspondente a $[\pi_i...\pi_{j-1}]$, T_3 correspondente a $[\pi_j...\pi_{k-1}]$ e T_4 correspondente a $[\pi_k...\pi_n]$. Juntando as sub-árvores na ordem T_1, T_3, T_2 e T_4 formamos a nossa árvore T com os *nós* reordenados, logo o método retorna T.

O algoritmo 7 atualiza nossa estrutura de ciclos ao ocorrer uma transposição. Na linha 2 testamos se os três *nós* que ocupam as posições i, j e k na permutação pertencem ao mesmo ciclo e se este ciclo é orientado. Caso as proposições forem verdadeiras, na linha 3 retiramos o ciclo da estrutura, pois este será quebrado na transposição, na linha 4 atualizamos os atributos próximo e anterior no ciclo de $nó_i$, $nó_j$ e $nó_k$ devido os três *nós* terem a aresta de realidade modificada no diagrama de realidade e desejo ao realizar a transposição. Na linha 5 adicionamos a estrutura os novos ciclos que passaram a fazer parte. Caso seja verdadeira apenas a proposição que os *nós* são do mesmo ciclo, o algoritmo executa a linha 8, atualizando os atributos próximo e anterior no ciclo de $nó_i$, $nó_j$ e $nó_k$, não ocorrendo mudanças na estrutura de ciclos.

Algoritmo 6: MakeTransposition

Entrada: T, st_ciclos, i, j, k

Saída: T

```
1  início
2      no_i representa a folha de T que ocupa a posição i
3      no_j representa a folha de T que ocupa a posição j
4      no_k representa a folha de T que ocupa a posição k
5      mesmoCiclo ← falso
6      cicloOrientado ← falso
7      se no_i no_j no_k ∈ mesmo ciclo então
8          mesmoCiclo ← verdadeiro
9          se no_i no_j no_k for orientado então
10             cicloOrientado ← verdadeiro
11     updateStructureCycle(st_ciclos, no_i, no_j, no_k, mesmoCiclo, cicloOrientado)
12     (a_l, a_r) = Split(T, i)
13     (b_l, b_r) = Split(a_r, j − i)
14     (c_l, c_r) = Split(b_r, k − j)
15     T_1 = a_l
16     T_2 = b_l
17     T_3 = c_l
18     T_4 = c_r
19     Join(Join(Join(T_1, T_3), T_2), T_4)
20 fim
```

Para casos onde os *nós* representantes das posições i, j e k não pertencerem ao mesmo ciclo, o conjunto pertencerá no máximo a dois ciclos diferentes, devido ao algoritmo de Hartman e Shamir quebrar arestas na transposição de no máximo dois ciclos diferentes. Sendo $nó_i$ e $nó_j$ participantes do mesmo ciclo e $nó_k$ de outro ciclo na estrutura como na linha 10, o algoritmo, na linha 13 descarta ambos os ciclos da estrutura de ciclos e em seguida, na linha 14, atualiza os atributos próximo e anterior no ciclo dos *nós*. Adicionamos á estrutura os novos ciclos que possuem os *nós* atualizados de acordo com a estrutura convencional da linha 15. O mesmo

procedimento é feito para outra combinação dos *nós* em dois ciclos diferentes.

Algoritmo 7: updateStructureCycle

Entrada: T, *st_ciclos*, *orientado*, *mesmoCiclo*, $nó_i$, $nó_j$, $nó_k$

```
1  início
2      se orientado = verdadeiro e mesmoCiclo = verdadeiro então
3          Retira o ciclo de st_ciclos
4          Atualiza os atributos de nó_i, nó_j, nó_k
5          Atribuir os ciclos que contém nó_i, nó_j, nó_k a st_ciclos
6      se orientado = falso e mesmoCiclo = verdadeiro então
7          Atualiza os atributos de nó_i, nó_j, nó_k
8      senão
9          se nó_i e nó_j ∈ ao ciclo então
10             nó_aux_i ← nó que representa o ciclo de nó_i
11             nó_aux_k ← nó que representa o ciclo de nó_k
12             Retira nó_aux_i e nó_aux_k de st_ciclos
13             Atualiza os atributos de nó_i, nó_j, nó_k
14             se nó_i e nó_j ∈ ao ciclo então
15                 Atribuir os ciclos que contém nó_j, nó_k a st_ciclos
16             senão
17                 Atribuir os ciclos que contém nó_j, nó_i a st_ciclos
18         senão
19             nó_aux_i ← nó que representa o ciclo na estrutura
20             nó_aux_j ← nó que representa o ciclo na estrutura
21             Retira nó_aux_i e nó_aux_j de st_ciclos
22             Atualiza os atributos de nó_i, nó_j, nó_k
23             se nó_i e nó_j ∈ ao ciclo então
24                 Atribuir os ciclos que contém nó_k, nó_i a st_ciclos
25             senão
26                 Atribuir os ciclos que contém nó_j, nó_i a st_ciclos
27 fim
```

