Universidade Federal do ABC
Centro de Matemática, Computação e Cognição
Relatório Final para o programa IC – Edital 01/2019

Problemas de Transformação de Strings por Operações de Rearranjo

Aluno:
Gustavo da Silva Teixeira

Supervisor:
Carla Negri Lintzmayer

Setembro/2020

Resumo

Nos *Problemas de Transformação de Strings (ou Ordenação de Permutações) por Operações de Rearranjo*, dadas duas sequências de símbolos – onde cada símbolo aparece o mesmo número de vezes em cada sequência – e um conjunto de operações de rearranjo permitidas, que trocam as posições de determinado conjunto de elementos das sequências, deseja-se encontrar uma sequência de menor custo, formada por operações de rearranjo, que transforme uma sequência de símbolos na outra.

Se considerarmos que as sequências possuem apenas uma cópia de cada símbolo, podemos representá-las como *permutações*, sendo esta a abordagem mais tradicional para lidar com os problemas de Transformação por Operações de Rearranjo. Porém, outra abordagem é a que considera que há pelo menos um símbolo com mais de uma cópia dentro das sequências. Nestes casos, a representação se dá por meio de *strings*.

Desde o final da década de 1970, quando Gates e Papadimitriou publicaram sobre limitantes para o *Problema das Panquecas* (ou *Ordenação de Permutações por Reversões de Prefixo*), vários trabalhos foram desenvolvidos a fim de obter resultados para problemas relacionados. Os principais esforços têm por objetivo encontrar algoritmos que retornem soluções próximas à solução ótima, visto que grande parte dos problemas relacionados são *NP-difíceis*. No entanto, comparando a abordagem que considera strings com a abordagem que considera permutações, ainda pode-se observar a falta de resultados algorítmicos na literatura para os problemas sobre strings.

Esta Iniciação Científica teve por objetivo estudar os Problemas de Transformação de Strings por Operações de Rearranjo, do ponto de vista teórico, investigando técnicas algorítmicas e resultados existentes na área até então.

Este projeto descreve os resultados obtidos, principalmente, do estudo do Problema de Transformação de Strings por Reversões, como a busca por soluções ótimas, o desenvolvimento de algoritmos, desde ideias simples até a implementação de um Algoritmo Genético para o problema, e a análise comparativa entre os resultados dos algoritmos aqui propostos.

Sumário

1 Introdução

Descrito em 1975 [13], o *Problema das Panquecas* consiste em um garçom que, diante de uma pilha de panquecas de tamanhos distintos, a fim de servi-las com uma melhor apresentação, as rearranja invertendo a ordem das panquecas no topo da pilha, quantas vezes for necessário, até que a pilha fique ordenada pelo tamanho das panquecas, com a menor panqueca no topo e a maior na base. A partir deste cenário, o problema é: dada uma pilha com n panquecas, qual o número mínimo de operações de inversão de topo necessárias para ordenar a pilha?

No problema acima, podemos representar a pilha de panquecas como uma *permutação*, onde cada um de seus elementos representa o tamanho de uma panqueca (o valor do i-ésimo elemento da permutação representa o tamanho da panqueca na i-ésima posição da pilha, do topo à base). Uma *reversão de prefixo* é a operação que inverte a ordem dos primeiros elementos da permutação. Com essa representação, queremos encontrar o menor número de reversões de prefixo que ordene a permutação. O nome dado ao problema, neste formato, é *Problema de Ordenação por Reversões de Prefixo*.

Nota-se, assim, uma relação de problemas deste tipo com problemas de Biologia Computacional em que, dados os genomas de dois organismos que compartilham genes similares, deseja-se saber quais operações de mutação ocorreram entre eles. Este pode ser considerado um dos desafios da ciência moderna, que ajudaria a entender como a evolução dos seres vivos ocorre. Um *rearranjo de genoma* é um tipo de mutação de larga escala que pode ocorrer em um genoma. Assim, a distância evolutiva entre dois organismos pode ser inferida pelo número de rearranjos que aconteceram para transformar o genoma de um organismo no genoma do outro.

Entretanto, permutações são sequências em que cada elemento ocorre apenas uma vez. Considerando a relação com os problemas em genomas, se iniciaram, naturalmente, estudos de problemas relacionados, porém com sequências em que os elementos podem se repetir, chamadas de *strings*.

O objetivo principal da pesquisa foi analisar, sob um ponto de vista teórico e computacional, os problemas já propostos e resultados já existentes para estes, bem como as técnicas empregadas para a obtenção dos resultados, a fim de propor novos algoritmos. Dentro deste contexto, este trabalho descreve os resultados obtidos da investigação, principalmente, do Problema de Transformação de Strings por Reversões, para o qual propusemos alguns algoritmos, com níveis diferentes de sofisticação, a fim de buscar soluções de melhor qualidade para o problema. Os algoritmos aqui propostos foram implementados e, a partir de testes em determinados conjuntos de entradas, foi feita uma análise comparativa entre os resultados obtidos por cada um deles.

O restante do documento está dividido como segue. Na Seção 2, definimos conceitos e notações necessários para o entendimento dos problemas abordados, dos resultados existentes e dos algoritmos aqui propostos; na Seção 3, descrevemos os métodos utilizados as atividades desenvolvidas durante o período de pesquisa; na Seção 4, citamos diferenças importantes entre permutações e strings, a fim de destacar resultados em permutações que não têm equivalência em strings; na Seção 5, descrevemos os algoritmos desenvolvidos e os métodos heurísticos aplicados; na Seção 6, descrevemos como foram realizados os testes para comparar a qualidade das soluções dos algoritmos propostos e mostramos os resultados obtidos após os experimentos; por fim, na Seção 7, apresentamos nossas conclusões sobre o trabalho desenvolvidos e as perspectivas futuras dentro do tema trabalhado.

Algumas seções, que já estavam presentes no relatório parcial e se referem a resultados de atividades realizadas no primeiro semestre de pesquisa, foram incluídas como apêndices do presente relatório. No Apêndice A, elencamos alguns resultados importantes para os problemas de ordenação de permutações e de transformação de strings por reversões, além de citar problemas relacionados; no Apêndice B, descrevemos o estudo feito sobre soluções ótimas para os problemas estudados, a fim de identificar padrões que auxiliassem na elaboração de algoritmos.

2 Definições e notação

Uma *permutação* é uma tupla $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, de comprimento $|\pi| = n$, com elementos $\pi_i \in \{1, 2, \ldots, n\}$, onde $|\pi_i| \neq |\pi_j| \leftrightarrow i \neq j$, isto é, em que não há elementos repetidos. Em uma *permutação com sinais*, cada elemento tem um sinal "+" ou "−" associado. Em uma *permutação sem sinais*, os sinais são omitidos.

A *permutação identidade* $\iota = (1, 2, 3, \ldots, n)$ é a permutação de n elementos na qual $|\pi_i| < |\pi_j| \leftrightarrow i < j$. Com esta representação, podemos considerar que o problema de transformar uma permutação em outra, de mesmo tamanho, se reduz a transformar uma permutação na permutação identidade, i. e., *ordená-la*.

Dada uma permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, sua *permutação inversa*, denotada por π^{-1}, é a permutação obtida ao trocarmos o valor de cada um dos elementos π_i por suas respectivas posições i, com $1 \leq i \leq n$. Isto significa que $\pi^{-1}_{\pi_i} = i$, para $1 \leq i \leq n$.

Definimos o *grupo simétrico* S_n como sendo o conjunto que contém todas as $n!$ permutações sem sinais possíveis de comprimento n, ou as $2^n n!$ permutações com sinais possíveis.

Seja uma permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, de comprimento n. A versão estendida de π, denotada por π^l, é obtida ao adicionarmos a π os elementos $\pi_0 = 0$ e $\pi_{n+1} = n + 1$, ou seja, $\pi^l = (0, \pi_1, \pi_2, \ldots, \pi_n, n + 1)$.

Seja π^l a versão estendida da permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$. Dizemos que um par (π_i^l, π_{i+1}^l), com $0 \leq i \leq n$, é um *breakpoint de reversão* de π se $\pi_{i+1}^l - \pi_i^l \neq 1$. Quando $|\pi_{i+1}^l - \pi_i^l| > 1$, dizemos que o par (π_i^l, π_{i+1}^l) é um *breakpoint forte*.

Uma *string* $S = s_1 s_2 \ldots s_n$ é uma sequência de *elementos* s_i, com $1 \leq i \leq n$, tirados de um *alfabeto* $\Sigma = \{0, 1, 2, \ldots\}$ de *símbolos*, que pode conter símbolos repetidos. Se o alfabeto tem tamanho $|\Sigma| = k$, uma string descrita sobre este alfabeto é chamada de *string k-ária*. Uma *string com sinais* é uma string onde cada elemento possui um sinal "+" ou "−" associado. Quando não há informação sobre a orientação dos elementos, os sinais são omitidos e a string é dita *sem sinais*. Note que, por definição, uma permutação é uma string na qual os elementos do alfabeto não se repetem. O *comprimento* de uma string $S = s_1 s_2 \ldots s_n$ é denotado por $|S|$ e representa a quantidade de posições para símbolos que ela possui, i.e., $|S| = n$.

Uma *substring* de uma string S de comprimento n, é uma string $S' = s_i s_{i+1} \ldots s_{j-1} s_j$, onde $1 \leq i \leq j \leq n$, e $s_i s_{i+1} \ldots s_{j-1} s_j$ aparece consecutivamente em S.

Um *prefixo* de uma string S é uma substring de S que contém seu primeiro elemento. O *maior prefixo comum* entre duas strings S e T é a substring maximal $L = s_1 s_2 \ldots s_{p-1} s_p$ tal que, para todo $1 \leq i \leq p$, $s_i = t_i$. O comprimento do maior prefixo comum entre S e T é denotado por $lcp(S, T)$.

Um *sufixo* de uma string S é uma substring de S que contém seu último elemento. O *maior sufixo comum* entre duas strings S e T é a substring maximal $L = s_j s_{j+1} \ldots s_{n-1} s_n$ tal que, para todo $j \leq i \leq n$, $s_i = t_i$. O comprimento do maior sufixo comum entre S e T é denotado por $lcs(S, T)$.

Denotamos por $f(S, x)$ a quantidade de ocorrências do símbolo x na string S. Da mesma forma, denotamos por $f(S, S')$ a quantidade de ocorrências da *substring* S' na string S.

Duas strings S e T são ditas *compatíveis* se (i) ambas têm o mesmo comprimento n, (ii) ambas são descritas sobre o mesmo alfabeto Σ, e (iii) $f(S, x) = f(T, x)$, para todo $x \in \Sigma$. Nos problemas aqui tratados, sempre consideraremos que as strings analisadas são compatíveis.

Uma *classe de equivalência*, denotada por $\mathcal{L} = (a_0, a_1, \ldots, a_{k-1})$, é o conjunto de strings compatíveis com exatamente a_i ocorrências de cada símbolo $i \in \Sigma$, onde $|\Sigma| = k$. Note que, por esta definição, $n = a_0 + a_1 + \cdots + a_{k-1}$, i.e., a soma da quantidade de ocorrências de cada símbolo é igual ao comprimento das strings.

Definimos a *string identidade* S_ι de uma classe de equivalência como sendo a string $S_\iota = s_1 s_2 \ldots s_n$, onde $s_i \leq s_{i+1}$, para $1 \leq i < n$, ou seja, é uma string ordenada de forma não decrescente pelo valor de seus elementos.

Uma *reversão* $\rho(i, j)$, com $1 \leq i < j \leq n$, representa o rearranjo $(1\ 2\ \ldots\ i-2\ i-1\ \underline{j\ j-1 \ldots i+1\ i}\ j+1\ j+2\ \ldots\ n-1\ n)$. Isso significa que, quando aplicada a uma string $S = s_1 s_2 \ldots s_n$, a reversão inverte o segmento que vai da posição i à posição j, transformando

a string S em $S \cdot \rho(i,\, j) = s_1 \ldots s_{i-1}\ \underline{s_j\ s_{j-1} \ldots s_{i+1}\ s_i}\ s_{j+1} \ldots s_n$.

Uma *reversão de prefixo*, denotada $\rho_p(j)$, com $1 < j \leq n$, é uma reversão que contém, necessariamente, o primeiro elemento da string, i.e., equivale a $\rho(1, j)$. Uma *reversão de sufixo*, denotada $\rho_s(i)$, com $1 \leq i < n$, é uma reversão que contém, necessariamente, o último elemento da string, i.e., equivale a $\rho(i, n)$.

Uma *reversão com sinais* $\bar{\rho}(i,\, j)$, com $1 \leq i < j \leq n$, é o rearranjo $(1\ 2 \ldots i-2\ i-1\ \underline{-j\ -(j-1) \ldots -(i+1)\ -i}\ j+1\ j+2 \ldots n-1\ n)$. Isso significa que, quando aplicada a uma string com sinais S, a reversão com sinais inverte o segmento que vai da posição i à posição j de S, invertendo, também, o sinal de cada elemento deste segmento, transformando a string S em $S \cdot \bar{\rho}(i,\, j) = s_1 \ldots s_{i-1}\ \underline{-s_j\ -s_{j-1} \ldots -s_{i+1}\ -s_i}\ s_{j+1} \ldots s_n$.

Uma *reversão de prefixo com sinais*, denotada $\bar{\rho}_p(j)$, com $1 < j \leq n$, é uma reversão com sinais que contém, necessariamente, o primeiro elemento da string com sinais, i.e., equivale a $\bar{\rho}(1, j)$. Já uma *reversão de sufixo com sinais*, denotada $\bar{\rho}_s(i)$, com $1 \leq i < n$, é uma reversão com sinais que contém, necessariamente, o último elemento da string com sinais, i.e., equivale a $\bar{\rho}(i, n)$.

Seja uma string $S = s_1 s_2 \ldots s_n$, com ou sem sinais. A *string reversa* de S, que denotamos por S^R, é a string resultante da aplicação da reversão $\rho(1,\, n)$ em S.

Duas strings $S = s_1 \ldots s_n$ e $T = t_1 \ldots t_n$ são *idênticas*, o que denotamos por $S = T$, se $s_i = t_i$ para cada $i \in \{1, \ldots, n\}$. Dizemos que as strings S e T são *congruentes*, o que denotamos por $S \cong T$, se $S = T$ ou $S = T^R$.

Problemas de Transformação por Operações de Rearranjo são formulados, de maneira mais geral, da seguinte forma: dadas duas strings compatíveis e um conjunto de operações de rearranjo permitidas, encontre o menor custo de uma sequência de operações de rearranjo que transforme uma string na outra. O conjunto das operações permitidas é o que faz a diversidade desta classe de problemas.

Seja $c(\ell) = \ell^\alpha$ o custo de um rearranjo de comprimento ℓ, isto é, o custo de um rearranjo que envolve ℓ elementos. Na abordagem tradicional definimos $\alpha = 0$, de modo que todos os rearranjos têm o mesmo custo unitário e o objetivo se resume a encontrar o menor número de rearranjos necessários para transformar uma string em outra.

