Um algoritmo para a construção de vetores de sufixo generalizados em memória externa

Felipe Alves da Louza

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura:_____

Um algoritmo para a construção de vetores de sufixo generalizados em memória externa

Felipe Alves da Louza

Orientadora: **Profa. Dra. Cristina Dutra de Aguiar Ciferri**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

USP – São Carlos
Fevereiro de 2014

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

LL895a
a

Louza, Felipe Alves da
Um algoritmo para construção de vetores de sufixo generalizados em memória externa / Felipe Alves da Louza; orientadora Cristina Dutra de Aguiar Ciferri. -- São Carlos, 2013.
91 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2013.

1. vetor de sufixo generalizado. 2. memória externa. 3. indexação. 4. dados biológicos. 5. montagem de genomas. I. Ciferri, Cristina Dutra de Aguiar, orient. II. Título.

A austera disciplina antecede a espontaneidade do amor.
Humberto de Campos

Agradecimentos

Agradeço a Deus, sem Ele nada é possível.

Agradeço à minha família, já incluindo minha querida Camila.

Agradeço à minha orientadora Cristina, pelo trabalho, empenho e amizade.

Agradeço aos professores Guilherme P. Telles, Steve Hoffmann e Peter Stadler, que constribuiram com o desenvolvimento deste trabalho.

Agradeço aos meus amigos, matemáticos e não matemáticos.

Agradeço à FAPESP pelo apoio financeiro.

E agradeço a todos os professores e funcionários do ICMC-USP.

Resumo

O vetor de sufixo é uma estrutura de dados importante utilizada em muitos problemas que envolvem cadeias de caracteres. Na literatura, muitos trabalhos têm sido propostos para a construção de vetores de sufixo em memória externa. Entretanto, esses trabalhos não enfocam conjuntos de cadeias, ou seja, não consideram vetores de sufixo generalizados. Essa limitação motiva esta dissertação, a qual avança no estado da arte apresentando o algoritmo eGSA, o primeiro algoritmo proposto para a construção de vetores de sufixo generalizados aumentado com o vetor de prefixo comum mais longo (LCP) e com a transformada de *Burrows-Wheeler* (BWT) em memória externa. A dissertação foi desenvolvida dentro do contexto de bioinformática, já que avanços tecnológicos recentes têm aumentado o volume de dados biológicos disponíveis, os quais são armazenados como cadeias de caracteres. O algoritmo eGSA foi validado por meio de testes de desempenho com dados reais envolvendo sequências grandes, como DNA, e sequências pequenas, como proteínas. Com relação aos testes comparativos com conjuntos de grandes cadeias de DNA, o algoritmo proposto foi comparado com o algoritmo correlato mais eficiente na literatura de construção de vetores de sufixo, o qual foi adaptado para construção de vetores generalizados. O algoritmo eGSA obteve um tempo médio de 3,2 a 8,3 vezes menor do que o algoritmo correlato e consumiu 50% menos de memória. Para conjuntos de cadeias pequenas de proteínas, foram realizados testes de desempenho apenas com o eGSA, já que no melhor do nosso conhecimento, não existem trabalhos correlatos que possam ser adaptados. Comparado com o tempo médio para conjuntos de cadeias grandes, o eGSA obteve tempos competitivos para conjuntos de cadeias pequenas. Portanto, os resultados dos testes demonstraram que o algoritmo proposto pode ser aplicado eficientemente para indexar tanto conjuntos de cadeias grandes quanto conjuntos de cadeias pequenas.

Palavras-chave: vetor de sufixo generalizado, memória externa, indexação, dados biológicos, montagem de genomas.

Abstract

The suffix array is an important data structure used in several string processing problems. In the literature, several approaches have been proposed to deal with external memory suffix array construction. However, these approaches are not specifically aimed to index sets of strings, that is, they do not consider generalized suffix arrays. This limitation motivates this master's thesis, which presents eGSA, the first external memory algorithm developed to construct generalized suffix arrays enhanced with the longest common prefix array (LCP) and the Burrows-Wheeler transform (BWT). We especially focus on the context of bioinformatics, as recent technological advances have increased the volume of biological data available, which are stored as strings. The eGSA algorithm was validated through performance tests with real data from DNA and proteins sequences. Regarding performance tests with large strings of DNA, we compared our algorithm with the most efficient and related suffix array construction algorithm in the literature, which was adapted to construct generalized arrays. The results demonstrated that our algorithm reduced the time spent by a factor of 3.2 to 8.3 and consumed 50% less memory. For sets of small strings of proteins, tests were performed only with the eGSA, since to the best of our knowledge, there is no related work that can be adapted. Compared to the average time spent to index sets of large strings, the eGSA obtained competitive times to index sets of small strings. Therefore, the performance tests demonstrated that the proposed algorithm can be applied efficiently to index both sets of large strings and sets of small strings.

Keywords: generalized suffix array, external memory, indexing, biological data, genome assembly.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo

1

Introdução

Esta dissertação investiga a construção de vetores de sufixo generalizados em memória externa para dados biológicos. Em particular, é proposto o algoritmo eGSA (acrônimo para *External Generalized Enhanced Suffix Array Construction Algorithm*) [Louza et al., 2013], o primeiro algoritmo proposto para a construção de vetores de sufixo generalizados aumentados com o vetor de prefixo comum mais longo (LCP) e com a transformada de *Burrows-Wheeler* (BWT) em memória externa.

No contexto desta dissertação, são considerados dados biológicos referentes às sequências de DNA e de proteínas, as quais são armazenadas como cadeias de caracteres em bancos de dados biológicos (BDBs). Portanto, a motivação para o desenvolvimento desta dissertação é feita dentro do contexto da bioinformática, embora o algoritmo eGSA possa ser usado para a indexação de quaisquer tipos de cadeias de caracteres.

Em bioinformática, os avanços tecnológicos recentes têm diminuído o custo e o tempo para gerar grandes volumes de dados biológicos. Por exemplo, o *GenBank* [Benson et al., 2013], um dos BDBs mais importantes, disponibiliza aproximadamente 154 GB provenientes de mais de 167 milhões de sequências de DNA[1]. Nesse sentido, o desenvolvimento de métodos eficientes para tratar, armazenar e analisar dados biológicos é cada vez mais importante [Lesk, 2002].

[1]Fonte: `http://www.ncbi.nlm.nih.gov/genbank/statistics`

Por meio de consultas por similaridade em BDBs, pesquisas em bioinformática podem, por exemplo, determinar a função de uma nova proteína e descobrir o gene responsável por alguma doença, dentre outros [Mount, 2004]. A similaridade entre duas sequências biológicas pode ser calculada por meio do alinhamento dessas sequências [Setubal e Meidanis, 1997]. No entanto, métodos para alinhamento são computacionalmente caros e raramente aplicados diretamente a grandes BDBs. Para auxiliar esses métodos, são utilizadas estruturas de dados para indexar as cadeias de caracteres no BDB e realizar comparações eficientemente [Baets et al., 2012].