O algoritmo de Feng e Zhu define um método Query que é utilizado no entrelaçamento de arestas no diagrama de realidade e desejo de uma permutação, este funciona da seguinte forma: Fornecemos um par de arestas de realidade de um ciclo qualquer como parâmetro e o método nos retorna, caso exista, outro par de arestas de outro ciclo, que intersecta com o ciclo dado.

O teorema 2.17 mostra que usando o método *query* na estrutura árvore de permutação, encontramos o par de arestas de realidade que intersecta o par de arestas passado como parâmetro.

Teorema 3.1 (Bafna and Pevzner 1998)**.** *Seja b_i e b_j duas arestas de realidade em um ciclo desorientado C, $i < j$. Seja $\pi_k = max_{i<m\leq j}\pi_m = \pi_{k+1}$, então a aresta de realidade b_k e b_{l-1} pertencem ao mesmo ciclo, e o par $< b_k, b_{l-1} >$ intersecta o par $< b_i, b_j >$.*

Algoritmo 8: Query

Entrada: T, i, j

Saída: *nó*

```
1 início
2     (a_l, a_r) = Split(T, i)
3     (b_l, b_r) = Split(a_r, j − i)
4     T_1 = a_l
5     T_2 = b_l
6     T_3 = b_l
7     Nó = folha da árvore correspondente ao nó T_2; Join(t_1, Join(T_2, T_3))
8 fim
```

O algoritmo 8 encontra um par de arestas de realidade que intersecta o par $< b_i, b_j >$ dado. Primeiro, a árvore T é transformada em três sub-árvores t_1 que corresponde à $[\pi_1...\pi_i]$, T_2 que corresponde a $[\pi_k...\pi_n]$ e T_3 que corresponde à $[\pi_{j+1}...\pi_n]$. Isto pode ser feito em tempo $O(\log n)$ com duas operações de quebra em T. O valor da raiz de T_2 é o maior elemento do intervalo $[\pi_k...\pi_n]$, então percorremos a árvore e retornamos a folha que corresponde a este *nó*. Seja π_k e $\pi_l = \pi_k + 1$. O par $< b_k, b_{l-1} >$ intersecta o par $< b_i, b_j >$.

Teorema 3.2 (Feng and Zhu 2007)**.** *O procedimento Query pode ser realizado com complexidade de tempo* $O(\log n)$*.*

3.4 A classe SortingByTransposition

SortingByTransposition é a classe que implementa o algoritmo 1,5-aproximativo de Hartman e Shamir [17] para o problema de Ordenação por Transposições. O algoritmo 9 que reprenta esta implementação é descrito a seguir.

Primeiro encontra-se uma permutação simples equivalente à permutação da entrada que mantenha o limite inferior da distância de transposição. Os 2-ciclos da permutação são os primeiros a serem quebrados. Estes são quebrados realizando exaustivamente 2-transposições consumindo os ciclos pares.