Um *modelo de rearranjo* β é o conjunto de operações rearranjo que são permitidas durante o processo de transformação de strings. Escreveremos β como "$(\ |p|ps)(r|\bar{r})$", onde p significa *prefixo*, ps significa *prefixo e sufixo*, r significa *reversão*, $\bar{r}$ significa *reversão com sinais*. Por exemplo, se $\beta = p\bar{r}$, então as operações permitidas são as reversões de prefixo com sinais.

Dadas duas strings compatíveis S e T e um modelo β, a *distância* $d_\beta(S, T)$ entre S e T é o menor custo de uma sequência de rearranjos em β que transforma S em T. Portanto, um *Problema de Transformação de Strings* tem como objetivo encontrar a distância entre S e T

dadas.

Dado um modelo de rearranjo β e um inteiro n, definimos o *diâmetro* $D_\beta(n)$ como o maior valor de $d_\beta(S,T)$ entre todos os pares possíveis de strings compatíveis S e T de tamanho n.

Consideraremos os *acrônimos* para os problemas da forma (TSB) (|P|PS) (R|$\bar{\text{R}}$), onde TSB significa "Transformação de Strings por", P significa "Prefixo", PS significa "Prefixo e Sufixo", R significa "Reversões" e $\bar{\text{R}}$ significa "Reversões com Sinais". Por exemplo, TSBPSR é o acrônimo para o problema de *Transformção de Strings por Reversões de Prefixo e Sufixo*.

Um *duo* é uma string de comprimento igual a 2. Geralmente, esta definição é utilizada para nos referirmos a pares de elementos subsequentes em uma string, sendo os *duos de uma string* S cada uma das substrings de comprimento 2 em S.

O conceito de *breakpoints de reversão*, muito utilizado em permutações, foi adaptado para strings considerando seus duos. Suponha que queremos transformar a string S na string T. Se S contém mais duos xy que T, com $x, y \in \Sigma$, então cada ocorrência de xy que S contém a mais que T deve ser separada em algum ponto da transformação. Cada um destes duos extras, como os xy a mais em S do que em T, são denominados *breakpoints de reversão de S com relação a T*. Uma diferença importante em relação aos breakpoints em permutações é que, em strings, não identificamos as posições dos breakpoints.

Seja $\delta(x) = x$, se $x > 0$, e $\delta(x) = 0$, caso contrário. Para a identificação e contagem dos breakpoints de reversão entre duas strings, consideramos um elemento especial $w < 0$, adicionado no início de ambas as strings, e um elemento especial $z > |\Sigma| - 1$, adicionado no final de ambas as strings. O *número de breakpoints de reversão* entre duas strings compatíveis S e T, denotado por $b_\rho(S,T)$, é dado por

$$b_\rho(S,T) = \sum_{w \le x < y \le z} \delta\left(f(S,xy) + f(S,yx) - f(T,xy) - f(T,yx)\right) + \sum_{x \in \Sigma} \delta\left(f(S,xx) - f(T,xx)\right) .$$

Uma *partição* de uma string $S = s_1 \dots s_n$ é uma sequência $\mathcal{P} = (P_1, P_2, \dots, P_m)$ de strings cuja concatenação é igual a S, isto é, $P_1 P_2 \dots P_m = S$. As strings P_i, com $1 \le i \le m$, são chamadas de *partes* de $\mathcal{P}$, e o número de partes em uma partição é o seu *tamanho*.

Dadas uma partição $\mathcal{P} = (P_1, \dots, P_m)$ de uma string S e uma partição $\mathcal{Q} = (Q_1, \dots, Q_m)$ de uma string T, dizemos que o par $\pi = (\mathcal{P}, \mathcal{Q})$ é uma *partição comum* de S e T, no que diz respeito à relação $\texttt{Rel} \in \{=, \cong\}$, se existe uma permutação σ de $(1, \dots, m)$ tal que $(P_i, Q_{\sigma(i)}) \in \texttt{Rel}$, para cada $i \in \{1, \dots, m\}$.

3 Metodologia

Inicialmente, estudamos os principais artigos científicos sobre os problemas abordados, tanto em permutações quanto em strings, promovendo uma familiaridade com os conceitos necessários para entender os objetos de estudo e o caminho trilhado pelos autores até chegarem aos resultados relevantes já propostos. Os levantamentos sobre estes tópicos encontram-se na Seção 2 e no Apêndice A.

Em seguida, buscamos propor algoritmos simples e incrementá-los conforme o avanço do estudo, isto é, tentamos propor, gradualmente, algoritmos que se aproximassem mais de soluções ótimas. Além disso, desenvolvemos e implementamos um algoritmo de força bruta para analisar soluções ótimas sobre alguns grupos de strings, com o intuito de identificar comportamentos de destaque em strings destes grupos como, por exemplo, grupos muito pequenos que atingem o diâmetro da classe de equivalência, onde todas as strings têm a mesma quantidade de breakpoints. Esses resultados estão documentados em seções posteriores.

Foram realizadas reuniões semanais entre aluno e orientadora, havendo a discussão dos tópicos estudados durante cada semana, bem como a definição dos passos seguintes. Os tópicos relevantes que foram levantados nas reuniões estão documentados no presente relatório.

4 Diferenças entre permutações e strings

Sabendo da diferença entre a definição de breakpoints de reversão para permutações e para strings, uma informação muito importante é evidenciada: em permutações, temos $b_\rho(\pi) = 0$ se e somente se $\pi = \iota$; porém, em strings, $b_\rho(S, T) = 0$ nem sempre implica em $S = T$, apesar de $S = T$ implicar em $b_\rho(S, T) = 0$. Por exemplo, se $S = 12132030$ e $T = 12303120$, então $b_\rho(S, T) = 0$, mas $S \neq T$.

Quando lidamos com permutações, cada elemento ocorre apenas uma vez, de forma que podemos desconsiderar prefixos e/ou sufixos comuns ao ordenar as sequências (lembrando que os problemas de transformação se reduzem a problemas de ordenação).

Por exemplo, se quisermos transformar a permutação $\pi = (5, 3, 6, 1, 4, 2, 7)$ na permutação $\sigma = (5, 4, 6, 3, 1, 2, 7)$, podemos reduzir o problema a ordenar a permutação $\sigma^{-1}\pi = (1, 4, 3, 5, 2, 6, 7)$, i.e., transformar $\sigma^{-1}\pi$ em ι. A solução ótima para esse exemplo tem distância igual a 2, aplicando, por exemplo, as reversões $\rho(4, 5)$ e $\rho(2, 4)$.

É fácil observar que as permutações têm prefixos e sufixos comuns e, eliminando os elementos que os constituem, o problema passa a ser ordenar a permutação $\sigma^{-1}\pi = (4, 3, 5, 2)$. A solução ótima para este novo problema também tem distância igual a 2, aplicando, por exemplo, as reversões $\rho(3, 4)$ e $\rho(1, 3)$.

A ideia de que distância de reversão entre duas permutações é igual à distância de reversão entre suas cópias sem prefixos e/ou sufixos comuns se estende para todas as permutações, sendo justificada pelo fato de não existirem símbolos duplicados [22].

No Problema de Transformação de Strings por Reversões, se quisermos encontrar soluções ótimas, não podemos desconsiderar os prefixos e/ou sufixos comuns. Por exemplo, sejam $S = 12030123$ e $T = 31021203$. A solução ótima desta instância para o problema de transformação por reversões tem distância igual a 3, aplicando, por exemplo, as reversões $\rho(7, 8)$, $\rho(1, 7)$ e $\rho(4, 8)$. Desconsiderando o sufixo comum entre S e T, temos $S' = 1203012$ em $T' = 3102120$. A distância para transformar S' em T' é 4, aplicando, por exemplo, as reversões $\rho(1, 2)$, $\rho(1, 4)$, $\rho(2, 6)$ e $\rho(6, 7)$.

Deste modo, informalmente, podemos considerar que o fato de existirem duplicatas nas strings nos "auxilia" na obtenção de distâncias menores, visto que símbolos iguais podem ser permutados sem prejudicar a solução.

5 Algoritmos desenvolvidos

Considerando, inicialmente, apenas o problema de *Transformação de Strings por Reversões* (TSBR), analisamos como transformar uma string $S = s_1 \dots s_n$ em uma string compatível $T = t_1 \dots t_n$, nos utilizando de métodos simples, que devolvam uma solução viável (neste caso, o comprimento de uma sequência de reversões que transforme S em T), mesmo que ainda longe da solução ótima.

5.1 A ideia inicial (Selection Sort)

O algoritmo mais simples para transformar uma string em outra compatível, por reversões, se assemelha muito ao algoritmo de ordenação Selection Sort.

Dadas duas strings compatíveis S e T, de tamanho n, a ideia consiste em, percorrendo os elementos de S, a partir do índice $i = 1$, comparar cada elemento s_i com t_i e:

(1) caso $s_i = t_i$, incrementar em 1 o índice i;

(2) caso $s_i \neq t_i$, percorrer S até encontrar o menor índice $j > i$ tal que $s_j = t_i$ e aplicar a reversão $\rho(i,\ j)$ à string S.

Contabilizando a quantidade de reversões que são aplicadas a S durante o processo, o algoritmo devolve esta quantidade ao fim de sua execução, determinada pelo momento em que $S = T$. O Algoritmo 1 mostra o pseudocódigo desta ideia.

É fácil perceber que, no pior caso, para duas strings compatíveis de tamanho n, serão necessárias $n - 1$ reversões para realizar a transformação. Como cada reversão leva tempo

$O(n)$ para ser aplicada, o algoritmo tem tempo $O(n^2)$.

Note que este processo é equivalente a dizer que, a cada reversão aplicada, aumentaremos o comprimento do maior prefixo comum entre S e T, denotado por $lcp(S, T)$, em pelo menos uma unidade. O algoritmo se encerrará quando $lcp(S, T) = n$, o que significa que $S = T$.

Algoritmo 1: SELECTION(S, T, n)

```
1  qtdRev ← 0
2  i ← 1
3  enquanto i < n faça
4      se s_i ≠ t_i então
5          j ← i + 1
6          enquanto s_j ≠ t_i faça
7              j ← j + 1
8          S ← S · ρ(i, j)
9          qtdRev ← qtdRev + 1
10     i ← i + 1
11 retorna qtdRev
```

A partir do critério de aumentar o comprimento do maior prefixo comum entre as strings S e T, apenas percorrendo suas posições e comparando se um elemento s_i e outro de t_j são iguais, poderíamos escolher melhor uma reversão a ser aplicada?

Sejam S e T duas strings compatíveis e t_i o elemento de menor posição i em T que não faça parte do maior prefixo comum entre S e T, i.e., $i = lcp(S, T) + 1$. Ao encontrar o elemento s_j, de menor posição j, que seja igual a t_i, com $j > i$, o Algoritmo 1 irá aplicar a reversão $\rho(i, j)$ em S, aumentando o maior prefixo comum entre as strings em pelo menos uma unidade.

Porém, como os elementos podem se repetir em strings, reverter o primeiro elemento encontrado que satisfaça a condição pode não ser a melhor escolha: digamos que, na reversão escolhida, $s_j = t_i$ e $s_{j-1} \neq t_{i+1}$, mas há outro elemento s_k, com $j < k \leq n$, tal que $s_k = t_i$ e $s_{k-1} = t_{i+1}$. Isto significa que pode haver uma reversão $\rho(i, k)$ que, ao ser aplicada a S, aumente mais o valor de $lcp(S, T)$ do que a aplicação de $\rho(i, j)$, onde $j < k \leq n$.

Isso mostra que, ao analisar todas as ocorrências do símbolo de menor posição que não pertence ao maior prefixo comum entre as strings, podemos escolher, a cada iteração, a reversão que mais aumente o valor de $lcp(S, T)$.

O Algoritmo 2 mostra como funciona esta ideia. Como a ideia é efetuar a melhor reversão a cada iteração – que, neste caso, é a que mais aumenta o valor de $lcp(S, T)$ – podemos perceber que o Algoritmo 2 faz sempre escolhas melhores que, ou tão boas quanto, o Algoritmo 1. Porém, com a inclusão do laço da linha 11, que determina a quantidade de elementos em que o maior prefixo comum será incrementado com a reversão que o algoritmo pretende aplicar,

podemos concluir que o Algoritmo 2 tem tempo $O(n^3)$.

Algoritmo 2: SELECTION_MELHORADO(S, T, n)

```
1  qtdRev ← 0
2  i ← 1
3  enquanto i < n faça
4      se s_i ≠ t_i então
5          j ← i + 1
6          melhorJ ← 0
7          maiorSub ← 0
8          enquanto j ≤ n faça
9              se s_j = t_i então
10                 tamSub ← 1
11                 enquanto t_{i+tamSub} = s_{j−tamSub} faça
12                     tamSub ← tamSub + 1
13                 se tamSub > maiorSub então
14                     maiorSub ← tamSub
15                     melhorJ ← j
16             j ← j + 1
17         S ← S · ρ(i, melhorJ)
18         qtdRev ← qtdRev + 1
19     i ← i + 1
20 retorna qtdRev
```

Ainda com base no critério de aumentar o comprimento do maior prefixo comum, poderíamos, a cada iteração, fazer uma escolha de reversão melhor que a do Algoritmo 2?

Os dois algoritmos anteriores analisavam apenas as reversões aplicáveis a S, escolhendo, a cada iteração, aquelas que mais aumentassem o tamanho do maior prefixo comum entre S e T, a aproximando cada vez mais de T.

Acontece que, se utilizarmos a ideia de aproximar T de S, também conseguiríamos realizar a transformação. Suponha que aplicar a sequência de reversões $(\rho_1, \rho_2, \rho_3, \rho_4)$ a uma string qualquer S transforma S em uma string compatível T. Isto é equivalente a dizer que a sequência inversa de reversões, $(\rho_4, \rho_3, \rho_2, \rho_1)$, quando aplicada em T, a transforma em S.

Assim, podemos aplicar reversões tanto em S quanto em T durante a transformação e, ao fim, a sequência de reversões que transforma S em T será formada pela sequência de reversões aplicadas em S, seguida pelo inverso da sequência de reversões aplicadas em T.

A ideia do próximo algoritmo é similar à do Algoritmo 2, de encontrar a reversão que mais aumente o comprimento do maior prefixo comum entre S e T, porém, não mais analisando apenas as reversões aplicáveis a S, mas também as reversões aplicáveis a T. O Algoritmo 8, no Apêndice C, mostra o funcionamento deste processo. Assim como o Algoritmo 2, ele tem tempo $O(n^3)$.

Até agora, os algoritmos estão buscando apenas reversões que aumentem o comprimento do maior prefixo comum, ou seja, que aumentem o valor de $lcp(S, T)$ a cada iteração. Sabendo que os conceitos de sufixo e de maior sufixo comum entre strings são análogos aos conceitos de prefixos, mas na outra extremidade das strings, conseguiríamos analisar mais reversões, candidatas a melhor reversão em uma determinada iteração do algoritmo?