O vetor de sufixo [Manber e Myers, 1990; Gonnet et al., 1992] é uma estrutura de dados importante utilizada em problemas que envolvem cadeias de caracteres. Seu uso permite resolver de forma eficiente (tempo) e econômica (espaço) problemas relacionados à recuperação de informação, compressão de dados, bioinformática e outros [Gusfield, 1997]. O vetor de sufixo pode ser aumentado com estruturas de dados auxiliares para melhorar algoritmos de construção e consulta (*e.g.* vetores LCP e BWT) [Abouelhoda et al., 2004]. Adicionalmente, vetores de sufixo podem ser generalizados [Shi, 1996] para indexar conjuntos de cadeias de caracteres (*e.g.* BDB), o que possibilita não somente a busca por padrões nas cadeias do conjunto, mas também a comparação entre todas as cadeias indexadas [Aluru, 2005].

Devido ao enorme volume de dados envolvidos em muitas aplicações de bioinformática, muitas vezes o vetor de sufixo e o BDB são muito maiores do que a capacidade da memória interna disponível. Em geral, algoritmos desenvolvidos para memória interna executam muitos acessos aleatórios à memória interna, o que impossibilita seu uso em memória externa. Com isso, muitos trabalhos têm sido propostos para construção [Crauser e Ferragina, 1999, 2002; Dementiev et al., 2005, 2008; Bingmann et al., 2013], manipulação e consulta [Cheung, 2003; Sinha et al., 2008; Moffat et al., 2009] em vetores de sufixo em memória externa. Nessa dissertação, o enfoque refere-se à construção de vetores de sufixo generalizados em memória externa.

Uma limitação importante dos trabalhos relacionados à construção de vetores de sufixo em memória externa é que eles não são voltados à indexação de conjuntos de cadeias, isto é, eles não consideram a construção de vetores de sufixo generalizados. Essa limitação motiva o desenvolvimento desta dissertação, a qual avança no estado da arte por meio da proposta do algoritmo eGSA, o primeiro algoritmo para construção de vetores de sufixo generalizados aumentados com os vetores LCP e BWT em memória externa.

A validação do algoritmo eGSA foi realizada por meio de testes de desempenho com conjuntos de cadeias grandes (DNA) e conjuntos de cadeias pequenas (proteínas). Foram

utilizados BDBs reais de tamanhos de 0,54 GB à 8,5 GB com 24 à 105 sequências de DNA e 0,7 à 11 milhões de sequências de proteínas. Para os conjuntos de cadeias grandes, foram realizados testes comparativos com o algoritmo eSAIS [Bingmann et al., 2013], o qual representa o trabalho correlato mais eficiente disponível na literatura para construção de vetores de sufixo. Nesses experimentos a entrada do algoritmo eSAIS foi adaptada para a construção de vetores generalizados. O algoritmo eGSA obteve um tempo médio de 3,2 a 8,3 vezes menor do que o eSAIS e um consumo de memória 50% menor. Para os conjuntos de cadeias pequenas, foram realizados testes apenas com o eGSA, desde que no melhor do nosso conhecimento, não existem trabalhos correlatos que possam ser adaptados para conjuntos com muitas cadeias. Comparado com o tempo médio para conjuntos de cadeias grandes, o eGSA obteve tempos competitivos para conjuntos de cadeias pequenas. Dessa forma, os resultados dos testes demonstraram que o algoritmo proposto pode ser aplicado eficientemente para indexar tanto conjuntos de cadeias grandes quanto conjuntos de cadeias pequenas.

Resultados preliminares obtidos com o desenvolvimento desta dissertação de mestrado foram publicados no artigo intitulado: "*External Memory Generalized Suffix and LCP Arrays Construction*", o qual foi apresentado no evento científico "*Annual Symposium on Combinatorial Pattern Matching (CPM)*", classificado como B1 no Qualis Ciência da Computação.

Este trabalho de mestrado foi realizado com bolsa FAPESP (processo número 2011/15423-9). Além disso, durante o projeto foi realizado um estágio de pesquisa no exterior com os pesquisadores Steve Hoffmann [2] e Peter F. Stadler [3] da Universidade de Leipzig - Alemanha com bolsa BEPE / FAPESP (processo número 2013/01752-6). Nesse estágio foi investigado o problema de montagem de genomas *de-novo*, para o qual é apresentada uma proposta inicial do algoritmo SG2L para a construção de grafos de cadeias utilizando vetores de sufixo generalizados. Essa proposta inicial compreende, portanto, uma contribuição adicional desta dissertação.

[2] `http://hoffmann.bioinf.uni-leipzig.de/LIFE/Steve.html`
[3] `http://www.bioinf.uni-leipzig.de/~studla/`

1.1 Organização do trabalho

Esta dissertação está estruturada da seguinte forma:

- No Capítulo 2 são descritos conceitos básicos relacionados à biologia molecular, bioinformática, banco de dados biológicos e indexação de dados biológicos.
- No Capítulo 3 são descritos conceitos relacionados ao vetor de sufixo e estruturas de dados auxiliares, e são detalhados algoritmos de construção e consulta em vetores de sufixo em memória interna.
- No Capítulo 4 são resumidos trabalhos correlatos de construção de vetores de sufixo em memória externa.
- No Capítulo 5 é apresentado o algoritmo de construção eGSA.
- No Capítulo 6 são descritas as conclusões desta dissertação e listados trabalhos futuros.

A construção de vetores de sufixo generalizados possibilita o uso dessa estrutura em outros importantes problemas de bioinformática, como a montagem de genomas *de-novo* [Baets et al., 2012]. Esse problema foi investigado durante o estágio BEPE, e o resultado obtido é descrito no Apêndice A, no qual é apresentado um algoritmo preliminar, chamado de SG2L, para a construção de grafos de cadeias utilizando vetores de sufixo generalizados.

Capítulo

2

Dados Biológicos

Nesse capítulo são descritos fundamentos relacionados aos dados biológicos. Na Seção 2.1 são resumidos conceitos de biologia molecular. Em seguida, na Seção 2.2 são listados os diferentes tipos de bancos de dados biológicos. Na Seção 2.3 são resumidos conceitos de bioinformática. Na Seção 2.4 são descritos aspectos relacionados à indexação de dados biológicos e é destacado o vetor de sufixo. Por fim, na Seção 2.5 são discutidas as considerações finais.

2.1 Biologia Molecular

Todos os organismos vivos, com exceção dos vírus, são compostos por células. Organismos unicelulares como bactérias e protozoários são formados por uma única célula. Já organismos pluricelulares, mais complexos como o ser humano, são formados por trilhões de células. Os vírus são os únicos organismos acelulares conhecidos [Bolsover, 2004].

As células podem ser classificadas em dois grupos: (i) células procariontes: sem envoltório nuclear e (ii) células eucariontes: com envoltório nuclear. Células procariontes são células mais simples, sem organelas citoplasmáticas e regiões bem definidas. Já células eucariontes são complexas e possuem duas regiões bem definidas, o citoplasma com organelas e o núcleo delimitado [Calladine, 2004]. Todas as células que compõem um organismo compartilham o mesmo material genético. Nas células procariontes, o mate-

rial genético está disperso no citoplasma, enquanto que nas eucariontes está agrupado na região nuclear.

Dentro das células existem importantes macromoléculas responsáveis pela manutenção do material genético e controle dos processos celulares, os ácidos desoxirribonucleicos (DNA), ribonucleicos (RNA), e as proteínas. O DNA é responsável por carregar e transmitir a informação genética dos organismos, o RNA envolve-se com o processo de síntese (produção) de proteínas, e as proteínas são responsáveis pelo controle dos processos celulares [Selzer et al., 2008].