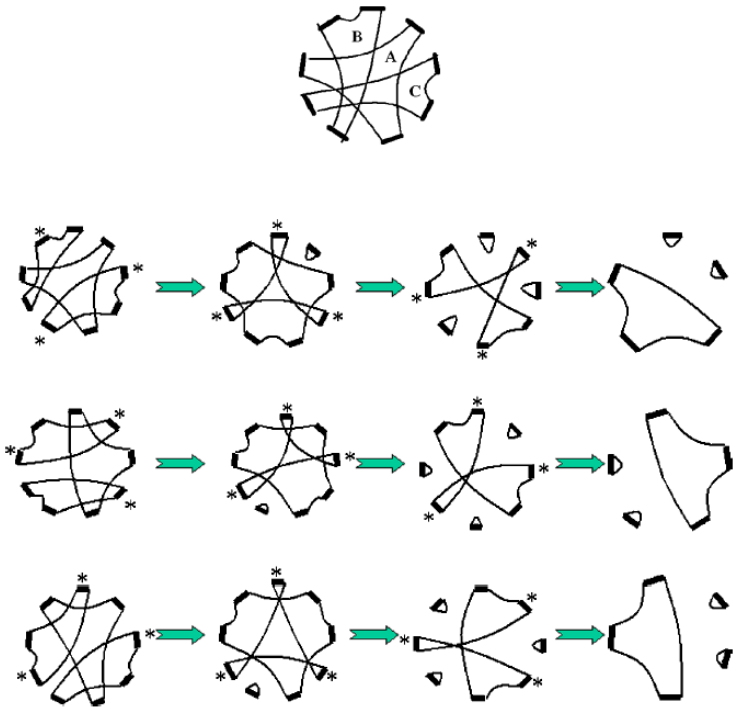

Figura 3.9: Os três casos possíveis para 3-ciclos não orientados. Sendo que um deles é saturado por outros dois e nenhum par é entrelaçado e dois deles não intersectam. Em cada caso, uma $(0, 2, 2)$-sequência de transposições é mostrada.

Neste momento, só possuímos 1-ciclo e 3-ciclos na permutação. Os 3-ciclos orientados são quebrados por uma 2-transposição. Entretanto, os 3-ciclos não orientados não são quebrados trivialmente. Estes são quebrados da seguinte forma: Seja A um 3-ciclos não orientado e a um par de arestas adjacentes do ciclo. Utilizando o método

Query e passando como atributo o par de arestas a do ciclo A, obteremos de um ciclo B um par de arestas adjacentes b que cruza com o par de arestas a do ciclo A. Caso o ciclo B seja orientado, aplicamos uma (2)-transposição nele. Ocorrendo de B não ser orientado, checamos se A e B são entrelaçados. Em sendo, aplicamos uma $(0, 2, 2)$-trasposição, senão obtemos outro par de arestas adjacentes c de um ciclo C que cruza com o par de arestas adjacentes a' do ciclo A. Para esta configuração dos ciclos aplicamos uma $(0, 2, 2)$-trasposição, figura 3.9. Com isso, conseguimos ordenar nossa permutação. Esta classe contém apenas o método **Main**, que é o método de execução da aplicação.

Utilizamos em *SortingByTransposition* instâncias das outras classes que implementamos, com isso dividimos o trabalho de ordenar uma permutação por transposições. Sendo cada classe responsável por uma parte do processamento. Começamos instânciando a classe **Permutation**, que contém o método para transformar uma permutação π em simples $\hat{\pi}$. Em sequência, instanciamos a classe **PermutationTree**, onde o método *Build* constroi a árvore de permutação de $\hat{\pi}$. Após construirmos a árvore, obtemos dela os ciclos de $\hat{\pi}$ e os adicionamos em nossa **estrutura de ciclos**.

Manipulando tanto a estrutura de ciclos quanto a árvore de permutação, conseguimos ordenar a permutação $\hat{\pi}$ de acordo com o algoritmo 9.

Algoritmo 9: Sort

Entrada: permutação

Saída: permutação ordenada

```
1  início
2      Transforme a permutação π em uma permutação π̂ simples
3      enquanto st_ciclos contiver um 2-ciclo faça
4          aplique uma 2-transposição.
5      enquanto st_ciclos contiver um 3-ciclo faça
6          Escolha um par adjacente de arestas de realidade a do ciclo A
7          se A é orientado então
8              Aplique a 2-transposição atuante sobre as arestas de A
9          senão
10             Escolha um par adjacente b (de um ciclo B) que cruza com a
11             se B é orientado então
12                 Aplique a 2-transposição atuante sobre as arestas de B
13             senão
14                 se Os ciclos A e B são entrelaçados então
15                     Aplique uma (0, 2, 2)-sequência
16                 senão
17                     {Existe um par adjacente a' em A que não cruza com B}
18                     Escolha um par adjacente c (de um ciclo C) que cruza com
                       o par a'
19                     {O ciclo A é fragmentado pelos ciclos B e C }
20                     Aplique uma (0, 2, 2)-sequência
21     Transforme a sequência de transposições que ordenaram π̂ em uma
       sequência de transposições que ordenem π
22 fim
```

3.5 Otimizando o número de transposições para o algoritmo de Hartman e Shamir

Analisando o algoritmo de Hartman e Shamir [17] para o problema de Ordenação por Transposições percebemos que poderíamos melhorar o resultado do cálculo da distância de transposição para ordenar uma permutação. O custo desta mudança será o aumento da complexidade do algoritmo de $O(n \log n)$ para $O(n^2 \log n)$.