O próximo algoritmo também utiliza a ideia de aumentar $lcp(S, T)$ a cada iteração, percorrendo ambas as strings, sendo muito parecido com o Algoritmo 8, com apenas uma modificação. Além de percorrer as strings apenas a partir da primeira posição, buscando a reversão que mais aumente o maior prefixo comum entre elas, podemos, também, percorrê-las a partir da última posição, buscando a reversão que mais aumente o maior sufixo comum entre elas, i.e., que mais aumente o valor de $lcs(S, T)$.

Desta forma, a cada iteração, encontraremos (entre todas as reversões aplicáveis a S e todas aplicáveis T) a reversão que mais aumente o prefixo comum e a que mais aumente o sufixo comum e, entre as duas reversões encontradas, aplicaremos a que mais aumentar o valor de $lcp(S, T) + lcs(S, T)$.

As quatro possíveis candidatas a melhor reversão, a cada iteração, serão: (a) ρ que mais aumente $lcp(S \cdot \rho, T)$; (b) ρ que mais aumente $lcp(S, T \cdot \rho)$; (c) ρ que mais aumente $lcs(S \cdot \rho, T)$; (d) ρ que mais aumente $lcs(S, T \cdot \rho)$. Entre as quatro, a que mais aumentar o valor $lcp(S, T) + lcs(S, T)$ será aplicada pelo algoritmo na respectiva string.

O Algoritmo 9, no Apêndice C, mostra o funcionamento da ideia descrita. Neste algoritmo, cada uma das linhas 6 a 9 representa um bloco equivalente ao bloco das linhas 10 a 17 (ou 18 a 25) do Algoritmo 8 e, portanto, cada uma possui o mesmo tempo de execução que o bloco. Se tivéssemos quatro blocos idênticos no Algoritmo 8, em vez dos dois blocos citados, seu tempo continuaria sendo $O(n^3)$. Assim, podemos concluir que o Algoritmo 9 também tem tempo $O(n^3)$.

Para concluir a ideia de aumentar, a cada iteração, o valor de $lcp(S, T)$ e/ou de $lcs(S, T)$, podemos propor um algoritmo que una cada uma das variações vistas até aqui e retorne a solução de menor valor encontrada entre todas estas variações. Os algoritmos propostos até este ponto se mostram, na ordem em que propusemos, progressivamente melhores, em média. Isto se deve ao fato das reversões candidatas em cada iteração do Algoritmo 2 também serem candidatas no Algoritmo 8, por exemplo, mas o contrário não ser válido. Porém, mesmo com uma melhora progressiva na média dos resultados, um algoritmo com menos reversões candidatas, a cada iteração, não tem resultado necessariamente pior que um algoritmo com mais candidatas em todas as instâncias. Assim, pelo fato dos algoritmos terem tempo de execução rápido, decidimos criar um novo algoritmo que une nove variações em relação ao valor que buscamos aumentar ($lcp(S, T)$, $lcs(S, T)$ ou $lcp(S, T) + lcs(S, T)$) e a qual string

aplicar reversões (S, T ou ambas). Ele aplica cada uma das variações e devolve o resultado da melhor delas. Especificamente, cada uma das variações busca apenas reversões que sejam: (1) aplicáveis a S e aumentem $lcp(S,T)$; (2) aplicáveis a T e aumentem $lcp(S,T)$; (3) aplicáveis a S e aumentem $lcs(S,T)$; (4) aplicáveis a T e aumentem $lcs(S,T)$; (5) aplicáveis a S e aumentem $lcp(S,T) + lcs(S,T)$; (6) aplicáveis a T e aumentem $lcp(S,T) + lcs(S,T)$; (7) aplicáveis a S ou a T e aumentem $lcp(S,T)$; (8) aplicáveis a S ou a T e aumentem $lcs(S,T)$; (9) aplicáveis a S ou a T e aumentem $lcp(S,T) + lcs(S,T)$. Como o algoritmo mais custoso incluído nesta solução é o Algoritmo 9, que tem tempo de execução $O(n^3)$, o novo algoritmo também executará em tempo $O(n^3)$.

5.2 Eliminando breakpoints

Como já vimos, *breakpoints de reversão* entre duas strings compatíveis S e T, nesta ordem, são duos (substrings de tamanho 2, ou pares formados por elementos subsequentes) que possuem mais ocorrências em S do que em T.

Quando queremos transformar strings por reversões, cada reversão aplicada em uma string pode remover, no máximo, 2 breakpoints [8]. Sabendo que, se $S = T$, então $b_\rho(S,T) = 0$, uma das estratégias que podemos utilizar em um algoritmo de transformação é a eliminação exaustiva de breakpoints.

Porém, um empecilho a esta estratégia é o fato de $b_\rho(S,T) = 0$ não implicar em $S = T$. Assim, se sempre eliminarmos breakpoints, até que não existam mais breakpoints entre as strings, não necessariamente teremos transformado uma string na outra. Outra observação é que só teremos $b_\rho(S,T) = 0$ se as extremidades das strings forem iguais, isto é, $s_1 \dots s_i = t_1 \dots t_i$, para algum $1 \leq i \leq n$, e $s_j \dots s_n = t_j \dots t_n$, para algum $1 \leq j \leq n$. Isto significa que quando $b_\rho(S,T) = 0$, obrigatorimente $lcp(S,T) + lcs(S,T) > 0$.

Para os algortimos propostos a seguir, o seguinte lema será importante, pois define a quantidade de breakpoints entre as strings resultantes da remoção dos prefixos e sufixos comuns a duas strings que, originalmente, não possuem breakpoints entre si.

Lema 1. *Sejam S e T duas strings compatíveis, com $b_\rho(S,T) = 0$. Se S' e T' são as strings resultantes da eliminação dos prefixos e sufixos comuns a S e T, respectivamente, então temos que $2 \leq b_\rho(S',T') \leq 4$.*

Demonstração. Sejam S e T duas strings compatíveis de comprimento n, descritas sobre um alfabeto Σ, com $b_\rho(S,T) = 0$. Assim como fazemos ao contar o número de breakpoints entre strings, adicionaremos o elemento $w < 0$ no início de ambas as strings e o elemento $z > |\Sigma| - 1$ no final de ambas as strings.

Denotaremos por A e B, respectivamente, o maior prefixo e o maior sufixo comuns entre as strings S e T. Chamaremos de a o último elemento da substring A e de b o primeiro elemento da substring B. O elemento imediatamente seguinte a a em S será chamado de c, enquanto o elemento imediatamente seguinte a a em T será chamado de d. O elemento imediatamente anterior a b em S será chamado de e, enquanto o elemento imediatamente anterior a b em T será chamado de f. Assim, as strings S e T podem ser representadas da seguinte forma:

$$S = A\ c\ \ldots e\ B$$
$$T = A\ d\ \ldots f\ B$$

É importante ressaltar que, apesar de termos nomeado os elementos com símbolos diferentes, não teremos, necessariamente, $c \neq d \neq e \neq f$.

Eliminando A e B das string S e T, criaremos, respectivamente, duas novas strings $S' = c\ \ldots e$ e $T' = d\ \ldots f$.

Como os segmentos A e B são exatamente iguais em S e em T, cada duo presente nos segmentos $w \ldots a$ e $b \ldots z$ contribui igualmente para a contagem de breakpoints em S e T, fazendo com que suas eliminações não afetem na contagem de breakpoints entre S' e T'. Porém, os duos de suas extremidades, que serão separados com a eliminação destes segmentos, afetam na contagem de breakpoints e necessitam de uma atenção especial: ac e eb em S, e ad e fb em T.

Ressaltamos que $c \neq d$ e $e \neq f$, pois, caso contrário, os segmentos A e B não seriam, respectivamente, os maiores prefixos e sufixos comuns entre S e T.

Ao eliminarmos o prefixo A, sabemos que um dos breakpoints de S' em relação a T' será o duo wc, e que um dos breakpoints de T' em relação a S' será o duo wd. Assim, a eliminação do prefixo A criou um breakpoint entre S' e T'.

Analogamente, temos esta relação nos sufixos comuns: ao eliminarmos o sufixo B, sabemos que um dos breakpoints de S' em relação a T' será o duo ez, e que um dos breakpoints de T' em relação a S' será o duo fz. Assim, a eliminação do sufixo B criou mais um breakpoint entre S' e T'.

Neste momento, sabemos que o prefixo e o sufixo eliminados geraram dois breakpoints entre S' e T'. Estes dois breakpoints sempre existirão, pois se devem ao fato de ambas as extremidades de S' serem diferentes das respectivas extremidades em T'.

Porém, outros dois breakpoints podem ser criados com a eliminação de prefixo e sufixo comuns. Para demonstrar este fato, vamos considerar três tipos de relações possíveis entre os duos ac, ad, eb e fb.

Caso I Quando $ac \ncong fb$ e $ad \ncong eb$ (i.e., quando não há pares de duos congruentes entre os duos cortados).

Neste caso, a eliminação do prefixo A de S (resp. T) corta uma ocorrência do duo ac em S (resp. ad em T), que passa a ocorrer uma vez menos em S' (resp. T') do que em T' (resp. S'), criando um breakpoint de T' (resp. S') em relação a S' (resp. T'), nesta ordem. Portanto, este corte cria um breakpoint adicional entre S' e T', totalizando, agora, três breakpoints.

De forma análoga, podemos fazer esta análise no sufixo comum B: a eliminação do sufixo B de S (resp. T) corta uma ocorrência do duo eb em S (resp. fb em T), que passa a ocorrer uma vez menos em S' (resp. T') do que em T' (resp. S'), criando um breakpoint de T' (resp. S') em relação a S' (resp. T'), nesta ordem. Deste modo, este corte cria um breakpoint adicional entre S' e T', totalizando, agora, quatro breakpoints.

Concluindo, quando não temos duos congruentes entre os quatro duos cortados pela eliminação de prefixos e sufixos, temos $b_\rho(S', T') = 4$.

Caso II Quando $ac \cong fb$ e $ad \ncong eb$, ou quando $ac \ncong fb$ e $ad \cong eb$ (i.e., quando há apenas um par de duos congruentes entre os quatro duos cortados).

Neste caso, vamos considerar que $ac \cong fb$ e $ad \ncong eb$, mas o mesmo raciocínio se aplica ao caso onde $ac \ncong fb$ e $ad \cong eb$. Quando dois dos duos cortados são congruentes, a eliminação de um duo em uma string é compensada pela eliminação de um duo congruente na outra string, e isto não cria breakpoints.

A eliminação do prefixo A de S corta uma ocorrência do duo ac em S, que passa a ocorrer uma vez menos em S' do que em T', criando um breakpoint de T' em relação a S', nesta ordem. Porém, como sabemos que $ac \cong fb$, a eliminação do sufixo B de T corta uma ocorrência do duo fb em T, que passa a ocorrer uma vez menos em T' do que em S', compensando o breakpoint criado ao cortar o duo ac em S, já que estes duos são congruentes. Dessa forma, estes dois cortes não criam um breakpoint (ou podemos considerar que um breakpoint é criado, mas eliminado em seguida).

Sabendo que $ad \ncong eb$, as eliminações de A e B cortarão, respectivamente, uma ocorrência de ad em T e uma ocorrência de eb em S, como visto no *Caso I*. Assim, este corte cria um breakpoint adicional entre S' e T', totalizando, agora, três breakpoints.

Concluindo, quando há apenas um par de duos congruentes entre os quatro duos que serão cortados pela eliminação de prefixos e sufixos, temos $b_\rho(S', T') = 3$.

Caso III Quando $ac \cong fb$ e $ad \cong eb$ (i.e., quando há dois pares de duos congruentes entre os duos cortados).

Sabendo que $ac \cong fb$ e $ad \cong eb$, podemos supor que não serão criados breakpoints neste caso, como já visto no *Caso II*, pois, aqui, temos dois pares de duos congruentes.

Concluindo, quando há dois pares de duos congruentes formando os quatro duos que serão cortados pela eliminação de prefixos e sufixos, temos $b_\rho(S', T') = 2$.

Portanto, considerando os três casos possíveis, sabemos que a eliminação de prefixos e sufixos comuns de S e T criam, respectivamente, S' e T', com $2 \leq b_\rho(S', T') \leq 4$.

□

Sabendo do resultado do Lema 1, poderíamos seguir uma das estratégias abaixo:

(1) eliminar breakpoints das strings originais, até que não haja mais nenhum, e só depois remover prefixo e/ou sufixo comuns entre as strings, repetindo o processo até que $S = T$;

(2) eliminar o máximo de breakpoints a cada iteração e, a cada reversão aplicada, se possível, eliminar prefixo e/ou sufixo comuns entre as duas strings, repetindo o processo até que $S = T$.

De antemão, sabemos que desconsiderar prefixos e sufixos comuns pode nos distanciar de soluções ótimas, como exemplificado na Seção 4. Porém, usar esta estratégia é uma alternativa para os algoritmos da seção anterior, que apenas tentam aumentar o comprimento dos prefixos e sufixos comuns.

Temos, então, que identificar a forma das reversões que eliminam breakpoints. A relação entre a quantidade removida de breakpoints e uma reversão $\rho(i, j)$, aplicada a S, é a seguinte:

- $\rho(i, j)$ remove 2 breakpoints caso $s_{i-1}s_i$ e $s_j s_{j+1}$ sejam breakpoints de S em relação a T e $s_{i-1}s_j$ e $s_i s_{j+1}$ sejam breakpoints de T em relação a S;
- $\rho(i, j)$ remove 1 breakpoint caso $s_{i-1}s_i$ e $s_j s_{j+1}$ sejam breakpoints de S em relação a T e $s_{i-1}s_j$ ou $s_i s_{j+1}$ (estritamente um deles) seja breakpoint de T em relação a S;
- $\rho(i, j)$ também remove 1 breakpoint caso $s_{i-1}s_j$ e $s_i s_{j+1}$ sejam breakpoints de T em relação a S e $s_{i-1}s_i$ ou $s_j s_{j+1}$ (estritamente um deles) seja breakpoint de S em relação a T.

Na estratégia (1), uma das opções ao zerar a quantidade de breakpoints é substituir o maior prefixo comum por w e o maior sufixo comum por z. Como visto no Lema 1, sabemos que a eliminação de prefixos e sufixos criará duas novas strings, de comprimento menor que as strings originais, mas com 2, 3 ou 4 breakpoints.

Dadas duas strings compatíveis S e T, se conseguirmos aplicar reversões que sempre diminuam ou mantenham a quantidade de breakpoints, quando esta quantidade for zerada, eliminaremos prefixos e sufixos comuns. As novas strings, obtidas com a eliminação das extremidades comuns, terão, no mínimo, 2 breakpoints. Repetindo o processo de aplicar reversões que diminuam ou mantenham a quantidade de breakpoints, ao zerar a breakpoints

entre este novo par de strings, reduziremos ainda mais o comprimento de ambas, mesmo que criando, a cada eliminação, ao menos 2 breakpoints.