A relação existente entre o DNA, o RNA e as proteínas é descrita como o dogma central da biologia molecular proposto por Crick [1970]. Esse dogma pode ser resumido da seguinte forma. Um pedaço do DNA é copiado em uma sequência de RNA em um processo chamado transcrição. Em seguida, em um processo chamado de tradução, a partir do RNA forma-se uma sequência de aminoácidos, produzindo assim uma proteína [Calladine, 2004].

2.1.1 DNA e RNA

O DNA e o RNA são formados por sequências de moléculas menores, denominadas nucleotídeos. Os nucleotídeos são compostos químicos formados por uma pentose, um grupo fosfato e uma base nitrogenada, sendo que essa pode ser de cinco tipos: adenina (A), timina (T), uracila (U), citosina (C) e guanina (G). Pode-se abreviar o tipo do nucleotídeo a partir de sua base nitrogenada. Os cinco tipos de nucleotídeos são descritos na Tabela 2.1 com seus respectivos símbolos.

Tabela 2.1: Nucleotídeos

Nucleotídeo	**Símbolo**
Adenina	A
Citosina	C
Guanina	G
Timina	T
Uracila	U

A composição dos nucleotídeos de RNA é muito semelhante aos de DNA, no entanto, diferem em relação às bases nitrogenadas presentes. Os nucleotídeos de DNA podem conter as bases A, C, G e T, já os de RNA podem conter A, C, G e U, isto é, a base timina está presente apenas no DNA, e a base uracila apenas no RNA. Existe um pareamento entre as bases nitrogenadas. No DNA, T liga-se com A, e C com G, enquanto que no RNA, U liga-se com A e C com G.

O DNA de um organismo é fisicamente organizado em cromossomos. Os cromossomos são formados por grandes sequências de nucleotídeos dispostos em uma estrutura chamada dupla hélice. Essas sequências são lidas no sentido de uma extremidade chamada 5' para a outra extremidade denominada 3' [Xiong, 2006]. A Figura 2.1 ilustra a estrutura dupla hélice de um cromossomo e destaca dois nucleotídeos. As linhas verticais representam uma unidade chamada par de bases (bp), a qual é usualmente utilizada para medir o tamanho de uma sequência. Pode-se notar que um par de base é uma ligação entre dois nucleotídeos e que as duas hélices que formam o cromossomo possuem sentidos opostos.

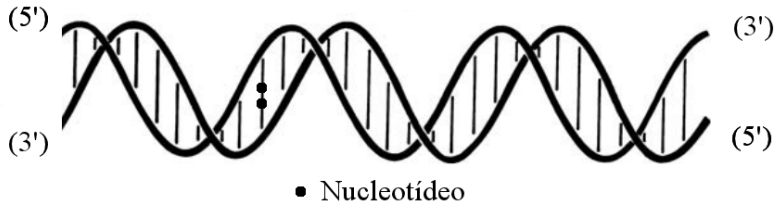

Figura 2.1: Estrutura dupla hélice dos cromossomos, adaptado de [Calladine, 2004]

O número de cromossomos pode variar de uma espécie para outra. Os cromossomos de uma mesma espécie geralmente possuem tamanhos e formas diferentes. Por exemplo, cada célula do ser humano possui 23 pares de cromossomos, sendo que um desses pares determina o sexo do indivíduo [Calladine, 2004].

O RNA é formado por uma sequência simples de nucleotídeos, e não por uma sequência de dupla hélice como o DNA. Existem três tipos de RNA, os quais participam do processo de síntese de proteínas: (i) RNA mensageiro (*mRNA*), o qual transmite a informação genética do DNA aos ribossomos; (ii) RNA transportador (*tRNA*), o qual transporta os resíduos de aminoácidos até os ribossomos para a síntese de proteínas; e (iii) RNA ribossômico (*rRNA*), o qual coordena a interação entre *mRNA*, *tRNA* e proteínas durante o processo de síntese de proteínas.

O RNA possui um papel importante no processo de síntese de proteínas. Além disso, alguns vírus possuem apenas RNA em seu material genético.

2.1.2 Proteínas

As proteínas são substâncias essenciais da estrutura das células, já que elas participam de todos os processos bioquímicos dentro e fora da célula. Proteínas são macromoléculas orgânicas formadas por sequências de aminoácidos, unidas por meio de ligações peptídicas [Lesk, 2002].

Os aminoácidos são pequenos blocos nitrogenados resultados da tradução de agrupamentos funcionais de sequências de nucleotídeos. Cada trinca de nucleotídeos é chamada de códon. Um códon possui a informação biológica necessária para codificar um aminoácido [Calladine, 2004]. Os 20 aminoácidos existentes são descritos na Tabela 2.2 com suas respectivas abreviações e símbolos.

Tabela 2.2: Aminoácidos

Aminoácidos	**Abreviação**	**Símbolo**
Alanina	Ala	A
Cisteina	Cys	C
Aspartato	Asp	D
Glutamato	Glu	E
Fenilalanina	Phe	F
Glicina	Gly	G
Histidina	His	H
Iaoleucina	Ile	I
Lisina	Lys	K
Leucina	Leu	L
Metionina	Met	M
Asparagina	Asn	N
Prolina	Pro	P
Glutamina	Gin	Q
Arginina	Arg	R
Serina	Ser	S
Treonina	Thr	T
Valina	Val	V
Triptofano	Trp	W
Treonina	Tyr	Y

Existem códons com funções especiais para indicar onde uma codificação deve iniciar e parar. Por exemplo, o códon ATG codifica o aminoácido metionina, chamado de *START-Códon*, o qual indica que uma região é o ponto de partida para codificação de proteína. Em contrapartida, os códons TAA, TAG e TGA são conhecidos como *STOP-Códons*. Esses códons não codificam nenhum aminoácido, mas indicam que a leitura deve parar naquele ponto [Bolsover, 2004].

Existem partes do DNA que não são utilizadas para codificar aminoácidos. Essas regiões são chamadas de RNA não-codificadores, e suas origens e funções ainda não foram totalmente desvendadas pela biologia molecular [Xiong, 2006]. Em contrapartida, as regiões funcionais do DNA, conhecidas como éxon, são responsáveis por codificar sequên-

cias de aminoácidos que formam proteínas. Na Figura 2.2 são representadas três proteínas codificadas a partir de genes distintos.

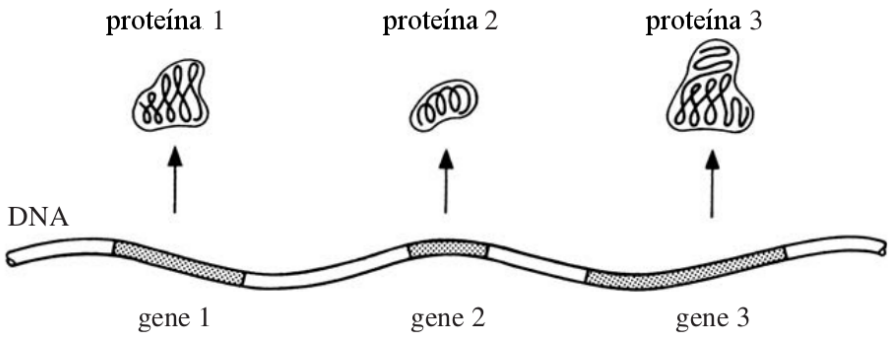

Figura 2.2: Genes no DNA que codificam proteínas, adaptado de [Calladine, 2004]

As funções biológicas desempenhadas por uma proteína dependem além da sua sequência de aminoácidos também de sua estrutura espacial. A estrutura espacial de uma proteína varia de acordo com sua complexidade, sendo que existem quatro níveis de organização estrutural [Bolsover, 2004]:

1. **Estrutura primária:** é o nível estrutural mais simples. É dada pela sequência de aminoácidos que forma a proteína.