Propomos quebrar os 3-ciclos da permutação em uma ordem diferente realizada no algoritmo de Hartman e Shamir [17]. Após quebrarmos os ciclos pares da permutação, percorreremos a *Estrutura de Ciclos* e quebramos todos os 3-ciclos orientados. Com isso os 3-ciclos não orientados passam a ser os únicos ciclos da permutação para quebrarmos. Realizamos então uma $(0, 2, 2)$-sequência de transposições, sendo que a primeira transposição não aumenta o número de ciclos ímpares, mas ordena um ciclo e garante que nas duas próximas transposições o número de ciclos ímpares irá aumentar. Ao realizar a $(0, 2, 2)$-sequência de transposições, pode ocorrer de ciclos não orientados se tornarem orientados. Percorremos novamente a *Estrutura de Ciclos* em busca dos ciclos orientados repetindo o processo descrito. Esta ordem proposta de quebra dos 3-ciclos é realizada até obtermos a permutação ordenada.

3.5.1 Comparando os Algoritmos

Iremos agora realizar alguns exemplos utilizando o algoritmo de Hartman e Shamir [17] implementado e comparar com a sua versão que otimiza o número de transposições.

Para organizar e obter as sequências de diferentes seres vivos e calcular a distância de transposição entre eles, utilizamos o banco de dados do *National Center for Biotechnology Information* (NCBI), banco de dados de sequências de proteínas e mapeamento de genomas. Esta exemplificação está organizada em etapas: Etapa 1.0, que mostra como o banco de dados NCBI (veja: http://www.ncbi.nlm.nih.gov/) gera e compara as sequências de genes. Etapa 1.1, que reorganiza a sequência gerada na etapa anterior. Etapa 2.0, nesta etapa aplica-se nosso programa implementado na versão normal e na versão otimizada. Por fim, exibimos o resultado da comparação das sequências analisadas.

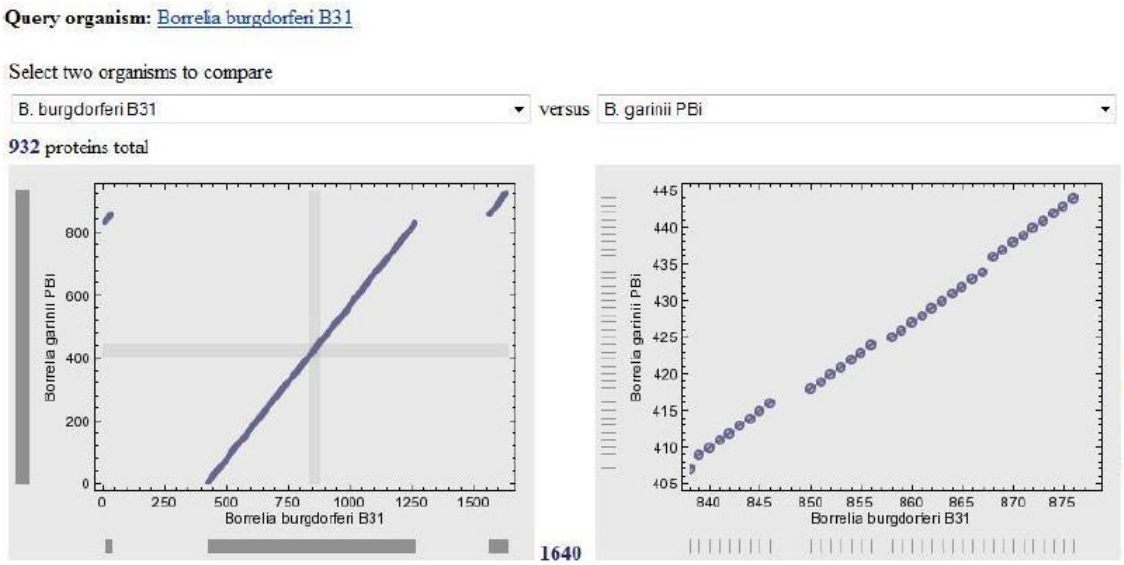

Figura 3.10: Gráficos gerados pelo Gene Plot comparando microorganismos do gênero Borrelia Fonte: NCBI.