A cada nova eliminação, repetimos os passos com o par de strings obtido do processo anterior. Em dado momento, ou o algoritmo vai terminar sua execução por termos aplicado uma reversão que fez com $S = T$, ou vai terminar pelo fato das duas strings terem sido tão reduzidas por eliminações de prefixos e sufixos comuns, que se tornaram dois duos não idênticos, mas congruentes. Neste segundo caso, a única reversão aplicável será a que fará com que ambas sejam idênticas. O Algoritmo 3 mostra o funcionamento desta estratégia.

Algoritmo 3: ELIMINA_BREAKPOINTS_1(S, T, n)

```
1  qtdRev ← 0
2  enquanto S ≠ T faça
3      enquanto b_ρ(S,T) ≠ 0 faça
4          encontra a reversão ρ(i,j) que não aumente ou que elimine a maior quantidade de
             breakpoints
5          S ← S · ρ(i,j)
6          qtdRev ← qtdRev + 1
7      se S ≠ T então
8          S,T ← S,T sem o prefixo comum
9          S,T ← S,T sem o sufixo comum
10 retorna qtdRev
```

Na estratégia (2), percebemos que, enquanto $S \neq T$, a quantidade de breakpoints nunca será zero, pois eliminar prefixo e/ou sufixo comuns a cada reversão realizada garante que ambas as extremidades de S e T sejam diferentes, sempre. Como sabemos que extremidades diferentes implicam em um número de breakpoints diferente de zero, podemos afirmar que só teremos a quantidade de breakpoints zerada quando $S = T$. Com isso, fica mais claro o critério de parada do algoritmo, e a confirmação de que sua execução terminará.

Porém, assim como na estratégia (1), consideramos que conseguiremos aplicar reversões que sempre diminuam ou mantenham a quantidade de breakpoints. O Algoritmo 4 mostra o funcionamento desta estratégia.

Ambos os algoritmos partem da premissa de que conseguimos zerar o número de breakpoints sem nunca o aumentarmos, i.e., encontraremos uma sequência de reversões que, a cada reversão aplicada, ou manterá a quantidade de breakpoints, ou eliminará pelo menos um deles, até que não exista mais nenhum. Além disso, quando não há reversões que eliminem breakpoints, é necessário definir um segundo critério de escolha de reversão.

Hannenhalli e Pevzner [17] provaram o Teorema 2 para o problema de reversão em permutações sem sinais, cuja validade em strings será uma de nossas investigações futuras.

Algoritmo 4: ELIMINA_BREAKPOINTS_2(S, T, n)

```
qtdRev ← 0
enquanto b_ρ(S,T) ≠ 0 faça
    S,T ← S,T sem o prefixo comum
    S,T ← S,T sem o sufixo comum
    encontra a reversão ρ(i,j) que não aumente ou que elimine a maior quantidade de
     breakpoints
    S ← S · ρ(i,j)
    qtdRev ← qtdRev + 1
retorna qtdRev
```

Teorema 2. [17] *Para toda permutação π no grupo simétrico S_n, existe uma sequência ótima de reversões que ordena π sem nunca aumentar seu número de breakpoints fortes.*

Infelizmente, não há resultado análogo ao Teorema 2 para strings, pois há casos em que não conseguimos encontrar uma sequência ótima de reversões sem que aumentemos o número de breakpoints entre as strings originais. Por exemplo, sejam as strings $S = 01021323$ e $T = 01321023$, sem breakpoints de reversão entre si, com distância de reversão $d(S, T) = 2$, e para as quais uma possível sequência ótima de reversões aplicadas a S é $\rho_1(3, 6), \rho_2(4, 5)$. Assim, teríamos $S = 01[0213]23 \rightarrow 013[12]023 \rightarrow 01321023 = T$. Note que a primeira reversão criou dois breakpoints entre as strings, que foram removidos pela segunda reversão, e esta, por sua vez, também tornou ambas as strings idênticas. Se tentássemos transformar S em T otimamente, i.e., com apenas duas reversões, e mantendo o par sem breakpoints entre si, não conseguiríamos: as únicas reversões que podem ser aplicadas a S e que não criam breakpoints são $\rho_1(3, 4)$ e $\rho_2(5, 6)$, que resultariam em $S \cdot \rho_1 = 01201323$ e $S \cdot \rho_2 = 01023123$. Porém, podemos notar que nenhuma das duas strings resultantes estão a apenas uma reversão de distância de T.

Sabendo da inexistência de um resultado análogo ao Teorema 2 para strings, duas informações importantes podem ser notadas: (i) evitar a criação de breakpoints pode não ser uma boa estratégia para nos aproximar de soluções ótimas, visto que há casos onde só conseguimos resolver otimamente aumentando, em algum momento, a quantidade de breakpoints; (ii) nem sempre será possível aplicar uma reversão que elimine breakpoints, mesmo havendo breakpoints entre o par de strings, o que faz com que necessitemos de um segundo critério para escolher a reversão quando as opções apenas mantêm ou aumentam a quantidade de breakpoints.

Por estarem descritos de uma forma simples, os Algoritmos 3 e 4 não têm garantia quanto ao término de sua execução, pois quando não há reversões que eliminem breakpoints, não há um segundo critério para escolher entre as reversões que mantêm o número de breakpoints.

Além disso, ao implementar o algoritmo, percebemos que, sem este segundo critério, nunca aumentar a quantidade de breakpoints pode nos levar a laço infinito, pois uma determinada reversão pode ser considerada a melhor em uma determinada iteração e, também, na iteração seguinte, gerando um ciclo. Para evitar estes ciclos, um dos critérios que podemos utilizar é o visto no Algoritmo 9, que escolhe, a cada iteração, a reversão aplicável a S ou a T que mais aumente o valor de $lcp(S,T) + lcs(S,T)$.

Após a definição deste critério, temos certeza que o algoritmo terminará sua execução, pois: havendo reversões que eliminem breakpoints, e utilizando a estratégia de remover os prefixos e sufixos comuns, estaremos cada vez mais perto de zerar os breakpoints das strings restante, o que vimos implicar em $S = T$; não havendo reversões que eliminem breakpoints, o algoritmo se comportará como uma das iterações do Algoritmo 9, o que também implicará em uma redução no comprimento das strings, por meio da remoção de prefixos e sufixos comuns, pois este novo critério tem justamente a finalidade de aumentar estes extremos comuns. O Algoritmo 5 mostra funcionamento desta ideia.

Algoritmo 5: ELIMINA_BREAKPOINTS_CORRIGIDO(S, T, n)

```
1  qtdRev ← 0
2  enquanto b_ρ(S,T) ≠ 0 faça
3      S,T ← S,T sem o prefixo comum
4      S,T ← S,T sem o sufixo comum
5      busque em S uma reversão ρ_br(i,j) que elimine a maior quantidade de breakpoints
6      se existir ρ_br(i,j) que elimina breakpoints então
7          S ← S · ρ_br(i,j)
8      senão
9          busque em S e em T a reversão ρ_ext(i,j) que mais aumente o valor de
             lcp(S,T) + lcs(S,T)
10         se ρ_ext(i,j) é aplicável em S então
11             S ← S · ρ_ext(i,j)
12         se ρ_ext(i,j) é aplicável em T então
13             T ← T · ρ_ext(i,j)
14     qtdRev ← qtdRev + 1
15 retorna qtdRev
```

Note que esta nova estratégia foi incluída apenas na estrutura do Algoritmo 4, que, no início de cada iteração, elimina os prefixos e sufixos comuns. Isso é justificado pelo fato de que, caso tentássemos implementar esta mesma ideia no Algoritmo 3, que não elimina as extremidades a cada reversão aplicada, haveria a possibilidade de não encontrarmos uma reversão que elimina breakpoints, o que faria com que o segundo critério (que busca aumentar o valor de $lcp(S,T) + lcs(S,T)$) escolhesse uma reversão que aumentaria o número de breakpoints. Porém, como a última reversão aplicada aumentou o número de breakpoints,

implicando na não remoção das extremidades comuns, esta mesma reversão seria escolhida na iteração seguinte, já que o primeiro critério é a escolha por uma reversão que elimine breakpoints, gerando um ciclo que impediria o término da execução.

5.3 Heurísticas e Meta-heurísticas

Um algoritmo é considerado um *método heurístico*, ou uma *heurística*, quando não há conhecimentos resultantes de uma análise formal de seu comportamento, mas, mesmo sem oferecer garantias (como um fator de aproximação, por exemplo), objetiva resolver problemas complexos consumindo uma quantidade não muito grande de recursos – tempo, principalmente – e retornando soluções de boa qualidade. As heurísticas, normalmente, partem de uma boa intuição, baseada em alguma propriedade específica observada no problema tratado, explorando esta propriedade a fim de aumentar a qualidade das soluções.

Uma *meta-heurística* é um conjunto de mecanismos que definem, de forma inteligente, métodos heurísticos aplicáveis a um extenso conjunto de diferentes problemas. Assim, trata-se de estrutura algorítmica geral que pode ser aplicada a diferentes problemas de otimização e, com poucas modificações, podemos adaptá-la a um problema específico [24].

5.4 Mapeamentos de Strings em Permutações

Ao lidarmos com strings em que há repetição de pelo menos um de seus símbolos, se renomearmos cada uma das ocorrências de cada símbolo repetido, dando um rótulo diferente a cada uma delas, transformaremos a string original em uma permutação. Podemos definir este processo como *mapeamento* de uma string em uma permutação.

Seja uma string $S = s_1\, s_2 \ldots s_n$, de comprimento n, descrita sobre um alfabeto Σ, com $|\Sigma| = k$, e pertencente a uma classe de equivalência $\mathcal{L} = (a_0, a_1, \ldots, a_{k-1})$, onde a_i representa o número de ocorrências do símbolo $i \in \Sigma$ em S. Um *mapeamento* m de S pode ser representado por um conjunto de k sequências m_i, com $0 \leq i < k$, sendo cada sequência m_i uma permutação do conjunto $[a_i] = \{1, 2, \ldots, a_i\}$. O elemento que ocupa a posição j de uma sequência m_i, o qual chamaremos de $m_{(i,j)}$, representa o índice que irá rotular a j-ésima ocorrência do símbolo i em S. A permutação resultante da aplicação de novos rótulos a S, a partir de um mapeamento m, será denotada por S^m.

Exemplo 1. *Seja a string* $S = 12323013$*, descrita sobre o alfabeto* $\Sigma = \{0, 1, 2, 3\}$*, com* $|\Sigma| = k = 4$*, pertencente à classe de equivalência* $\mathcal{L} = (1, 2, 2, 3)$*. Um mapeamento* m *possível para* S *pode ser:* $m_0 = (1)$, $m_1 = (2, 1)$, $m_2 = (1, 2)$, $m_3 = (2, 3, 1)$*. A permutação resultante da aplicação dos rótulos de* m *em* S *será* $S^m = (1_2, 2_1, 3_2, 2_2, 3_3, 0_1, 1_1, 3_1)$.

Um *mapeamento trivial* m de S é um mapeamento em que cada elemento $m_{(i,j)} = j$, o que é equivalente a dizer que cada sequência m_i está ordenada de forma crescente.

Exemplo 2. *Para a string* $S = 12323013$*, do exemplo anterior, o mapeamento trivial* z *seria* $z_0 = (1)$, $z_1 = (1, 2)$, $z_2 = (1, 2)$, $z_3 = (1, 2, 3)$. *E a permutação resultante do mapeamento trivial seria* $S^z = (1_1, 2_1, 3_1, 2_2, 3_2, 0_1, 1_2, 3_3)$.

Dadas duas strings compatíveis S e T, podemos gerar um mapeamento p para S e um mapeamento q para T e, rotulando cada uma das strings a partir de seus respectivos mapeamentos, obteremos as permutações S^p e T^q. Se a distância de reversão entre as strings S e T é $d(S, T)$, sabemos que $d(S, T) \leq d(S^p, T^q)$, para quaisquer mapeamentos p e q, e que existe pelo menos um par de mapeamentos p e q tais que $d(S, T) = d(S^p, T^q)$.

Note que uma sequência de reversões que transforme S em T também transformará S^p em T^q. Porém, em permutações, o problema de Transformação por Reversões é equivalente ao problema de Ordenação por Reversões, bastando renomear os elementos de ambas as permutações, de modo que uma delas seja a permutação identidade $\iota = (1, 2, 3, \ldots, n)$.

Assim, utilizando a estratégia de mapeamento das strings em permutações, podemos reduzir o problema de Transformação de Strings por Reversões ao problema de Ordenação de Permutações por Reversões e, consequentemente, obter sequências de reversões e distâncias aproximadas para o primeiro problema, a partir de algoritmos para o segundo.

5.5 Algoritmo de Aproximação para o Problema de Ordenação de Permutações por Reversões

Para os métodos heurísticos aqui propostos, mapearemos os pares de strings em pares de permutações e utilizaremos um algoritmo de aproximação clássico para o problema de Ordenação de Permutações por Reversões, proposto por Kececioglu e Sankoff, em 1995 [20].

Dada a versão estendida de uma permutação $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, definimos uma *strip* de π como um intervalo $[i, j]$, tal que $(i-1,\ i)$ e $(j,\ j+1)$ são breakpoints de π e não há breakpoint no intervalo. Ou seja, uma strip é uma substring maximal crescente ou descrescente. Caso a strip contenha apenas um elemento, a consideramos decrescente.

Com estas definições, podemos entender o funcionamento do algoritmo de Kececioglu e Sankoff, denominado KS95 e descrito no Algoritmo 6.

O algoritmo KS95 é uma 2-aproximação para o problema de Ordenação de Permutações por Reversões, executando em tempo $O(n^2)$, sendo n o comprimento da string de entrada. Este algoritmo não possui o melhor fator de aproximação conhecido para este problema, sendo o melhor resultado uma 1.375-aproximação, proposta por Berman *et al.* [5], mas sem implementação possível.

Algoritmo 6: KS95

Entrada: uma permutação π

$i = 0$

enquanto π *contiver breakpoints* **faça**

  $i = i + 1$

  Seja ρ_i uma reversão que remove o maior número possível de breakpoints de π, favorecendo reversões que deixem uma strip decrescente em π, em caso de empate no número de breakpoints removidos.

  $\pi = \pi \cdot \rho_i$

Saída: $i, (\rho_1, \rho_2, \ldots, \rho_i)$

Tendo o algoritmo KS95 um tempo de execução rápido e uma implementação simples, este foi escolhido para ser utilizado junto às heurísticas aqui propostas, que terão como foco a busca por mapeamentos de strings em permutações para as quais os valores das soluções calculadas por KS95 sejam os menores possíveis.