2. **Estrutura secundária:** é dada pelo arranjo espacial dos aminoácidos próximos entre si na estrutura primária.

3. **Estrutura terciária:** é caracterizada pelas interações de longa distância entre aminoácidos.

4. **Estrutura quaternária:** algumas proteínas são formadas por mais de uma cadeia polipeptídica. A estrutura quaternária é a estrutura gerada a partir da interação dessas cadeias.

Encontrar a estrutura espacial de uma proteína é um experimento caro e, às vezes, impraticável. Com isso, diversos estudos tentam prever funções proteicas a partir de suas estruturas primárias, já que essas estruturas são mais fáceis de serem obtidas e analisadas experimentalmente [Korf et al., 2003].

2.1.3 Genoma

O genoma de um organismo corresponde a toda informação hereditária presente em seu DNA (no caso de alguns vírus, no RNA). Essa informação inclui os genes e as regiões não codificadoras [Calladine, 2004]. Organismos simples, como o da bactéria Rhodococcus, possuem genomas pequenos. Esse organismo é formado por 9.702.737 bp. Por outro lado, organismos mais complexos tendem a possuir genomas bem mais extensos. O genoma do ser humano, por exemplo, é formado por aproximadamente 3 Gbp.

O genoma de um organismo é obtido por meio do sequenciamento de seu DNA [Setubal e Meidanis, 1997]. As máquinas de sequenciamento (sequenciadores) possuem a limitação de não realizar a leitura do DNA completo de uma única vez, e as cadeias obtidas têm tamanho no máximo de 1.000 bp. Então, durante o sequenciamento as moléculas de DNA são duplicadas e particionadas em pequenos pedaços, ilustrados na Figura 2.3, os quais são lidos em cadeias curtas (chamados de *reads*). O tamanho das *reads* e a quantidade de cópias geradas (cobertura) dependem da tecnologia de sequenciamento utilizada.

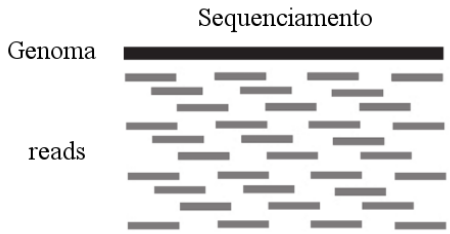

Figura 2.3: Sequenciamento de DNA, adaptado de [Barh et al., 2013]

O processo de reconstrução da sequência original é conhecido como montagem de genomas. Esse é um problema combinatório e computacionalmente caro. Existem diferentes propostas para resolver o problema de montagem. Essas propostas, em geral, seguem uma das seguintes estratégias [Scheibye-Alsing et al., 2009]: (i) montagem por referência ou (ii) montagem *de-novo*.

A montagem por referência pode ser realizada quando existe um genoma já sequenciado (referência) do mesmo organismo ou de uma espécie biologicamente próxima. A montagem é realizada por meio do alinhamentos das *reads* (mapeamento) com o genoma referência [Schbath et al., 2012]. Esse método é mais rápido quando comparado à montagem *de-novo*. Entretanto, depende do genoma referência e da qualidade desses dados.

A montagem *de-novo*, por sua vez, utiliza apenas a redundância nas *reads* sequenciadas [Compeau et al., 2011]. A montagem é feita por meio da identificação de sobreposições

entre as *reads*, e da geração de cadeias maiores, chamadas de *contigs*, por meio da concatenação das *reads* sobrepostas, e posterior agrupamento desses *contigs* em *scaffolds*, de forma a ordená-los corretamente no genoma.

Na prática, ferramentas de montagem de genomas (*e.g.* Celera [Myers et al., 2000], Velvet [Zerbino e Birney, 2008] e SGA [Simpson e Durbin, 2012]) consideraram heurísticas para tratar em um preprocessamento questões do processo de sequenciamento, como erros de leitura, baixa cobertura, quimeras e regiões de repetições no genoma, entre outros [Scheibye-Alsing et al., 2009].

2.2 Bancos de Dados Biológicos

Bancos de dados biológicos (BDBs) armazenam informações referentes aos seres vivos, como sequências de DNA, de RNA e de proteínas, mapas genéticos, anotações gênicas, estruturas tridimensionais de proteínas e outras informações relacionadas. BDBs podem ser divididos em três categorias a partir do tipo e do conteúdo dos dados neles armazenados [Xiong, 2006]:

1. **BDBs primários**: armazenam informações biológicas originais, isto é, sequências de DNA, de RNA e de proteínas, acompanhadas de informações relacionadas.
2. **BDBs secundários**: armazenam resultados de análises feitas a partir de dados primários.
3. **BDBs especializados**: armazenam dados referentes a um interesse particular.

Existem diversos BDBs públicos que disponibilizam o acesso aos seus dados na Internet. Anualmente, o periódico *NAR* (*Nucleic Acids Research*) dedica uma edição inteira (primeira edição em janeiro) com todos os BDBs atualmente disponíveis, organizados em uma tabela com o nome e respectivo endereço URL [Selzer et al., 2008].

O *GenBank*, o *EMBL* e o *DDBJ* são importantes BDBs primários e representam o conhecimento atual sobre dados biológicos primários e são fontes para muitos outros BDBs. Por meio de um projeto de colaboração, são trocadas diariamente informações, assegurando uma coleção uniforme e consistente entre esses bancos [Xiong, 2006].

O *SWISS-Prot* e o *PIR* (*Protein Information Resources*) são exemplos de BDBs secundários que armazenam informações referentes às proteínas, incluindo anotações de funcionalidades, estruturas tridimensionais e literatura associada. Enquanto que o *Flybase*,

o *HIV sequence database*, e o *RDP* (*Ribossomal Database Project*) são BDBs especializados para um organismo ou tipo de dado particular [Xiong, 2006]. Na Tabela 2.3 são relacionados os BDBs citados com seus respectivos endereços URL.

Tabela 2.3: Bancos de dados biológicos citados

BDB	URL
GenBank	`http://www.ncbi.nlm.nih.gov/genbank/`
EMBL	`http://www.ebi.ac.uk/embl/`
DDJB	`http://www.ddbj.nig.ac.jp/`
SWISS-Prot	`http://ca.expasy.org/sprot/`
PIR	`http://pir.georgetown.edu/`
Flybase	`http://flybase.org/`
HIV sequence database	`http://www.hiv.lanl.gov/content/index`
RDP	`http://rdp.cme.msu.edu/`

A maioria dos BDBs disponibiliza, além de seus dados, um conjunto de ferramentas para consulta, como é o caso do *GenBank*, que oferece diferentes versões da ferramenta BLAST[1] [Altschul et al., 1990] para a execução de pesquisas rápidas por similaridade sobre as sequências armazenadas no *GenBank*.