Etapa 1.0 O processo de obtenção de dados se fez através da comparação dois a dois de algumas sequências de genes disponíveis dos microorganismos, em sua maioria de bacilos e de bactérias, através do programa *Gene Plot* que se encontra disponível no site do NCBI. O programa permite selecionar qualquer microorganismo de sua lista e comparar a sequência de seus genes com outro microorganismo do mesmo gênero, além de mostrar a representação gráfica da comparação realizada. Como exemplo, temos os gráficos da figura 3.10. Os microorganismos do gênero *Borrelia* se diferem unicamente por transposição, devido ao deslocamento duplo mostrado no primeiro gráfico. O gráfico da direita é uma ampliação do primeiro no ponto indicado.

Etapa 1.1 O Gene Plot compara organismos aos pares. As sequências obtidas através do Gene Plot comumente apresentam genes repetidos. Ao remover estes genes repetidos, ambas sequências terão os mesmos genes, mas esses genes vão estar em uma ordem diferente. Então, ordenamos uma sequência em relação à outra para o resultado da distância de transposição ser em relação à permutação identidade.

Etapa 2.0 A permutação é ordenada na aplicação. Nossa aplicação retorna

então o cálculo da distância de transposição entre os microorganismos para as duas versões do algoritmo. A comparação da execução do algoritmo de Hartman e Shamir em relação a versão que otimiza o número de transposições entre microorganismos de gêneros distintos, foi postada na figura 3.11. Esta exibe nas duas primeiras colunas os microorganismos comparados, na coluna seguinte o número de genes de ambos. Por fim as duas últimas colunas representam respectivamente o número de transposições e o tempo gasto pelo programa para realizá-las. Sendo que à esquerda da barra representa o algoritmo original e à direita da barra é para o algoritmo otimizado. Visualizando a figura 3.11 referente ao nosso experimento percebemos que

Microorganismo 1	Microorganismo 2	N° de Genes	Número de Transposições/OT	Tempo Gasto/OT (segundos)
Acidovorax A.avenae subsp. citrulli AAC00-1	**A.sp.JS42**	2554	968/938	0,265/0,406
Arthobacter A.aurescens TCI	**A.sp.FB24**	3074	629/559	0,187/0,109
Azoarcus A sp. BH72	**A. EbN1**	2208	1052/950	0,297/0,453
B.fragilis_NCTC9343	**B.thetaiotaomicronVPI-5482**	2542	921/846	0,25/0,359
B.bacilliformisKC583	**B.henselae_str.Houston-1**	999	383/381	0,031/ 0,063
B.adolescentisATCC15703	**B.longumNCC2705**	1196	581/553	0,078/ 0,11
B.bronchiseptica RB50	**B.parapertussis 12822**	4077	303/303	0,037/ 0,047
B.cenocepaciaAU1054	**B.cepaciaAMMD**	5087	2136/2106	1,828/2,515
B.cenocepaciaAU1054	**BmalleiATCC23344**	3311	1672/1609	0,875/ 1,219
B.melitensis16M	**B.suis1330.txt**	2845	2129/2129	0,806/1,609

Figura 3.11: Execuções do algoritmo de Hartman e Shamir para as versões original e otimizada

o número de transposições realizadas utilizando o algoritmo de Hartman e Shamir modificado é menor ou no pior caso igual ao algoritmo original. Isso ocorre devido o algoritmo modificado realizar na quebra dos 3-ciclos sempre uma +2-transposição, e somente quando não for possivel realizar uma $(0, 2, 2)$-transposição. Observa-se

também que o tempo gasto para realizar a ordenação de uma permutação é maior para o algoritmo modificado, isto ocorre devido a quebra dos 3-ciclos na estrutura de ciclos ser realizada em tempo $O(n^2)$, no algoritmo original o consumo da lista de ciclos se faz em tempo $O(n)$.

Capítulo 4

Conclusão

Neste trabalho estudamos e implementamos o algoritmo 1,5-aproximativo de Hartman e Shamir [17] para o problema de ordenação por transposições, utilizando a estrutura de dados proposta por Feng e Zhu [11]. Propomos e implementamos também uma modificação no algoritmo de Hartman e Shamir com relação à ordem de quebra dos ciclos de uma permutação com o objetivo de conseguir ordenar uma permutação com um menor número de transposições que o algoritmo original.