5.6 Mapeamentos Aleatórios

A primeira heurística proposta consiste em, dadas duas strings compatíveis S e T e um inteiro m, gerar m mapeamentos aleatórios para S – consequentemente, gerar m permutações a partir de S – e um mapeamento trivial t para T e calcular, utilizando KS95, a quantidade de reversões que transforme cada uma das permutações geradas a partir de S na permutação T^t. Após gerar as m soluções por KS95, o algoritmo retorna a de menor valor.

5.7 Algoritmo Genético de Chaves Aleatórias Viciadas (BRKGA)

O outro método heurístico aqui proposto consiste na implementação da meta-heurística denominada Algoritmo Genético de Chaves Aleatórias Viciadas (BRKGA) para o problema de Transformação de Strings por Reversões. Porém, antes de chegarmos de fato ao BRKGA, é necessário entender o conceito de Algoritmos Genéticos e Algoritmos Genéticos de Chaves Aleatórias. O restante desta subseção é um resumo do que foi proposto por Gonçalves e Resende, em 2010 [16].

Os *Algoritmos Genéticos*, ou *GAs* (do inglês *Genetic Algorithms*), são algoritmos que aplicam o conceito de *sobrevivência do mais apto* para encontrar soluções, ótimas ou aproximadas, para problemas de Otimização Combinatória [15].

O funcionamento de um GA parte da analogia entre uma solução viável para o problema tratado e um *indivíduo* em uma *população*. Cada *indivíduo* tem um cromossomo correspondente que codifica a solução. Um cromossomo possui associado a ele um nível de *aptidão*, correlacionado ao valor da função objetivo correspondente da solução que o cromossomo co-

difica. Os GAs desenvolvem um conjunto de indivíduos que compõem uma população ao longo de várias *gerações*. A cada geração, uma nova população é criada, combinando elementos da população atual, de forma aleatória, porém dando preferência aos mais aptos, para produzir descendentes, que compõem a geração seguinte. É neste ponto que atua o conceito de sobrevivência do mais apto. A *mutação* aleatória também ocorre em algoritmos genéticos como um meio de escapar dos mínimos locais.

Nos *Algoritmos Genéticos com Chaves Aleatórias*, ou *RKGAs* (do inglês *Random-key Genetic Algorithms*), introduzidos por Bean, em 1994 [3], os cromossomos são representados como uma sequência de números reais gerados aleatoriamente no intervalo [0,1]. Um algoritmo determinístico, chamado *decodificador*, toma como entrada qualquer cromossomo e associa a ele uma solução do problema de otimização combinatória para a qual um valor objetivo ou de aptidão pode ser calculado.

Um RKGA desenvolve uma população de vetores de chaves aleatórias ao longo de várias iterações, chamadas *gerações*. A população inicial é composta de p vetores de chaves aleatórias. O valor de cada posição de cada vetor é gerado independentemente e de forma aleatória no intervalo real [0,1]. Após a aptidão de cada indivíduo ser calculada pelo decodificador, a população é dividida em dois grupos de indivíduos: um pequeno conjunto com p_e indivíduos de *elite* (com os melhores valores de aptidão) e o conjunto restante com $p - p_e$ indivíduos *não-elite*, onde $p_e < p - p_e$. Para evoluir a população, uma nova geração de indivíduos deve ser produzida. Um RKGA utiliza uma *estratégia elitista*, uma vez que todos os indivíduos de elite da geração k são copiados, sem alterações, para a geração $k + 1$. Esta estratégia mantém as boas soluções encontradas durante as iterações do algoritmo, resultando em uma heurística que nunca piora a solução parcial a cada iteração.

A *mutação* tem papel fundamental nos algoritmos genéticos, sendo usada para permitir que escapemos dos mínimos locais. Os RKGAs implementam mutações introduzindo os chamados *mutantes* na população. Um mutante é simplesmente um vetor de chaves aleatórias gerado da mesma maneira que um elemento da população inicial é gerado. A cada geração, um pequeno número p_m de mutantes é introduzido na população. Com os p_e indivíduos de elite e os p_m mutantes representados na população $k+1$, é necessário produzir mais $p-p_e-p_m$ indivíduos para completar o número p de indivíduos que compõem a população da geração $k + 1$. Isso é feito produzindo $p - p_e - p_m$ descendentes através do processo de *cruzamento*.

A Figura 1 ilustra o funcionamento da evolução. À esquerda da figura está a população da geração atual k. Depois que todos os indivíduos são classificados por seus valores de aptidão, os mais aptos são colocados na partição denominada ELITE, e os indivíduos restantes são colocados na partição identificada como NÃO ELITE. Os vetores de chaves aleatórias de elite são copiados sem alterações para a partição ELITE na próxima população (no lado

direito da figura). Vários indivíduos mutantes são gerados aleatoriamente e colocados na nova população na partição denominada MUT. O restante da população da geração seguinte é completado através do processo chamado *crossover* (*cruzamento*). Para gerar um elemento descendente por meio de *crossover* no RKGA, como proposto originalmente, são selecionados dois elementos pais, aleatoriamente, de toda a população. Como a diferença entre os RKGAs e o próximo tipo de algoritmos genéticos que trataremos aqui está justamente no processo de *crossover*, a seguir, descreveremos este processo junto a este outro tipo de GA.

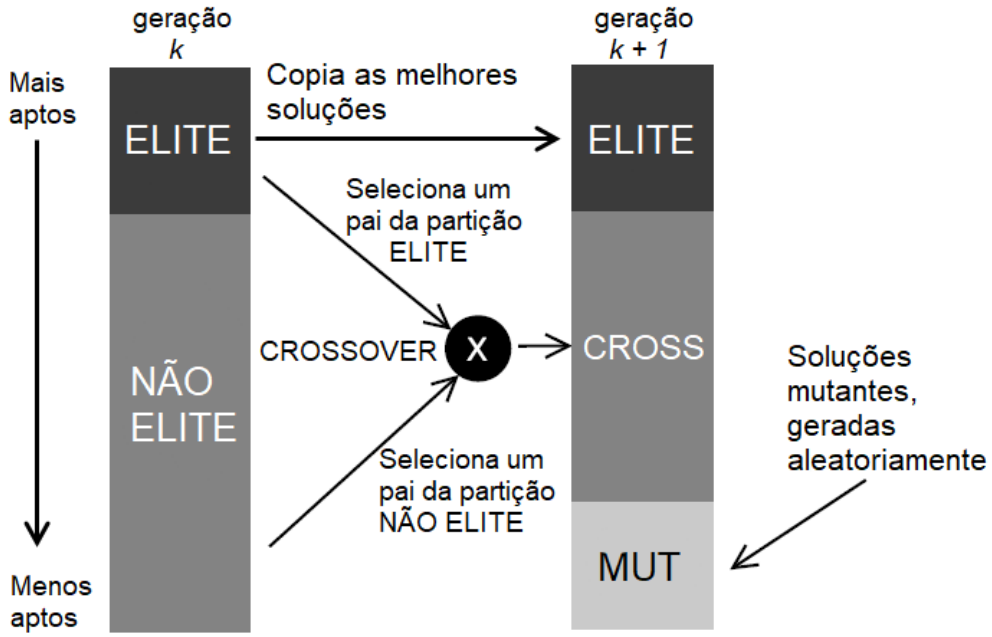

Figura 1: Funcionamento da transição entre gerações no BRKGA, traduzido e adaptado de Gonçalves e Resende [16].

Um *Algoritmo Genético de Chaves Aleatórias Viciadas*, ou *BRKGA* (do inglês *Biased Random-key Genetic Algorithm*), difere de um RKGA na maneira como os pais são selecionados para o *crossover* [16]. Em um BRKGA, cada elemento descendente da geração seguinte é gerado combinando um elemento pai selecionado aleatoriamente da partição rotulada ELITE na população atual e um elemento pai da partição rotulada como NÃO ELITE, também selecionado aleatoriamente. Em alguns casos, o segundo pai é selecionado de toda a população. É permitida a repetição na seleção de um parceiro e, portanto, um indivíduo pode produzir mais de um filho. Como $p_e < p - p_e$, a probabilidade $\frac{1}{p_e}$ de um determinado indivíduo de elite ser selecionado para acasalar é maior do que a probabilidade $\frac{1}{p-p_e}$ de um determinado indivíduo não elite ser selecionado e, portanto, o indivíduo de elite tem maior probabilidade de transmitir suas características para as gerações futuras do que um não elite. Também contribuem para esse fim o *cruzamento uniforme parametrizado*, o mecanismo usado para implementar o cruzamento nos BRKGAs, e o fato de que um dos pais é

sempre selecionado no grupo de elite.

Seja $\rho_e > \frac{1}{2}$ um parâmetro escolhido pelo usuário para um BRKGA, que representa a probabilidade de um descendente herdar chaves de seu pai de elite. Seja n o número de genes no cromossomo de um indivíduo. Para $1 \leq i \leq n$, a i-ésima chave $c(i)$ do descendente c assume o valor da i-ésima chave $e(i)$ do pai de elite e com probabilidade ρ_e e o valor da i-ésima chave $\bar{e}(i)$ do pai não pertencente à elite $\bar{e}$ com probabilidade $1 - \rho_e$. Dessa maneira, é mais provável que o descendente herde características do pai de elite do que as do pai não pertencente à elite. Como assumimos que qualquer vetor de chaves aleatórias pode ser decodificado em uma solução, o descendente resultante do acasalamento é sempre válido, ou seja, pode ser decodificado em uma solução válida para o problema de tratado.

As heurísticas BRKGA são baseadas em uma estrutura meta-heurística de uso geral. Nesta estrutura, representada na Figura 2, existe uma clara divisão entre a parte independente e a parte dependente do problema. A parte independente não tem conhecimento do problema que está sendo resolvido, sendo limitada a percorrer o espaço de soluções viáveis. A única conexão com o problema de Otimização Combinatória que está sendo resolvido é a parte do algoritmo dependente do problema, onde o decodificador produz soluções a partir dos vetores de chaves aleatórias e calcula a aptidão dessas soluções. Portanto, para especificar uma heurística BRKGA, é necessário apenas definir sua representação cromossômica e o decodificador.

A seguir, descreveremos a implementação deste método para o Problema de Transformação de Strings por Reversões.

5.8 Implementação do BRKGA para o Problema de Transformação de Strings por Reversões

A implementação da meta-heurística BRKGA proposta para o Problema de Transformação de Strings por Reversões tem como entradas um par de strings compatíveis S e T, um número inteiro p, representando o tamanho da população a cada geração, um número inteiro g, representando o número de gerações, um número inteiro q_e, representando a quantidade de indivíduos na elite da população a cada geração, um número inteiro q_m, representando o número de mutantes da população a cada geração e um número real ρ_e, onde $\frac{1}{2} < \rho_e < 1$, representando a probabilidade de um descendente herdar partes de seu pai pertencente à elite no processo de *crossover*.

O funcionamento do algoritmo segue, em ordem, cada uma das etapas a seguir:

Decodificador e População Inicial: inicialmente, o algoritmo gerará p mapeamentos triviais para a string S e um único mapeamento trivial para T. Para cada

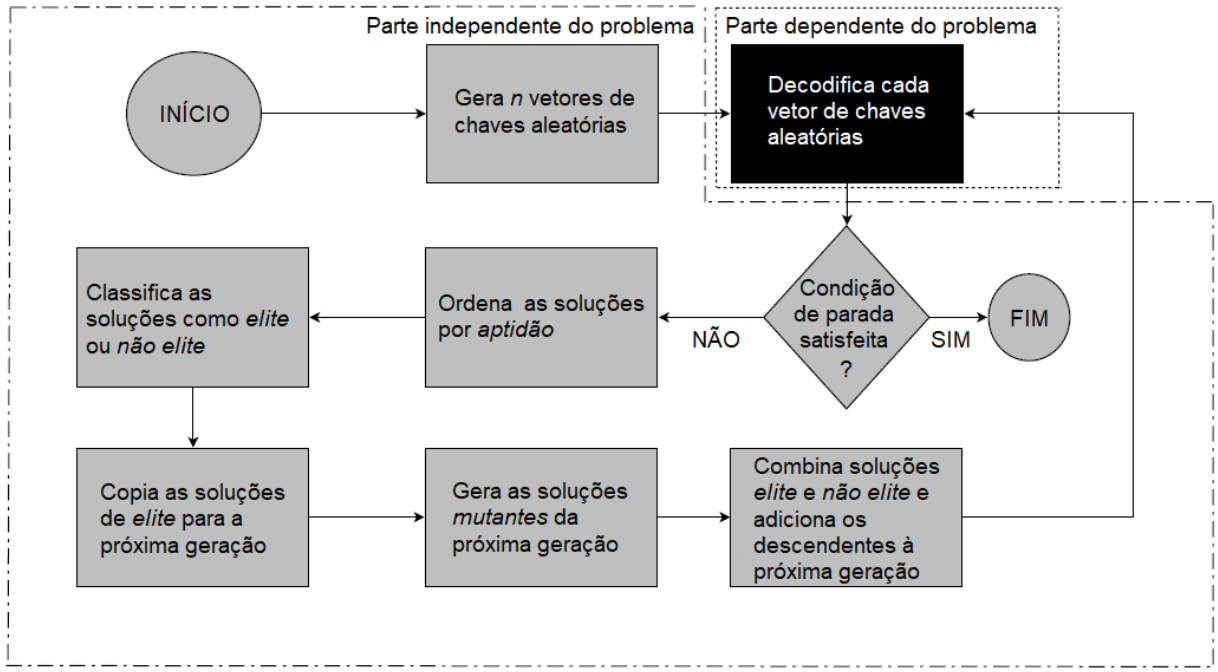

Figura 2: Fluxograma do BRKGA, traduzido e adaptado de Gonçalves e Resende [16].

elemento $m_{(i,j)}$ de cada sequência m_i de cada mapeamento trivial m gerado para S, será atribuída uma chave aleatória, representada por um número real no intervalo $[0, 1]$. O decodificador funcionará ordenando os elementos de cada uma das sequências m_i de cada mapeamento m por ordem não decrescente das chaves aleatórias associadas a cada elemento. Este processo é análogo ao embaralhamento dos elementos de cada uma das sequências dos mapeamentos triviais, e percebemos que o mapeamento resultante deste processo continua sendo um mapeamento válido para a string S. Cada um dos mapeamentos resultantes representará um indivíduo na população inicial e será associado a um valor de aptidão, calculado pela função de aptidão a seguir.

Função de Aptidão: para cada um dos p mapeamentos decodificados de S, é calculada uma solução, utilizando KS95, tendo como entrada a permutação gerada pelo mapeamento de S e a permutação gerada pelo mapeamento trivial único de T. Assim, teremos uma solução associada a cada um dos indivíduos da população. Quanto menor o valor da solução associada a um indivíduo, maior será seu valor de aptidão.

Elite: a elite da geração atual é formada pelos q_e indivíduos com os maiores valores de aptidão. Estes indivíduos são copiados sem alterações para a geração seguinte.

Mutação: para formar a partição mutante da geração seguinte, são gerados aleatoriamente q_m indivíduos (mapeamentos aleatórios).