Nesta dissertação são utilizados sequências de DNA provenientes de BDBs primários. Essas sequências são armazenadas nos BDBs primários como cadeias de caracteres, onde cada um dos caracteres representa um dos nucleotídeos A, C, G, T presentes na sequência. Nessas cadeias pode ocorrer o caractere N, que indica que o valor do nucleotídeo naquela posição da sequência é desconhecido. Em muitas aplicações, o caractere N é removido da cadeia [Barsky et al., 2008].

Mais detalhadamente, BDBs primários utilizam arquivos de texto simples (*raw file*[2]) para armazenar essas cadeias. O tipo de arquivo "fasta" é um dos formatos mais populares utilizado para armazenar dados biológicos primários.

Um arquivo no formato fasta, em geral, possui na sua primeira linha um cabeçalho indicado pelo símbolo ">". Esse cabeçalho contém o nome da sequência armazenada e algumas informações extras. A sequência de interesse é armazena nas cadeias de caracteres a partir da segunda linha do arquivo. Na Figura 2.4 é ilustrado um exemplo de arquivo do tipo fasta, o qual se refere ao início da sequência de DNA do cromossomo 4 do genoma Humano.

[1] disponível em: `http://blast.ncbi.nlm.nih.gov/`
[2] *raw file*: termo utilizado para denotar arquivos do tipo texto sem formatação.

```
>ref|NT_006316.16|:1-23002714 Homo sapiens chromosome 4 genomic
contig, GRCh37 reference primary assembly
GCTGGTTCTGAACTGTGTGAAGTATATTGAAAAAAACTAACTGATACAAAAATATAATTT
TATGATAGTTTTCTGGATGTCCCAATATAAACGATGTCAATTCTGCGACATGCTACAGTC
ATCCACGAAAGTAACCCGAATACCGACAAAAGAAGAGGAACGCCAACTTTGGATAGACGC
TCTAGGGGCTGATTTTGGTCGGAAAATAGTCGGGAAAAAATAGAGGACATTACAGATGAG
GATGAGGATGAAGATAGAAATTTGCCGACAACTTCGTCATGCCGCTGATTTTTTTGATGT
TCTACGCTTAAATTTTCAGCGAACGAACTATTTTTTATATTTTGATTGTTTTTAAATAAT
```

Figura 2.4: Exemplo de arquivo fasta (sequência de DNA)

2.3 Bioinformática

A *bioinformática* é a área da ciência que utiliza métodos da biologia, da matemática e da computação para solucionar questões relevantes a partir de dados biológicos [Ouellette e Baxevanis, 2005]. O objetivo da bioinformática é pesquisar a vida celular e o funcionamento de seus mecanismos em nível molecular. Essas pesquisas possuem grande impacto em outras áreas como a biotecnologia e as ciências biomédicas. Algumas aplicações envolvem, por exemplo, o desenvolvimento de novos remédios, análise forense, e bioengenharia agrícola.

A bioinformática difere de um campo relacionado conhecido como *biologia computacional*. A bioinformática limita-se à análise dos dados em nível molecular e é frequentemente denominada por *biologia molecular computacional*. Em contraste, a biologia computacional engloba todas as áreas da biologia que utilizam a computação. Por exemplo, modelagem matemática de ecossistemas, dinâmica de populações, estudos de comportamentos, e construção filogenética usando fósseis [Xiong, 2006].

Uma das principais áreas de interesse dentro da bioinformática é a identificação de similaridade entre dados biológicos, a qual possibilita indicar relações de homologia [Korf et al., 2003]. Diversos avanços na medicina têm sido obtidos por meio da pesquisa por similaridade em BDBs, como a determinação dos genes relacionados à cura de úlceras venosas e a identificação de padrões regulatórios, tais como aqueles representando sítios de fatores de transcrição e sítios de *splicing* de RNA mensageiro [Mount, 2004].

A similaridade entre duas sequências pode ser calculada por meio do alinhamento dessas sequências [Setubal e Meidanis, 1997]. Alinhamentos globais comparam sequências em toda sua extensão, enquanto que o alinhamento local busca encontrar regiões nas quais as sequências possuem alto grau de similaridade [Korf et al., 2003]. No entanto, métodos para calcular o alinhamento são computacionalmente caros, e raramente aplicados diretamente a grandes BDBs [Aluru, 2005].

Por exemplo, considere a tarefa de encontrar sequências similares a uma consulta em um BDB composto por bilhões de sequências. É computacionalmente caro fazer todos os alinhamentos entre a consulta e todas as sequências no BDB. Uma importante observação é que bons alinhamentos possuem regiões de correspondências exatas, e encontrar essas regiões possui custo computacional menor. Dessa forma, casamentos exatos podem ser utilizados como filtros para eliminar grande número de pares que não retornariam um bom alinhamento, realizando o alinhamento somente nos pares que têm correspondência maior do que um determinado limiar [Aluru, 2005]. Essa estratégia é utilizada nas principais ferramentas de alinhamento em BDBs, como a família BLAST [Altschul et al., 1990, 1997] e FASTA [Pearson e Lipman, 1988].

Em detalhes, a primeira etapa dessas ferramentas de alinhamento corresponde à etapa de casamento exato, na qual são realizadas buscas exatas por pequenas sub-cadeias da consulta no BDB. Os resultados, em seguida, passam por mais etapas para encontrar os alinhamentos desejados. Por exemplo, a etapa de casamento exato na ferramenta BLAST consome aproximadamente 80% do tempo de processamento completo [Sinha et al., 2008]. Dessa forma, o desenvolvimento de métodos para casamento exato desempenha um papel importante em pesquisas por similaridade em sequências biológicas.

2.4 Indexação de Dados Biológicos

A quantidade enorme de dados biológicos faz com que o desenvolvimento de métodos eficientes para tratar e analisar esses dados seja um problema fundamental em bioinformática [Lesk, 2002]. Em geral, pode-se utilizar estruturas de dados para indexar BDBs e auxiliar na recuperação dos dados presentes nesses BDBs. No caso de consultas por similaridades em sequências, podem ser utilizadas estruturas de dados para as cadeias de caracteres. Na literatura, dentre as principais estruturas de dados para cadeias destacam-se a árvore de sufixo e o vetor de sufixo [Gusfield, 1997].

A árvore de sufixo (*suffix tree*), proposta por Weiner [1973], armazena todos os sufixos de uma cadeia de caracteres em uma árvore enraizada na qual cada folha representa um sufixo. Casamentos exatos utilizando árvores de sufixo podem ser respondidos em tempo linear ao tamanho da consulta. Além disso, árvores de sufixo podem ser utilizadas para encontrar a sub-cadeia comum mais longa entre duas ou mais cadeias, detectar palíndromos, repetições, entre outras [Gusfield, 1997].

No entanto, embora a árvore de sufixo apresente-se como uma solução eficiente para muitos problemas, o volume de memória que ela ocupa é muito grande, dificultando sua aplicação para grandes cadeias. Por exemplo, uma árvore de sufixo para uma sequência de DNA de 700 Mbp ocupa aproximadamente 40 GB de espaço em memória [Aluru, 2005].