Ao implementar a estrutura de dados proposta por Feng e Zhu, contribuímos com a criação de novos atributos representados nos elementos da permutação. Percebemos que seria interessante e vantajoso para cada elemento da permutação guardar como atributo, além da posição que ocupa na permutação, também dados sobre o ciclo a que pertence no diagrama de realidade e desejo. Feito isto, conseguimos mapear todos os ciclos da permutação.

Outra contribuição para a implementação da estrutura de Feng e Zhu foi a criação de uma estrutura para gerenciar os ciclos da permutação. Esta estrutura é formada por um *ArrayList* de *LinkedList* (Listas Encadeadas), onde as listas pertencentes ao array são constituídas por ciclos com propriedades semelhantes. A separação dos ciclos de uma permutação por semelhança nos ajuda a realizar as transposições no algoritmo 1,5-aproximativo de Hartman e Shamir [17].

A estrutura de Feng e Zhu nos dá uma menor complexidade na realização das transposições em uma permutação utilizando árvores binárias balanceadas. Implementamos o algoritmo de Hartman e Shamir utilizando esta estrutura somada com nossa contribuição. Na implementação modificamos a ordem de quebra dos ciclos

proposta no algoritmo de Hartman e Shamir e passamos a quebrar os ciclos usando a estrutura de ciclos. Por fim, propusemos a modificação no algoritmo de Hartman e Shamir: primeiro quebramos todos os ciclos orientados, depois consideramos os ciclos não orientados para orientá-los e posteriormente quebrá-los. Desta forma, conseguimos diminuir o número final de transposições utilizadas para inúmeros casos.

Referências Bibliográficas

[1] BAFNA, V., PEVZNER, P., 1996, "Genome Rearrangements and Sorting by Reversals", *SIAM Journal on Computing*, v. 25, pp. 272.

[2] BAFNA, V., PEVZNER, P. A., 1998, "Sorting by Transpositions", *SIAM J. Discret. Math.*, v. 11 (May), pp. 224–240. ISSN: 0895-4801. doi: 10.1137/S089548019528280X. Disponível em: <`http://portal.acm.org/citation.cfm?id=292280.292293`>.

[3] BERGERON, A., 2005, "A very elementary presentation of the Hannenhalli-Pevzner theory", *Discrete Applied Mathematics*, v. 146, n. 2, pp. 134–145. ISSN: 0166-218X.

[4] BERMAN, P., HANNENHALLI, S., KARPINSKI, M., 2002, "1.375-approximation algorithm for sorting by reversals", *Algorithms ESA 2002*, pp. 401–408.

[5] BULTEAU, L., FERTIN, G., RUSU, I., 2010, "Sorting by Transpositions is Difficult", *Arxiv preprint arXiv:1011.1157*.

[6] CAPRARA, A., 1997, "Sorting by reversals is difficult". In: *Proceedings of the first annual international conference on Computational molecular biology*, p. 83. ACM. ISBN: 0897918827.

[7] CHRISTIE, D., 1998, *Genome Rearrangement Problems*. Department of Computing Science, University of Glasgow, University of Glasgow.

[8] DIAS, Z., MEIDANIS, J., 2002, "Sorting by Prefix Transpositions". In: *String processing and information retrieval: 9th International Symposium,*

SPIRE 2002, Lisbon, Portugal, September 11-13, 2002: proceeedings, p. 65. Springer Verlag. ISBN: 3540441581.

[9] ELIAS, I., HARTMAN, T., 2006, "A 1.375-approximation algorithm for sorting by transpositions", *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pp. 369–379. ISSN: 1545-5963.

[10] ERIKSEN, N., 2001, "(1+ varepsilon)-Approximation of Sorting by Reversals and Transpositions", *Algorithms in Bioinformatics*, pp. 227–237.

[11] FENG, J., ZHU, D., 2007, "Faster algorithms for sorting by transpositions and sorting by block interchanges", *ACM Trans. Algorithms*, v. 3 (August). ISSN: 1549-6325. doi: http://doi.acm.org/10.1145/1273340.1273341. Disponível em: <`http://doi.acm.org/10.1145/1273340.1273341`>.