Crossovers: o número q_c de indivíduos que devem ser gerados por *crossover* para a próxima geração é igual a $p - q_e - q_m$. Para gerar cada um dos descendentes, selecionamos aleatoriamente um pai de elite e um pai não pertencente à elite e, para formar cada sequência m_i do descendente, sorteamos um número real no intervalo $[0, 1]$. Se o número sorteado para uma sequência m_i for menor ou igual a ρ_e, a sequência herdada pelo descendente será a sequência m_i do pai de elite. Caso contrário, o descendente herdará a sequência m_i do pai não pertencente à elite. Note que escolhemos a estratégia de sortear um número real para decidir de quem herdar cada sequência m_i, e não cada elemento $m_{(i,j)}$. O motivo desta escolha é o fato do decodificador, como proposto, rearranjar os elementos $m_{(i,j)}$ de cada sequência m_i dos indivíduos. Assim, se escolhêssemos sortear um número para cada elemento $m_{(i,j)}$, o decodificador poderia transformar uma das sequências m_i do descendente em uma sequência que não representaria a herança de características de nenhum dos pais, o que iria contra o objetivo principal de um algoritmo genético.

Ao fim deste processo, a geração seguinte estará formada por um total de p indivíduos.

Após a criação de g gerações de p indivíduos, o algoritmo devolverá como solução para a Transformação por Reversões entre as string S e T originais o menor entre os valores das soluções associadas a indivíduos da geração g.

6 Resultados experimentais

Com o intuito de comparar, na prática, a qualidade das soluções dos algoritmos propostos, descritos nas seções anteriores, foram gerados 11 conjuntos de 100 entradas cada para teste. Cada entrada consiste em duas strings compatíveis, S e T, geradas aleatoriamente como descrito a seguir: uma string identidade S é criada e seus elementos são embaralhados de maneira uniforme (todas as permutações possíveis dos elementos desta string têm a mesma probabilidade de ocorrer); após o embaralhamento, copiamos a string resultante S para a string T, obtendo, agora, duas strings idênticas; por fim, são aplicadas *opt* reversões à string S, onde *opt* é um número inteiro definido para cada um dos conjuntos de 100 entradas.

Vale destacar que, como foram aplicadas *opt* reversões a partir de duas strings idênticas para criar cada uma das entradas, sabemos que a distância ótima entre as strings de cada par será, necessariamente, menor ou igual a *opt*.

Seja n o comprimento das string S e T, descritas sobre um alfabeto Σ, onde $|\Sigma| = k$, e seja $f(S, i)$ o número de ocorrências de cada símbolo de $i \in \Sigma$ na string S. Em todos os conjuntos definidos para os testes, cada par de strings S e T terá a mesma quantidade de ocorrências para cada símbolo, ou seja, $f(S, i) = f(S, j), \forall\, i, j \in \Sigma$. Denotaremos, então, a

quantidade de ocorrências de cada símbolo em cada uma das strings apenas por um número inteiro f. Consequentemente, para cada um dos onze conjuntos, teremos $n = k \cdot f$.

Para formar os onze conjuntos citados, definimos $f = 5$ para nove deles e, para os dois conjuntos restantes, $f = 2$. O comprimento das strings nos nove primeiros conjuntos foi definido no intervalo $\{20, 30, \ldots, 100\}$ Nos dois conjuntos onde $f = 2$, definimos $n \in \{50, 100\}$.

No algoritmos que recebem parâmetros adicionais, além do par de strings S e T, fizemos uma busca em conjuntos de valores diversos para cada um dos parâmetros, de modo a obter uma configuração que retornasse, em média, melhores resultados.

Para a heurística dos Mapeamentos Aleatórios, o único parâmetro adicional é a quantidade m de mapas aleatórios gerados para a string S. Testamos uma quantidade de mapeamentos que fosse suficiente para que a média dos valores das soluções geradas pelo algoritmo fosse próxima a *opt* para os conjuntos de strings com os menores valores de n, entre 20 e 40. O valor escolhido foi $m = 100000$.

Para a meta-heurística BRKGA, temos cinco parâmetros adicionais: p, g, q_e, q_m e ρ_e. Os conjuntos de valores, para cada um dos parâmetros, em que buscamos a melhor configuração foram os seguintes: $(p, g) \in \{(100, 1000), (200, 500), (500, 200), (1000, 100)\}$, $q_e \in \{0.15p, 0.2p, 0.3p, 0.4p\}$, $q_m \in \{0.15p, 0.2p, 0.3p, 0.4p\}$ e $\rho_e \in \{0.55, 0.6, 0.65, 0.7\}$. As combinações dos valores destes conjuntos de parâmetros foram aplicadas ao conjunto de 100 entradas onde $n = 50$. A configuração que resultou na menor média dos valores das soluções calculadas foi a seguinte: $p = 1000$, $g = 100$, $q_e = 0.3p$, $q_m = 0.3p$ e $\rho_e = 0.6$. Além desta configuração, também testamos, para todos os conjuntos de entradas, uma configuração arbitrária, apenas para a comparação entre as médias dos valores das soluções, partindo de valores diferentes para os parâmetros. A segunda configuração foi a seguinte: $p = 200$, $g = 500$, $q_e = 0.2p$, $q_m = 0.15p$ e $\rho_e = 0.6$.

A Tabela 1 mostra os resultados experimentais obtidos para cada um dos onze conjuntos de strings, representados por cada uma de suas linhas. As quatro primeiras colunas representam os valores que definem a classe de equivalência das strings de cada conjunto: o comprimento n, o tamanho k do alfabeto, a quantidade de ocorrências f de cada um dos símbolos e a quantidade *opt* de reversões aplicadas em uma das strings de cada par, a partir de duas strings idênticas.

Cada uma das demais colunas representa as médias dos valores das soluções calculadas por cada um dos algoritmos testados, para cada conjunto de 100 entradas. A coluna EXT-1 se refere aos resultados do Algoritmo 9, que busca em ambas as strings, a cada iteração, a reversão que mais aumente o valor de $lcp(S, T) + lcs(S, T)$. A coluna EXT-2 se refere ao algoritmo citado no fim da Seção 5.1, que une todas as variações de algoritmos da referida seção, que buscam aumentar o comprimento dos prefixos e/ou sufixos comuns. A coluna

BREAKS se refere ao Algoritmo 5, que busca, a cada iteração, eliminar a maior quantidade possível de breakpoints e, caso não seja possível, busca aumentar o valor de $lcp(S,T) + lcs(S,T)$. A coluna MAPAS se refere ao método heurístico de Mapeamentos Aleatórios, da Seção 5.6, tendo como parâmetro $m = 100000$. A coluna BRKGA-1 se refere à implementação da meta-heurística BRKGA, como descrito na Seção 5.8, com os parâmetros $p = 200$, $g = 500$, $q_e = 0.2p$, $q_m = 0.15p$ e $\rho_e = 0.6$. Já a coluna BRKGA-2 se refere à implementação da meta-heurística BRKGA com os parâmetros $p = 1000$, $g = 100$, $q_e = 0.3p$, $q_m = 0.3p$ e $\rho_e = 0.6$.

A última linha, MÉDIA ERRO, se refere à média do erro entre a estimativa de distância média d_{alg} calculada por um dos algoritmos para cada conjunto e a quantidade de reversões *opt* aplicadas a partir de strings idênticas para formar os pares do conjunto. Para cada conjunto i, o erro E_i da estimativa de distância foi calculado por $E_i = (d_{alg} - opt)/opt$. A média do erro representa, em porcentagem, quão longe de *opt* o algoritmo estimou a distância para as 1100 instâncias testadas.

Tabela 1: Resultados experimentais dos algoritmos propostos.

n	k	f	*opt*	EXT-1	EXT-2	BREAKS	MAPAS	BRKGA-1	BRKGA-2
20	4	5	10	6,76	6,14	9,35	7,37	6,17	5,67
30	6	5	10	11,10	9,70	15,36	14,04	9,68	8,48
40	8	5	15	17,04	15,64	24,46	21,87	14,94	12,97
50	10	5	15	20,73	19,16	28,87	29,87	18,12	14,87
60	12	5	20	28,45	26,87	40,30	38,57	24,41	19,98
70	14	5	20	31,48	29,39	42,21	46,76	26,76	21,16
80	16	5	25	38,99	36,73	52,76	55,65	34,78	27,21
90	18	5	25	41,23	39,28	55,94	64,23	36,58	28,90
100	20	5	25	46,41	43,60	59,12	73,02	41,56	33,32
50	25	2	15	21,96	19,39	19,33	17,96	14,61	14,61
100	50	2	25	42,49	38,59	32,99	44,52	25,62	25,61
MÉDIA ERRO				41,18%	30,34%	77,03%	86,83%	16,94%	−0,88%

Ao analisarmos os resultados da Tabela 1, podemos observar que o algoritmo MAPAS foi o que mais se distanciou, em média, do valor de *opt* para as instâncias testadas. Apesar de BREAKS ter mostrado resultados piores que MAPAS para alguns conjuntos com strings de menor comprimento n, este cenário se inverteu nos conjuntos com $f = 5$ e $n > 60$. Mesmo gerando 100 mil mapeamentos aleatórios para cada instância, os resultados mostraram que o crescimento dos valores de n e/ou de f aumenta muito o espaço de busca desta heurística, i.e., a quantidade de mapeamentos m é muito pequena em relação a quantidade de mapeamentos possíveis para a string que se está mapeando. Para uma string pertencente à classe de equivalência $\mathcal{L} = (f_0, f_1, \ldots, f_{k-1})$, a quantidade de mapeamentos possíveis M é dada por $M = f_0!f_1! \cdots f_{k-1}!$, o que implica, para strings onde todos os símbolos ocorrem a mesma

quantidade f de vezes, em $M = (f!)^f$. Calculando este valor para os conjuntos testados com os maiores valores de n e $f = 5$, podemos notar que 100 mil mapeamentos representa um valor extremamente pequeno em relação à quantidade de mapeamentos possíveis.

Na comparação entre os dois algoritmos que utilizam a ideia de aumentar os extremos comuns às strings, EXT-1 e EXT-2, o segundo sempre foi melhor, pois, como visto no final da Seção 5.1, EXT-1 é uma das variações contidas EXT-2. Contudo, a diferença entre os dois foi pequena nos conjuntos testados. Um fato interessante observado nos resultados destes dois algoritmos foi as médias dos valores de suas soluções não terem sido menores sempre que o valor de f foi menor, nos conjuntos com $n = 50$ e $n = 100$, tendo ocorrido apenas no último caso.

Entre os três algoritmos iniciais, os que aumentam as extremidades comuns, EXT-1 e EXT-2, se saíram bem melhor do que o que tenta sempre eliminar breakpoints, BREAKS, quando as strings tinham $f = 5$. Porém, para os dois conjuntos com $f = 2$, o algoritmo BREAKS levou vantagem sobre os dois primeiros. Este resultado é interessante pelo fato da estratégia de remover breakpoints funcionar melhor em permutações, que podemos enxergar como strings onde $f = 1$. Um exemplo disso é o Algoritmo 6 (KS95), que utiliza esta estratégia em permutações e possui um fator de aproximação bem definido.

Vemos, também, que as duas configurações do BRKGA levaram ampla vantagem sobre os demais algoritmos e, entre as duas, a segunda retornou distâncias menores, como já era esperado. Entre si, a segunda configuração foi melhor em todos os conjuntos com $f = 5$, mas nos conjuntos com $f = 2$, os resultados praticamente empataram. Talvez isto se deva ao fato de que o espaço de busca de soluções nestes dois conjuntos era significativamente menor que nos demais e, mesmo com a primeira configuração não tendo o melhor ajuste, foi o suficiente para encontrar soluções de boa qualidade neste espaço. Além disso, em muitos conjuntos, a segunda configuração retornou distâncias menores que o valor de *opt*, o que mostra que para algumas instâncias, o algoritmo retornou, possivelmente, soluções muito próximas às ótimas.

É importante destacar que o crescimento, mesmo que não tão expressivo, nos valores de n e f pode aumentar drasticamente o espaço de busca por mapeamentos, como revelado pelo cálculo da quantidade de mapeamentos possíveis para uma string. Dessa forma, para valores altos destas duas variáveis, mesmo o BRKGA com uma configuração bem ajustada pode necessitar de muitas gerações para explorar suficientemente o espaço de soluções, o que pode tornar este método inviável em relação ao consumo de tempo, ao buscarmos soluções com qualidade similar às retornadas nos testes realizados neste trabalho.

7 Conclusões e perspectivas de trabalhos futuros

Durante o período de pesquisa, conseguimos realizar muitas atividades que familiarizaram o aluno com o tema abordado, o introduziram à metodologia utilizada na área, o desafiaram a buscar novas soluções e implementá-las, a fim de comparar e reportar os resultados obtidos, e o incentivaram a refletir sobre os próximos passos que podem ser seguidos para obter resultados relevantes para os problemas estudados.

A partir dos conceitos estudados, tanto para os problemas em permutações quanto para em strings, propusemos diversos algoritmos para o Problema de Transformação de Strings por Reversões, utilizando diferentes estratégias para buscar soluções mais próximas às ótimas. Testes foram realizados e foi apresentada uma análise dos resultados obtidos, comparando a qualidade das soluções dos diferentes algoritmos.

Um algoritmo que retorna soluções ótimas também foi proposto e, mesmo não sendo eficiente no que diz respeito ao consumo de tempo, foi utilizado para calcular as distâncias exatas de algumas classes de strings, podendo auxiliar na identificação de padrões que norteiem o desenvolvimento de novos algoritmos.

Métodos heurísticos também foram estudados e implementados, sendo a aplicação da meta-heurística BRKGA o algoritmo que retornou os melhores resultados em nossos testes comparativos.

Futuramente, seria interessante estudar aspectos estruturais do Problema de Transformação de Strings por Reversões com o objetivo de melhorar os resultados referentes aos limitantes da distância de reversão, ou ao diâmetro das classes de equivalência, além de buscar relações entre propriedades das strings que possam nos auxiliar no desenvolvimento de algoritmos com mais garantias quanto à qualidade das soluções, como um fator de aproximação bem definido.

Referências

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

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

[3] J. C. Bean. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6(2):154–160, May 1994.

[4] P. Berman and M. Karpinski. On Some Tighter Inapproximability Results (Extended Abstract). In J. Wiedermann, P. E. Boas, and M. Nielsen, editors, *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICAL'1999)*, volume 1644 of *Lecture Notes in Computer Science*, pages 200–209. Springer-Verlag, London, UK, 1999.

[5] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-Approximation Algorithm for Sorting by Reversals. In R. Möhring and R. Raman, editors, *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag Berlin Heidelberg New York, Berlin/Heidelberg, Germany, 2002.

[6] L. Bulteau, G. Fertin, and C. Komusiewicz. (Prefix) Reversal Distance for (Signed) Strings with Few Blocks or Small Alphabets. *Journal of Discrete Algorithms*, 37:44–55, 2016.