Manber e Myers [1990, 1993]; Gonnet et al. [1992] propuseram o vetor de sufixo (*suffix array*) como uma alternativa econômica em termos de espaço às árvores de sufixo. Um vetor de sufixo representa todos os sufixos de uma cadeia de caracteres ordenados lexicograficamente. [Abouelhoda et al., 2004] mostraram que qualquer problema que pode ser resolvido com árvores de sufixo é resolvível com a mesma complexidade assintótica utilizando vetores de sufixo com o auxílio de outras estruturas de dados (*e.g.* vetores LCP e BWT). Com isso, vetores de sufixo tornaram-se a estrutura de indexação utilizada em muitos, se não todos, problemas de processamento de cadeias de caracteres nos quais podem ser utilizados a árvore de sufixo [Puglisi et al., 2007].

Por fim, vetores de sufixo podem ser generalizados [Shi, 1996] para indexar conjuntos de cadeias de caracteres (*e.g.* BDB), o que possibilita não somente a busca por padrões nas cadeias do conjunto, mas também a comparação entre todas as cadeias indexadas no vetor de sufixo generalizado [Aluru, 2005].

2.5 Considerações Finais

Nesse capítulo foram resumidos fundamentos relacionados aos dados biológicos. Na Seção 2.1 foram descritos conceitos de biologia molecular. Em seguida, na Seção 2.2 foram listados os diferentes tipos de bancos de dados biológicos. Na Seção 2.3 foram resumidos os objetivos da bioinformática. Por fim, na Seção 2.4 foi discutida a importância da indexação de dados biológicos para o desenvolvimento de métodos eficientes em bioinformática, e destacado o uso de vetores de sufixo nesse problema.

Nesta dissertação é investigado o uso de vetores de sufixo na indexação de dados biológicos referentes às sequências de DNA e de proteínas. No Capítulo 3 são descritos os vetores de sufixo em conjunto com algoritmos relacionados. Em seguida, no Capítulo 4 são investigados trabalhos relacionados à construção de vetores de sufixo em memória externa. Por fim, no Capítulo 5 é proposto o algoritmo desenvolvido nesta dissertação, o qual consiste no primeiro algoritmo na literatura voltado à construção de vetores de sufixo generalizados em memória externa.

Capítulo

3

Vetores de Sufixo

Nesse capítulo são descritos conceitos e algoritmos relacionados aos vetores de sufixo. Na Seção 3.1 é definida a terminologia utilizada. Em seguida, na Seção 3.2 são descritos o vetor de sufixo em conjunto com estruturas de dados auxiliares. Nas Seções 3.3 e 3.4 são resumidos algoritmos de construção e consulta para vetores de sufixo em memória interna, respectivamente. Por fim, na Seção 3.5 são destacadas considerações finais.

3.1 Terminologia e Definições

Seja $\Sigma = \{\alpha_1, \alpha_2, \ldots, \alpha_\sigma\}$ um alfabeto ordenado de σ caracteres e \$ um caractere terminal, tal que $\$ \notin \Sigma$, e $\$ < \alpha_i, \forall \alpha_i \in \Sigma$.

Definição 3.1 *Σ^* é o conjunto de todas as cadeias finitas geradas a partir de Σ, tal que $S = S[1]S[2]\ldots S[n]$ é uma cadeia em Σ^* com $|S| = n$ caracteres, e $S[i] \in \Sigma$ $(1 \leq i \leq n)$.*

Definição 3.2 *$\Sigma^\$$ é o conjunto das cadeias de Σ^* concatenadas com o caractere terminal \$, tal que $S\$ \in \Sigma^\$ \iff S \in \Sigma^*$.*

Nesse capítulo, são consideradas cadeias de caracteres sobre o alfabeto $\Sigma = \{A, C, G, T\}$ e $\$ < A < C < G < T$.

Seja S uma cadeia em $\Sigma^{\$}$, e $|S| = n$. Uma sub-cadeia de S é denotada por $S[i, j] = S[i] \ldots S[j]$, para $1 \leq i \leq j \leq n$. Em particular a sub-cadeia $S[1, i]$ é um prefixo de S de tamanho i e $S[i, n]$ é um sufixo de S de tamanho $n - i + 1$. Um sufixo que começa com $\alpha \in \Sigma$ é denotado por α-$sufixo$.

Sejam S_a e S_b duas cadeias em $\Sigma^{\$}$, tais que $|S_a| = n_a$ e $|S_b| = n_b$.

Definição 3.3 *A cadeia S_a é menor (lexicograficamente) que S_b, denotado por $S_a < S_b$, se existir um índice j, tal que: $S_a[1, j-1] = S_b[1, j-1]$, e $S_a[j] < S_b[j]$, com $1 < j \leq min\{n_a, n_b\}$.*

Definição 3.4 *Caso S_a e S_b sejam iguais, isto é, $j = n_a = n_b$ então $S_a < S_b$ se $a < b$.*

Sejam as cadeias $S_a = GATAGA\$$ e $S_b = GAA\$$ em $\Sigma^{\$}$.

Por exemplo, a cadeia S_b é menor que a cadeia S_a pois $S_a[1, 2] = S_b[1, 2] = GA$ e $S_b[3] < S_a[3]$ $(A < T)$.

Definição 3.5 *O prefixo comum mais longo entre S_a e S_b é a sub-cadeia $S_a[1, k]$, tal que: $S_a[1, k] = S_b[1, k]$ e $S_a[k+1] \neq S_b[k+1]$.*

Definição 3.6 *O tamanho do prefixo comum mais longo entre S_a e S_b é denotado por $lcp(S_a, S_b)$.*

Por exemplo, o prefixo comum mais longo entre S_a e S_b é a sub-cadeia $S_a[1, 2] = S_b[1, 2] = GA$. Com isso, $lcp(S_a, S_b) = 2$.

Definição 3.7 *A concatenação de duas cadeia S_a e S_b é representada por $S_a \cdot S_b$.*

Por exemplo, a concatenação de S_a com S_b é igual à $S_a \cdot S_b = GATAGA\$GAA\$$.

Seja $P \in \Sigma^*$ uma cadeia de tamanho m, denominada *padrão*, a ser procurada na cadeia $T \in \Sigma^{\$}$ de tamanho n, denominada *texto*.

Definição 3.8 *A busca por um padrão ou casamento (exato) retorna todas as occ ocorrências, se existir, do padrão P no texto T.*

Por exemplo, seja o padrão $P = ACA$ e o texto $T = CGACAGGACACAG\$$, o padrão P ocorre em T nas posições 3, 8 e 10. Note que duas ou mais ocorrências de P podem se sobrepor, como é o caso das ocorrências nas posições 8 e 10.

A busca por um padrão P pode ser feita em tempo $O(n + occ)$ por meio de uma busca sequencial em T [Knuth et al., 1977].

3.2 Estruturas de Dados

O vetor de sufixo (*suffix array*), proposto por Manber e Myers [1990, 1993]; Gonnet et al. [1992], é um vetor de inteiros que fornece a ordem lexicográfica de todos os sufixos de um texto T.

Definição 3.9 *Um vetor de sufixo, denotado por SA, para* $T \in \Sigma^{\$}$, $|T| = n$, *é um vetor de* n *inteiros* $SA = [j_1, j_2, \ldots, j_n]$, *de forma que:* $T[j_1, n] < \cdots < T[j_n, n]$.

Uma partição de SA que contém apenas α-$sufixos$ é denotada por α-*bucket*. A função $pos(SA, T[i, n])$ retorna a posição do sufixo $T[i, n]$ em SA e a função $suff(i)$ retorna o sufixo na posição i de SA.