[12] FEUK, L., MACDONALD, J., TANG, T., et al., 2005, "Discovery of human inversion polymorphisms by comparative analysis of human and chimpanzee DNA sequence assemblies", *PLoS Genet*, v. 1, n. 4, pp. e56.

[13] FORTUNA, V., MEIDANIS, J., WALTER, M., et al., 2006, *Distâncias de transposiçao entre genomas*. Tese de Doutorado, Biblioteca Digital da Unicamp.

[14] GU, Q., PENG, S., SUDBOROUGH, H., 1999, "A 2-approximation algorithm for genome rearrangements by reversals and transpositions* 1", *Theoretical Computer Science*, v. 210, n. 2, pp. 327–339. ISSN: 0304-3975.

[15] HANNENHALLI, S., PEVZNER, P., 1999, "Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals", *Journal of the ACM (JACM)*, v. 46, n. 1, pp. 1–27. ISSN: 0004-5411.

[16] HARTMAN, T., 2003, "A simpler 1.5-approximation algorithm for sorting by transpositions". In: *Combinatorial pattern matching*, pp. 156–169. Springer.

[17] HARTMAN, T., SHAMIR, R., 2006, "A simpler and faster 1.5-approximation algorithm for sorting by transpositions", *Inf. Comput.*, v. 204 (February),

pp. 275–290. ISSN: 0890-5401. doi: http://dx.doi.org/10.1016/j.ic.2005.09.002. Disponível em: <http://dx.doi.org/10.1016/j.ic.2005.09.002>.

[18] KESMIR, C., NOORT, V., BOER, R., et al., 2003, "Bioinformatic analysis of functional differences between the immunoproteasome and the constitutive proteasome", *Immunogenetics*, v. 55, n. 7, pp. 437–449. ISSN: 0093-7711.

[19] LABARRE, A., 2006, "New bounds and tractable instances for the transposition distance", *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, pp. 380–394. ISSN: 1545-5963.

[20] LIN, G., XUE, G., 1999, "Signed genome rearrangement by reversals and transpositions: models and approximations". In: *Proceedings of the 5th annual international conference on Computing and combinatorics*, pp. 71–80. Springer-Verlag. ISBN: 3540662006.

[21] MEIDANIS, J., DIAS, Z., 2000, "An alternative algebraic formalism for genome rearrangements", *Comparative Genomics: Empirical and Analytical Approaches to Gene Order Dynamics, Map Alignment and Evolution of Gene Families*, pp. 213–223.

[22] MEIDANIS, J., WALTER, M., DIAS, Z., 2000, "Reversal distance of signed circular chromosomes", *Unpublished*.

[23] NADEAU, J., TAYLOR, B., 1984, "Lengths of chromosomal segments conserved since divergence of man and mouse", *Proceedings of the National Academy of Sciences of the United States of America*, v. 81, n. 3, pp. 814.

[24] PALMER, J., HERBON, L., 1988, "Plant mitochondrial DNA evolved rapidly in structure, but slowly in sequence", *Journal of Molecular Evolution*, v. 28, n. 1, pp. 87–97. ISSN: 0022-2844.

[25] SETUBAL, J., MEIDANIS, J., SETUBAL-MEIDANIS, ., 1997, *Introduction to computational molecular biology*. Baffins Lane, Chichester, West Sussex PO19 1UD, Pws Publishing. ISBN: 0534952623.

[26] WALTER, M., DIAS, Z., MEIDANIS, J., 2002, "A new approach for approximating the transposition distance". In: *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pp. 199–208. IEEE, . ISBN: 0769507468.

[27] WALTER, M., CURADO, L., OLIVEIRA, A., 2003, "Working on the problem of sorting by transpositions on genome rearrangements". In: *Combinatorial pattern matching*, pp. 372–383. Springer.

[28] WALTER, M. E. M. T., FREIRE, A. L., MOURA, C., et al., 2004, "An approximation algorithm for the problem of sorting by transversals", pp. 191–195.

[29] WALTER, M., DIAS, Z., MEIDANIS, J., 2002, "Reversal and transposition distance of linear chromosomes". In: *String Processing and Information Retrieval: A South American Symposium, 1998. Proceedings*, pp. 96–102. IEEE, . ISBN: 0818686642.

[30] WATSON, J., CRICK, F., 2003, "A structure for deoxyribose nucleic acid", *A century of Nature: twenty-one discoveries that changed science and the world*, p. 82.