[7] A. Caprara. Sorting Permutations by Reversals and Eulerian Cycle Decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110, 1999.

[8] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, and T. Jiang. Assignment of Orthologous Genes via Genome Rearrangement. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(4):302–315, 2005.

[9] B. Chitturi and I. H. Sudborough. Bounding Prefix Transposition Distance for Strings and Permutations. *Theoretical Computer Science*, 421:15–24, 2012.

[10] D. A. Christie. A 3/2-Approximation Algorithm for Sorting by Reversals. In H. Karloff, editor, *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'1998)*, pages 244–252, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics.

[11] D. A. Christie and R. W. Irving. Sorting Strings by Reversals and by Transpositions. *SIAM Journal on Discrete Mathematics*, 14(2):193–206, 2001.

[12] A. K. Dutta, M. Hasan, and M. S. Rahman. Prefix Transpositions on Binary and Ternary Strings. *Information Processing Letters*, 113(8):265–270, 2013.

[13] H. Dweighter. Problem E2569. *American Mathematical Monthly*, 82:1010, 1975.

[14] G. Fertin, L. Jankowiak, and G. Jean. Prefix and Suffix Reversals on Strings. In C. Iliopoulos, S. Puglisi, and E. Yilmaz, editors, *String Processing and Information Retrieval*, volume 9309 of *Lecture Notes in Computer Science*, pages 165–176. Springer International Publishing, Switzerland, 2015.

[15] D. E. Goldberg and J. H. Holland. Genetic algorithms and machine learning. *Machine Learning*, 3(2/3):95–99, 1988.

[16] J. F. Gonçalves and M. G. C. Resende. Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525, Aug. 2010.

[17] S. Hannenhalli and P. Pevzner. To Cut ... or Not to Cut (Applications of Comparative Physical Maps in Molecular Evolution). In E. Tardos, editor, *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'1996)*, pages 304–313, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[18] S. Hannenhalli and P. A. Pevzner. Transforming Cabbage into Turnip: Polynomial Algorithm for Sorting Signed Permutations by Reversals. *Journal of the ACM*, 46(1):1–27, 1999.

[19] C. Hurkens, L. van Iersel, J. Keijsper, S. Kelk, L. Stougie, and J. Tromp. Prefix Reversals on Binary and Ternary Strings. *SIAM Journal on Discrete Mathematics*, 21(3):592–611, 2007.

[20] J. D. Kececioglu and D. Sankoff. Exact and Approximation Algorithms for Sorting by Reversals, with Application to Genome Rearrangement. *Algorithmica*, 13:180–210, 1995.

[21] P. Kolman and T. Waleń. Reversal Distance for Strings with Duplicates: Linear Time Approximation Using Hitting Set. In T. Erlebach and C. Kaklamanis, editors, *Proceedings of the 4th International Workshop on Approximation and Online Algorithms (WAOA'2006)*, pages 279–289, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[22] C. N. Lintzmayer. *The Problem of Sorting Permutations by Prefix and Suffix Rearrangements*. PhD thesis, University of Campinas, Institute of Computing, 2016. In English.

[23] A. J. Radcliffe, A. D. Scott, and E. L. Wilmer. Reversals and Transpositions Over Finite Alphabets. *SIAM Journal on Discrete Mathematics*, 19(1):224–244, 2005.

[24] I. R. Sucupira. Métodos heurísticos genéricos: Meta-heurísticas e hiper-heurísticas. *USP: São Paulo*, 2004.

A Trabalhos relacionados

A.1 Ordenação por reversões em permutações (sem sinais)

O problema de ordenação de permutações por reversões é um dos problemas mais estudados na área de rearranjo de genomas. O problema foi provado ser NP-difícil [7] e não ser aproximável dentro de um fator de aproximação melhor que 1.0008 [4].

Bafna e Pevzner [2] mostraram que o diâmetro de reversão em permutações sem sinais é igual a $n-1$, sendo n o comprimento da permutação. Porém, para cada grupo simétrico S_n, apenas duas permutações necessitam de $n-1$ reversões para serem ordenadas: a chamada *permutação de Gollan*, e a sua inversa.

Sobre os limitantes do problema, podemos citar aqueles que se baseiam na quantidade de breakpoints fortes na permutação π, denotados por $sb(\pi)$. Como limitante inferior, temos $d_\rho(\pi) \geq sb(\pi)/2$ [2]. Como limitante superior, temos $d_\rho(\pi) \leq sb(\pi) - 1$ [20].

A melhor aproximação até hoje para este problema é (11/8)-aproximação proposta por Berman *et al.* [5], que consiste em um algoritmo que melhorou a resolução do problema da decomposição em ciclos alternantes, problema este muito relacionado com a distância de reversão em permutações sem sinais [7, 10].

A.2 Ordenação por reversões em permutações com sinais

Ao contrário do problema da distância de reversão em permutações sem sinais, que é NP-difícil, o problema de reversões em permutações com sinais é polinomial [18].

O diâmetro de reversão com sinais em permutações de comprimento n é igual a $n+1$, sendo atingido pelas permutações que tenham apenas um ciclo em seu grafo de breakpoints e que tenham apenas elementos positivos.

A distância de reversão com sinais de uma permutação com sinais π, denotada $d_{\rho_p}(\pi)$, é dada por $d_{\rho_p}(\pi) = n + 1 - c(BG(\pi)) + t(\pi)$, onde $c(BG(\pi))$ é a quantidade de ciclos no grafo de breakpoints de π, e $t(\pi)$ é a quantidade de folhas da árvores de componentes não-orientados [18]. Ela pode ser calculada em tempo $O(n)$ [1].

A.3 Resultados existentes para os problemas em strings

Existem vários resultados com relação a limitantes para as distâncias e os diâmetros dos problemas de transformação de strings por reversões e/ou transposições, nas versões gerais ou restritas ao prefixo e ao sufixo [6, 11, 12, 14, 19, 23]. Ainda assim, poucos são os resultados algorítmicos para esses problemas.

Para a variação de transformação de strings, sabemos que quando os modelos de rearranjo envolvem reversões, reversões com sinais, transposições, reversões de prefixo, reversões de prefixo com sinais, transposições de prefixo, ou reversões de prefixo e sufixo, os problemas são NP-difíceis [6, 8, 9, 11, 14, 19, 23]. Além disso, para reversões e transposições de prefixo, existe uma $O(\log n \log^* n)$-aproximação [9] e, para reversões e reversões com sinais, existe uma $\theta(t)$-aproximação, para quando a quantidade de vezes que cada símbolo do alfabeto aparece na string é no máximo t [21]. Para reversões, existe ainda um PTAS para instâncias nas quais a distância entre as duas strings de tamanho n é no máximo cn, para $c > 0$ [23].

A.4 Partição Comum Mínima em Strings (MCSP)

O *Problema da Partição Comum Mínima em Strings* (do inglês *Minimum Common String Partition Problem*, e que chamaremos de *MCSP*) consiste em encontrar uma partição comum entre S e T com tamanho mínimo, denotada por $\texttt{MCSP}(S, T)$.

Quando só é permitido que cada símbolo ocorra no máximo k vezes em cada string, chamamos o problema de *k-MCSP*.

Na versão *sem sinais* do problema, chamada apenas de *MCSP*, a entrada consiste em duas strings sem sinais, e a relação utilizada é $=$. Na versão *com sinais* do problema, chamada de *SMCSP*, a entrada consiste em duas strings com sinais, e a relação utilizada é $\cong$.

Para strings sem sinais, há outra variação do problema, chamada de *MCSP reverso (RMCSP)*, na qual strings sem sinais são comparadas utilizando a relação $\cong$.

O SMCSP foi introduzido por Chen *et al.* [8] como uma ferramenta para lidar com o problema da Transformação de Strings com Sinais por Reversões.

A.5 Hitting Set Mínimo

Seja U um conjunto e $\mathcal{S}$ uma coleção de subconjuntos de U, isto é, $\mathcal{S} = \{S_1, \ldots, S_k\}$ tal que $S_i \subseteq U$, para $1 \leq i \leq k$. O *Problema do Hitting Set Mínimo* consiste em encontrar o menor conjunto $H \subseteq U$ tal que $H \cap S_i \neq \emptyset$, para cada $1 \leq i \leq k$.

Kolman e Walen [21] utilizaram um algoritmo para o Problema do Hitting Set Mínimo como parte de sua $\theta(k)$-aproximação para o k-MCSP.

B Soluções ótimas

Com o intuito de investigar soluções ótimas para o problema de Transformação de Strings por Reversão, decidimos calcular a distribuição da quantidade de breakpoints, as distâncias exatas e a distribuição destas distâncias para algumas classes de strings.

Para tanto, foram desenvolvidos programas para as seguintes tarefas:

- encontrar todas as strings existentes em uma classe de equivalência;
- encontrar todos os pares possíveis dentro da classe de equivalência, sem que haja repetição de pares (uma string S nunca é comparada com ela mesma, mas sempre com uma string T lexicograficamente maior que S);
- calcular a quantidade de breakpoints de reversão entre os pares de strings, agrupando-os por quantidade de breakpoints;
- calcular, para cada um dos grupos criados, as distâncias entre todos os pares de strings, utilizando um algoritmo de força bruta.

A seguir, explicamos a realização de cada um dos tópicos acima e, ao final, dispomos os resultados em tabelas que relacionam a quantidade de breakpoints às distâncias.

Definimos o *piso de uma string* S como sendo o elemento de maior posição de S que é menor do que o elemento imediatamente seguinte (que deve necessariamente existir). Note que a única string de uma classe de equivalência $\mathcal{L}(a_1, \ldots, a_k)$ que não tem piso é string identidade reversa, S_ι^R. Isto se deve ao fato de que S_ι^R é a única string da classe para a qual não há elementos consecutivos i e j tais que $i < j$. Dada uma string S, a função PISO(S) retorna a posição do piso de S, caso exista. Se o elemento não existir, a função retorna -1.

Definimos o *teto de um elemento x em um segmento de uma string S* como sendo o menor elemento que é maior do que x em tal segmento. Se existirem vários elementos de mesmo valor no segmento que satisfaçam a definição, consideramos o de menor posição, i.e., o elemento mais à esquerda. Dado um valor x, uma string S e duas posições ini e fim, que definirão o intervalo fechado em que buscaremos pelo teto de x, a função TETO(x, S, ini, fim) devolve a posição do elemento buscado. Ao contrário do piso, no algoritmo proposto sempre haverá um teto para qualquer elemento.

O Algoritmo 7 mostra como funciona o processo de obtenção de todas as strings de uma classe de equivalência. Seja S_ι a string identidade de uma determinada classe de equivalência $\mathcal{L}(a_1, \ldots, a_k)$. Ao passarmos S_ι como entrada para o algoritmo, ele devolve uma sequência $\mathcal{S}$, contendo todas as strings pertencentes a $\mathcal{L}$, sem repetição, e ordenadas lexicograficamente.

Sabendo que em $\mathcal{L}(a_1, \ldots, a_k)$ cada elemento a_i, com $1 \leq i \leq k$, representa a quantidade de ocorrências do símbolo i em cada uma das strings desta classe, e que $a_1 + a_2 + \cdots + a_k = n$, concluímos que o tamanho da sequência $\mathcal{S}$ é $|\mathcal{S}| = \frac{n!}{a_1! \cdot a_2! \cdot \cdots \cdot a_k!}$.

Algoritmo 7: CRIA_CLASSE(k, S_ι)

Entrada: inteiro k e string identidade S_ι da classe de equivalência $\mathcal{L}(a_1, \ldots, a_k)$

$n \leftarrow a_1 + a_2 + \cdots + a_k$
$\mathcal{S} \leftarrow (S_\iota)$
$S \leftarrow S_\iota$
faça
  $t \leftarrow$ TETO($s_p,\ S,\ p+1,\ n$)
  troca de posição os elementos s_p e s_t em S
  ordena de forma não-decrescente a substring $s_{p+1} \ldots s_n$
  $p \leftarrow$ PISO(S)
  adiciona S ao final da sequência $\mathcal{S}$
enquanto $p \neq -1$

Saída: a sequência $\mathcal{S}$, de todas as strings da classe $\mathcal{L}$, ordenadas de forma não-decrescente, lexicograficamente

Após obtermos a sequência $\mathcal{S}$ de todas as strings possíveis de uma determinada classe de equivalência $\mathcal{L}(a_1, \ldots, a_k)$, podemos utilizar $\mathcal{S}$ para criar um conjunto $\mathcal{P}$, de pares ordenados (X, Y) tais que $X, Y \in \mathcal{S}$, e X seja lexicograficamente menor que Y. Em outras palavras, o conjunto $\mathcal{P}$ é o conjunto de todos os pares diferentes de strings (X, Y) de uma classe de equivalência $\mathcal{L}$.

Sabemos que o conjunto formado por todos os pares de elementos de $\mathcal{S}$ tem tamanho $|\mathcal{S} \times \mathcal{S}| = |\mathcal{S}|^2$. Ao construir o conjunto $\mathcal{P}$, não consideraremos pares formados por strings idênticas. Seja $\mathcal{M}$ o conjunto formado por todos os pares de elementos de $\mathcal{S}$, sem pares formados por strings idênticas, tendo tamanho $|\mathcal{M}| = |\mathcal{S}|^2 - |\mathcal{S}|$. Como o conjunto $\mathcal{P}$ conterá apenas pares onde uma string é lexicograficamente menor que a outra, se ele contém um par (X, Y), então não conterá o par (Y, X). Portanto, o tamanho do conjunto $\mathcal{P}$ será dado pela metade de $\mathcal{M}$, ou seja, $|\mathcal{P}| = \frac{|\mathcal{M}|}{2} = \frac{|\mathcal{S}|^2 - |\mathcal{S}|}{2}$.

Com o conjunto $\mathcal{P}$ de pares de strings compatíveis, calculamos $b_\rho(X, Y)$ para cada par $(X, Y) \in \mathcal{P}$, onde cada cálculo é feito em tempo linear, pois basta percorrer as strings uma única vez para determinar o número de breakpoints entre elas. Feito isso, agrupamos os pares por número de breakpoints de reversão, a fim de analisar a distribuição deste parâmetro na classe de equivalência e calcular as distâncias em cada grupo.

Dado um par $(X, Y) \in \mathcal{P}$, desenvolvemos um algoritmo de força bruta que verifica se uma sequência de reversões aplicada em X a transforma em Y, testando todas as possíveis sequências com apenas uma reversão, depois todas as sequências de duas reversões, e assim por diante. Para criar uma sequência, geramos todas as reversões possíveis que podem ser aplicadas a X, sendo que essas reversões $\rho(i, j)$ são criadas em ordem crescente de i e j, onde $1 \leq i < j \leq |X|$. Nunca aplicamos a mesma reversão de forma subsequente.