Definição 3.10 $pos(SA, T[i, n]) = j \Leftrightarrow SA[j] = i$.

Definição 3.11 $suff(i) = T[SA[i], n]$.

Por exemplo, seja o texto $T = GATAGA\$$, o vetor de sufixo para T é ilustrado na Figura 3.1. A coluna $suff(i)$ é apenas ilustrativa, isto é, $suff(i)$ não é armazenada em SA. Note que, sufixos em posições consecutivas em SA compartilham um mesmo prefixo (que pode ser nulo). Por exemplo, os sufixos $T[5, 7]$ e $T[1, 7]$ estão nas posições 5 e 6, respectivamente, e compartilham o prefixo "GA".

i	$SA[i]$	$suff(i)$
1	7	\$
2	6	A\$
3	4	AGA\$
4	2	ATAGA\$
5	5	GA\$
6	1	GATAGA\$
7	3	TAGA\$

Figura 3.1: Vetor de sufixo SA para $T = GATAGA\$$

O vetor de sufixo pode ser construído em tempo $O(n)$ (discutido na Seção 3.3). Uma vez construído, a busca por um padrão P em T utilizando SA pode ser resolvida por meio de uma busca binária em tempo $O(m \log n + occ)$ (discutido na Seção 3.4). Estruturas de dados auxiliares podem ser utilizadas para melhorar algoritmos de construção e consulta

em vetores de sufixo [Grossi, 2011], dentre as quais destacam-se o vetor de prefixo comum mais longo e a transformada de *Burrows-Wheeler*.

O vetor de *lcp*, denotado por *LCP*, armazena o *lcp* entre dois sufixos em posições consecutivas de *SA*, e pode ser obtido em tempo $O(n)$ a partir de *SA* [Kasai et al., 2001; Gog e Ohlebusch, 2011] ou durante a construção de *SA* [Fischer, 2011].

Definição 3.12 $LCP[1] = 0$ *e* $LCP[i] = lcp(T[SA[i-1], n], T[SA[i], n])$.

Uma importante propriedade do vetor *LCP* é que o *lcp* entre dois sufixos não consecutivos nas posições x e y de *SA* corresponde ao valor mínimo (rmq) em *LCP* entre as posições $x+1$ e y, isto é, $lcp(T[SA[x], n], T[SA[y], n]) = min_{x<k\leq y}\{LCP[k]\}$.

Definição 3.13 $rmq_{LCP}(x, y) = min_{x<k\leq y}\{LCP[k]\}$, *para* $x < y$.

A transformada de *Burrows-Wheeler* corresponde a uma permutação de T que busca agrupar caracteres iguais [Burrows e Wheeler, 1994]. O vetor *BWT* representa essa permutação e pode ser obtido em tempo $O(n)$ a partir de *SA*.

Definição 3.14 $BWT[i] = T[SA[i]-1]$ *se* $SA[i] \neq 1$ *ou* $BWT[i] = \$$.

O vetor de sufixo em conjunto com outras estruturas de dados é chamado de vetor de sufixo aumentado (*enhanced*) [Abouelhoda et al., 2004] e é denotado por *ESA*.

Por exemplo, sejam os textos $T_1 = GATAGA\$$ e $T_2 = TAGAGA\$$. Os vetores de sufixo aumentados com os vetores *LCP* e *BWT* para T_1 e T_2 são ilustrados na Figura 3.2.

i	$ESA_1[i]$			$suff(i)$
	SA	*LCP*	*BWT*	
1	7	0	A	$
2	6	0	G	A$
3	4	1	T	AGA$
4	2	1	G	ATAGA$
5	5	0	A	GA$
6	1	2	$	GATAGA$
7	3	0	A	TAGA$

(a) ESA_1 para $T_1 = GATAGA\$$

i	$ESA_2[i]$			$suff(i)$
	SA	*LCP*	*BWT*	
1	7	0	A	$
2	6	0	G	A$
3	4	1	G	AGA$
4	2	3	T	AGAGA$
5	5	0	A	GA$
6	3	2	A	GAGA$
7	1	0	$	TAGAGA$

(b) ESA_2 para $T_2 = TAGAGA\$$

Figura 3.2: Vetores de sufixo aumentados (*ESA*) para T_1 (a) e T_2 (b).

O vetor de sufixo pode ser generalizado para indexar conjuntos de cadeias de caracteres [Shi, 1996]. Um vetor de sufixo generalizado para um conjunto de cadeias $\mathcal{T}$ fornece a ordem de todos os sufixos de todas as cadeias $\mathcal{T}$.

Definição 3.15 *Um vetor de sufixo generalizado, denotado por GSA, para um conjunto de k cadeias* $\mathcal{T} = \{T_1, \ldots, T_k\}$, $|T_i| = n_i$, $\mathcal{T} \in \Sigma^{\$}$, *é um vetor de pares de inteiros* (a, b) $GSA = [(a_1, b_1), (a_2, b_2), \ldots, (a_N, b_N)]$, *de forma que:* $T_{a_1}[b_1, n_{a_1}] < \cdots < T_{a_N}[b_N, n_{a_N}]$.

O tamanho de *GSA* é proporcional ao número de sufixos em $\mathcal{T}$, isto é, $N = \sum_{i=1}^{k} |T_i|$. A função *pos* e os vetores *LCP* e *BWT* também podem ser generalizados. O vetor de sufixo generalizado aumentado com outras estruturas é denotado por *GESA*.

Por exemplo, sejam os textos $T_1 = GATAGA\$$ e $T_2 = TAGAGA\$$. O vetor de sufixo generalizado aumentado com *LCP* e *BWT* para $\mathcal{T} = \{T_1, T_2\}$ é ilustrado na Figura 3.3. Note que podem haver sufixos iguais de diferentes cadeias em *GSA*, isso não acontecia em *SA*. Nesse caso o sufixo da cadeia T_i com menor índice i é o menor. Por exemplo os sufixos *suff*(3) e *suff*(4). O tamanho de *GESA* é igual à $N = |T_1| + |T_2| = 7 + 7 = 14$.

i	*GESA*[i]			*suff*(i)
	GSA	*LCP*	*BWT*	
1	(1,7)	0	A	$
2	(2,7)	1	A	$
3	(1,6)	0	G	A$
4	(2,6)	1	G	A$
5	(1,4)	1	T	AGA$
6	(2,4)	3	G	AGA$
7	(2,2)	3	T	AGAGA$
8	(1,2)	1	G	ATAGA$
9	(1,5)	0	A	GA$
10	(2,5)	2	A	GA$
11	(2,3)	2	A	GAGA$
12	(1,1)	2	$	GATAGA$
13	(1,3)	0	A	**TAGA**$
14	(2,1)	4	$	**TAGA**GA$

Figura 3.3: Vetor de sufixo generalizado aumentado (*GESA*) para $\mathcal{T} = \{T_1, T_2\}$.

Com respeito ao tamanho das estruturas de dados, considerando $|\Sigma| < 255$, $|T_i| < 2^{32}$ e $k < 2^{32}$, a quantidade de memória necessária para armazenar *SA* e *LCP* é $4 \times n$ *bytes* cada e para armazenar *BWT* é $1 \times n$ *bytes*. No caso das estruturas generalizadas, *GSA* ocupa $2 \times 4 \times N$ *bytes*, *LCP* $4 \times N$ *bytes* e *BWT* ocupa $1 \times N$ *bytes* [Aluru, 2005]. Dessa

forma, a complexidade assimptótica de espaço de todas as estruturas é linear em relação ao tamanho das cadeias.