Por exemplo, consideremos um par de strings de comprimento n que tem como uma das

soluções ótimas a sequência $(\rho(1,2),\ \rho(1,4),\ \rho(3,5))$. O algoritmo testa, primeiro, todas as sequências de uma reversão: $(\rho(1,2))$, $(\rho(1,3))$, $\ldots$, $(\rho(n-2,n))$, $(\rho(n-1,n))$. Depois, testa todas as sequências de duas reversões: $(\rho(1,2),\ \rho(1,3))$, $(\rho(1,2),\ \rho(1,4))$, $\ldots$, $(\rho(n-2,n),\ \rho(n-1,n))$, $(\rho(n-1,n),\ \rho(n-2,n))$. Não encontrando uma solução válida com duas reversões, o algoritmo utiliza o mesmo raciocínio para testar todas as possíveis sequências de três reversões, caso em que vai encontrar a sequência válida $(\rho(1,2),\ \rho(1,4),\ \rho(3,5))$.

Ao encontrar a primeira sequência válida, o algoritmo devolve o tamanho desta (no exemplo acima, devolve o valor 3), e termina sua execução.

Como os testes são executados em ordem crescente do tamanho das sequências, a primeira sequência que transformar X em Y será uma sequência ótima e, portanto, seu tamanho será uma solução ótima para a distância de reversões.

Sendo X e Y compatíveis, sabemos que o algoritmo terminará sua execução, pois haverá uma sequência de no máximo $(n - a_{max})$ reversões que transformará X em Y, onde a_{max} é a quantidade de ocorrências do símbolo mais frequente [23].

Agrupamos então todos os pares strings de $\mathcal{P}$ com relação ao número de breakpoints entre as strings de cada par. Por fim, calculamos a distância para cada um dos grupos, criando tabelas que relacionam a quantidade de breakpoints à quantidade mínima de reversões necessárias para transformar as strings.

O procedimento descrito foi realizado, até agora, para todas classes de equivalência que formam strings de comprimento n, com $5 \leq n \leq 8$, descritas sobre um alfabeto Σ, com $|\Sigma| = k = 4$. Os dados obtidos encontram-se nas tabelas ao final desta seção.

Após a classificação das distâncias conforme a quantidade de breakpoints em cada classe, como mostram as tabelas ao final da seção, pudemos analisar alguns pares de strings com comportamento "inesperado" quanto à relação distância/breakpoints. Abaixo, relacionamos alguns casos importantes e destacamos e exemplificamos os casos inesperados encontrados.

Classe $\mathcal{L}(2,2,2,2)$. O diâmetro é igual a 6, e apenas 24 pares o atingem, todos contendo 9 breakpoints (p. ex. $S = 03032121$ e $T = 33101022$). Além disso, a maior distância entre os pares com 0 breakpoints é igual a 3 (p. ex. $S = 01023213$ e $T = 02310123$). Temos, também, pares com 2 breakpoints, mas que precisam de 4 reversões para serem transformados (p. ex. $S = 01032312$ e $T = 30210132$).

Classe $\mathcal{L}(3,2,2,1)$. O diâmetro é igual a 5, e os pares que o alcançam têm de 4 a 9 breakpoints. Também há pares com 0 breakpoints com distância igual a 3 (p. ex. $S = 00120132$ e $T = 01002312$), e pares com 2 breakpoints com distância igual a 4 (p. ex. $S = 00112032$ e $T = 10123002$).

Classe $\mathcal{L}(3,3,1,1)$. O diâmetro é igual a 5, e os pares que o alcançam têm de 6 a 9 breakpoints. Se destacam 8 pares com 0 breakpoints com distância igual a 4 (p. ex. $S = 11201300$ e $T = 12001130$), e pares com 2 breakpoints com distância igual a 4 (p. ex. $S = 00103211$ e $T = 01100312$).

Classe $\mathcal{L}(4,2,1,1)$. Não há pares com 9 breakpoints. O diâmetro de reversão é igual a 4, e os pares que o alcançam têm de 2 a 8 breakpoints. Se destacam os pares com 2 breakpoints que atingem o diâmetro (p. ex. $S = 00102031$ e $T = 02001103$).

Classe $\mathcal{L}(5,1,1,1)$ Não há pares com mais de 6 breakpoints (que, neste caso, seriam de 7 a 9 breakpoints). O diâmetro é igual a 3, e os pares que o alcançam têm de 0 a 6 breakpoints. Aqui, se destaca o fato de pares com 0 breakpoints atingirem o diâmetro (p. ex. $S = 00023010$ e $T = 02300100$).

Classe $\mathcal{L}(2,2,2,1)$. O diâmetro é igual a 5, e os pares que o alcançam têm 7 ou 8 breakpoints. Aqui, há 24 pares com 0 breakpoints e distância igual a 3 (p. ex. $S = 0230121$ e $T = 0321201$).

Classe $\mathcal{L}(3,2,1,1)$. O diâmetro é igual a 4, e os pares que o alcançam têm de 2 a 8 breakpoints. Destacam-se os 26 pares com 0 breakpoints com distância igual a 3 (p. ex. $S = 0031201$ e $T = 0310021$), 12 pares com 2 breakpoints com distância igual a 4 (p. ex. $S = 0021103$ e $T = 0210031$), e apenas 16 pares que possuem 8 breakpoints (p. ex. $S = 0301012$ e $T = 1132000$).

Classe $\mathcal{L}(4,1,1,1)$. Não há pares com 7 ou 8 breakpoints. O diâmetro é igual a 3, e os pares que o alcançam têm de 2 a 6 breakpoints. Os pares com 0 breakpoints têm distância igual a 1 ou 2. Se destacam os pares com 2 breakpoints que atingem o diâmetro (p. ex. $S = 0001203$ e $T = 0123000$).

Classe $\mathcal{L}(2,2,1,1)$. O diâmetro é igual a 4, e os pares que o alcançam têm de 4 a 7 breakpoints. Os pares com 0 breakpoints têm distância igual a 1 ou 2. Destaque para o fato de existirem apenas 52 pares com 7 breakpoints (p. ex. $S = 010123$ e $T = 113002$).

Classe $\mathcal{L}(3,1,1,1)$. Não há pares com 7 ou 8 breakpoints. O diâmetro é igual a 3, e os pares que o alcançam têm de 2 a 6 breakpoints. Os pares com 0 breakpoints têm distância igual a 1 ou 2. Destaque para o fato de existirem 48 pares com 2 breakpoints que atingem o diâmetro (p. ex. $S = 001320$ e $T = 103200$).

Classe $\mathcal{L}(2,1,1,1)$. O diâmetro é igual a 3, os pares que o alcançam têm de 3 a 6 breakpoints, e todos pares com 5 ou 6 breakpoints alcançam o diâmetro. Aqui, destacam-se os 9 pares

com 0 breakpoints, que têm todos distância igual a 1 (p. ex. $S = 30120$ e $T = 30210$), e os 12 pares com 6 breakpoints, todos atingindo o diâmetro (p. ex. $S = 00312$ e $T = 32010$).

Tabela 2: Classe de equivalência $\mathcal{L}(2, 2, 2, 2)$.

	Distância						
# breakpoints	1	2	3	4	5	6	Total
0	2160	3840	1296	0	0	0	7296
1	0	0	0	0	0	0	0
2	28080	86592	69108	2904	0	0	186684
3	0	129468	206640	24792	0	0	360900
4	0	164220	602604	137208	0	0	904032
5	0	0	570048	321024	552	0	891624
6	0	0	200964	397224	4176	0	602364
7	0	0	0	175284	13032	0	188316
8	0	0	0	19140	12288	0	31428
9	0	0	0	0	1272	24	1296
Total	30240	384120	1650660	1077576	31320	24	3173940

Tabela 3: Classe de equivalência $\mathcal{L}(3, 2, 2, 1)$.

	Distância					
# breakpoints	1	2	3	4	5	Total
0	1830	2886	658	0	0	5374
1	0	0	0	0	0	0
2	17490	62682	41170	908	0	122250
3	0	71144	120002	8408	0	199554
4	0	90576	310096	48404	16	449092
5	0	0	255206	109162	20	364388
6	0	0	88716	119250	416	208382
7	0	0	0	52882	1184	54066
8	0	0	0	5196	1874	7070
9	0	0	0	0	184	184
Total	19320	227288	815848	344210	3694	1410360

Tabela 4: Classe de equivalência $\mathcal{L}(3,3,1,1)$.

	Distância					
# breakpoints	1	2	3	4	5	Total
0	1440	2244	484	8	0	4176
1	0	0	0	0	0	0
2	10880	43022	26718	652	0	81272
3	0	37792	62496	3712	0	104000
4	0	50468	149340	14368	0	214176
5	0	0	106512	29048	0	135560
6	0	0	40440	29452	4	69896
7	0	0	0	15216	80	15296
8	0	0	0	1892	300	2192
9	0	0	0	0	72	72
Total	12320	133526	385990	94348	456	656640

Tabela 5: Classe de equivalência $\mathcal{L}(4,2,1,1)$.

	Distância				
# breakpoints	1	2	3	4	Total
0	1305	1872	366	0	3543
1	0	0	0	0	0
2	7515	33320	17924	224	58983
3	0	22896	42132	1422	66450
4	0	30612	89084	6682	126378
5	0	0	50460	14634	65094
6	0	0	17942	9346	27288
7	0	0	0	4308	4308
8	0	0	0	336	336
9	0	0	0	0	0
Total	8820	88700	217908	36952	352380

Tabela 6: Classe de equivalência $\mathcal{L}(5,1,1,1)$.

	Distância			
# breakpoints	1	2	3	Total
0	720	1080	60	1860
1	0	0	0	0
2	2304	11928	5088	19320
3	0	4668	7884	12552
4	0	5940	12216	18156
5	0	0	3336	3336
6	0	0	1056	1056
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
Total	3024	23616	29640	56280

Tabela 7: Classe de equivalência $\mathcal{L}(2,2,2,1)$.

	Distância					
# breakpoints	1	2	3	4	5	Total
0	360	378	24	0	0	762
1	0	0	0	0	0	0
2	5310	10692	3390	0	0	19392
3	0	18126	13602	204	0	31932
4	0	21369	43164	1980	0	66513
5	0	0	43818	6936	0	50754
6	0	0	13176	11100	0	24276
7	0	0	0	4254	36	4290
8	0	0	0	180	36	216
Total	5670	50565	117174	24654	72	198135

Tabela 8: Classe de equivalência $\mathcal{L}(3, 2, 1, 1)$.

	Distância				
# breakpoints	1	2	3	4	Total
0	340	296	26	0	662
1	0	0	0	0	0
2	3230	8086	2128	12	13456
3	0	9226	8004	68	17298
4	0	11132	19804	510	31446
5	0	0	16018	1616	17634
6	0	0	4726	1876	6602
7	0	0	0	876	876
8	0	0	0	16	16
Total	3570	28740	50706	4974	87990

Tabela 9: Classe de equivalência $\mathcal{L}(4, 1, 1, 1)$.

	Distância			
# breakpoints	1	2	3	Total
0	270	222	0	492
1	0	0	0	0
2	1305	4344	822	6471
3	0	2622	2754	5376
4	0	3066	4338	7404
5	0	0	1806	1806
6	0	0	396	396
7	0	0	0	0
8	0	0	0	0
Total	1575	10254	10116	21945

Tabela 10: Classe de equivalência $\mathcal{L}(2, 2, 1, 1)$.

	Distância				
# breakpoints	1	2	3	4	Total
0	60	30	0	0	90
1	0	0	0	0	0
2	1110	1310	136	0	2556
3	0	2752	782	0	3534
4	0	2908	2882	20	5810
5	0	0	3142	32	3174
6	0	0	686	208	894
7	0	0	0	52	52
Total	1170	7000	7628	312	16110

Tabela 11: Classe de equivalência $\mathcal{L}(3,1,1,1)$.

	Distância			
# breakpoints	1	2	3	Total
0	72	24	0	96
1	0	0	0	0
2	648	1128	48	1824
3	0	1296	600	1896
4	0	1308	1128	2436
5	0	0	792	792
6	0	0	96	96
7	0	0	0	0
Total	720	3756	2664	7140

Tabela 12: Classe de equivalência $\mathcal{L}(2,1,1,1)$.

	Distância			
# breakpoints	1	2	3	Total
0	9	0	0	9
1	0	0	0	0
2	261	150	0	411
3	0	492	36	528
4	0	402	186	588
5	0	0	222	222
6	0	0	12	12
Total	270	1044	456	1770

C Pseudocódigos dos algoritmos desenvolvidos

Algoritmo 8: TRATANDO_JUNTAS(S, T, n)

```
1  qtdRev ← 0
2  i ← 1
3  enquanto i < n faça
4      se s_i ≠ t_i então
5          j ← i + 1
6          melhorJ ← 0
7          maiorSub ← 0
8          reverteS ← verdadeiro
9          enquanto j ≤ n faça
10             se s_j = t_i então
11                 tamSub ← 1
12                 enquanto t_{i+tamSub} = s_{j−tamSub} faça
13                     tamSub ← tamSub + 1
14                 se tamSub > maiorSub então
15                     maiorSub ← tamSub
16                     melhorJ ← j
17                     reverteS ← verdadeiro
18             se t_j = s_i então
19                 tamSub ← 1
20                 enquanto s_{i+tamSub} = t_{j−tamSub} faça
21                     tamSub ← tamSub + 1
22                 se tamSub > maiorSub então
23                     maiorSub ← tamSub
24                     melhorJ ← j
25                     reverteS ← falso
26             j ← j + 1
27         se reverteS = verdadeiro então
28             S ← S · ρ(i, melhorJ)
29         senão
30             T ← T · ρ(i, melhorJ)
31         qtdRev ← qtdRev + 1
32     i ← i + 1
33 retorna qtdRev
```

Algoritmo 9: CORRIGE_EXTREMOS(S, T, n)

```
1  qtdRev ← 0
2  i ← 1
3  j ← n
4  enquanto i < j faça
5      se s_i ≠ t_i ou s_j ≠ t_j então
6          encontra ρ_ps(i, a) com maior valor lcp(S · ρ_ps(i, a), T)
7          encontra ρ_pt(i, b) com maior valor lcp(S, T · ρ_pt(i, b))
8          encontra ρ_ss(i, c) com maior valor lcs(S · ρ_ss(i, c), T)
9          encontra ρ_st(i, d) com maior valor lcs(S, T · ρ_st(i, d))
10         ρ(x, y) ← a reversão que mais aumente (lcp(S, T) + lcs(S, T)), entre
             ρ_ps(i, a), ρ_pt(i, b), ρ_ss(c, j), ρ_st(d, j)
11         se ρ(x, y) = ρ_ps(i, a) ou ρ(x, y) = ρ_ss(c, j) então
12             S ← S · ρ(x, y)
13         se ρ(x, y) = ρ_pt(i, b) ou ρ(x, y) = ρ_st(d, j) então
14             T ← T · ρ(x, y)
15         qtdRev ← qtdRev + 1
16     se s_i = t_i então
17         i ← i + 1
18     se s_j = t_j então
19         j ← j − 1
20 retorna qtdRev
```