3.3 Etapa de Construção

A etapa de construção de um vetor de sufixo consiste em ordenar todos os sufixos do texto T. Valores de *lcp* podem indicar o nível de dificuldade dessa ordenação [Sadakane, 1998]. Em geral, são utilizados os valores de *lcp*-médio e *lcp*-máximo, os quais podem ser obtidos do vetor *LCP* (Definições 3.16 e 3.17). Esses valores variam de acordo com o domínio dos dados, e indicam aproximadamente o número de comparações que devem ser feitas para ordenar dois sufixos, isto é, a dificuldade da ordenação [Adjeroh et al., 2008].

Definição 3.16 *lcp-médio* $= \dfrac{1}{(n-1)} \sum_{i=1}^{n} LCP[i]$

Definição 3.17 *lcp-máximo* $= \max_{1<i\leq n} \{LCP[i]\}$

A ordenação é um problema clássico em *computação*. Em uma abordagem simples, pode-se considerar os sufixos como elementos de um vetor. Então, a ordenação pode ser realizada por meio de algum algoritmo de ordenação padrão, como o *quick-sort* ou *radix-sort* utilizando comparações caractere-por-caractere [Adjeroh et al., 2008]. No entanto, esses algoritmos não consideram características importantes existentes entre os sufixos de uma mesma cadeia, as quais podem ser exploradas para uma ordenação mais eficiente. Por exemplo, sufixos de uma mesma cadeia possuem sub-cadeias intercaladas, compartilham sub-cadeias em comum e não possuem espaço em branco [Aluru, 2005].

Na literatura, existem diversos algoritmos para construção de vetores de sufixo em memória interna. Manber e Myers [1990, 1993] propuseram, juntamente com o conceito de vetor de sufixo, um algoritmo para a construção em tempo $O(n \log n)$. Desde então, diversos outros algoritmos têm sido propostos. Em 2003, Kärkkäinen et al. [2003], Ko e Aluru [2003] e Kim et al. [2003a] propuseram simultaneamente os primeiros algoritmos de construção em tempo $O(n)$. Itoh e Tanaka [1999], Seward [2000] e Manzini e Ferragina [2004] propuseram algoritmos que utilizam pouca memória auxiliar durante a construção e Maniscalco e Puglisi [2006], Larsson e Sadakane [2007] propuseram algoritmos não lineares, porém rápidas na prática. Por fim, [Nong et al., 2009, 2011] propuseram os primeiros algoritmos de construção em tempo $O(n)$, rápidos na prática, e que utilizam pouca memória auxiliar. Os trabalhos de Puglisi et al. [2007]; Dhaliwal et al. [2012]

apresentam revisões e comparam muitos desses algoritmos na prática. Atualmente o algoritmo proposto por Nong et al. [2011] apresenta-se como o algoritmo mais eficiente para construção de vetores de sufixo em memória interna.

Com respeito à construção de vetores de sufixo generalizados em memória interna para conjuntos de k cadeias $\mathcal{T} = \{T_1, \ldots, T_k\}$, Shi [1996] propôs um algoritmo de construção em memória interna em tempo $O(N \log n_{max})$, onde n_{max} é o tamanho da maior cadeia do conjunto $\mathcal{T}$ e $N = \sum_{i=1}^{k} |T_i|$. Além disso, é possível construir o vetor generalizado utilizando um algoritmo para construção de vetores de sufixo. Para isso, é necessário adaptar a entrada do algoritmo concatenando todas as cadeias $T_i \in \mathcal{T}$ utilizando diferentes símbolos terminais $\$_i$ em uma única cadeia $T = T_1\$_1 \ldots T_k\$_k$, de forma que $\$_i < \$_j$ se $i < j$. Entretanto, devido ao número de possíveis símbolos $\$_i$ ser limitado pelo tamanho da variável utilizada menos o tamanho do alfabeto da cadeia, essa estratégia limita o número de cadeias em $\mathcal{T}$ [Bauer et al., 2012]. No caso de sequências de DNA, usando variáveis de 1 *byte* para cada caractere, k é limitado por $256 - |\Sigma| = 252$.

Dentre os trabalhos existentes na literatura para construção de vetores de sufixo, destacam-se o algoritmo MM [Manber e Myers, 1990, 1993] e o algoritmo DC3 [Kärkkäinen et al., 2003, 2006]. Esses algoritmos são base para trabalhos voltados à construção de vetores de sufixo em memória externa, os quais são descritos no Capítulo 4, e, portanto, serão descritos nas Seções 3.3.1 e 3.3.2, respectivamente.

3.3.1 Algoritmo MM

O algoritmo MM, proposto por Manber e Myers [1990, 1993], realiza a construção de *SA*, para um texto de tamanho n, em no máximo $\lceil \log n \rceil$ passos. A cada passo, são realizados no máximo n comparações. Dessa forma, o algoritmo MM possui complexidade de tempo $O(n \log n)$. O algoritmo MM utiliza as seguintes definições.

Sejam S_a e S_b duas cadeias em $\Sigma^\$$, tais que $|S_a| = n_a$ e $|S_b| = n_b$.

Definição 3.18 *A cadeia S_a é h-menor que S_b, denotado por $S_a <_h S_b$, se o prefixo $S_a[1, h]$ é menor que o prefixo $S_b[1, h]$.*

Definição 3.19 *Um vetor de sufixo parcial de ordem-h, denotado por SA^h, para $T \in \Sigma^\$$, $|T| = n$, é um vetor de inteiros $SA^h = [j_1, j_2, \ldots, j_n]$, de forma que: $T[j_1, n] <_h T[j_2, n] <_h \cdots <_h T[j_n, n]$.*

Definição 3.20 *Um bucket-h é um intervalo de SA^h que contém todos os sufixos que possuem os h primeiros caracteres iguais.*

Por exemplo, seja $T = GATAGA\$$, o vetor de sufixo parcial SA^1 é ilustrado na Figura 3.4(a). Os *buckets-1* são ilustrados pelas linhas horizontais na figura e os h caracteres são destacados em negrito. É possível observar que todos os sufixos estão parcialmente ordenados em SA^1 (apenas pelo primeiro caractere).

O algoritmo MM é resumido no Algoritmo 1. No passo inicial (linha 2) todos os sufixos de T são ordenados de acordo com seus primeiros caracteres. Esses sufixos são alocados em *buckets-1*. Ao final desse passo obtém-se o vetor parcial SA^1. Essa ordenação pode ser realizada em tempo linear utilizando algum algoritmo padrão de ordenação, por exemplo o *bucket sort* [Adjeroh et al., 2008].

Algoritmo 1: Algoritmo MM

Entrada: texto T, $|T| = n$
Saída: vetor de sufixo SA

1 **início**
2 $SA^1 \leftarrow$ sufixos ordenados de acordo com seus primeiros caracteres;
3 **para** $i \leftarrow 1$ *até* $\lceil \log n \rceil$ **faça**
4 $h \leftarrow 2^i$;
5 **para** $j \leftarrow 1$ *até* n **faça**
6 $u \leftarrow SA[j] - h$;
7 Mova $T[u, n]$ para a primeira posição disponível em seu *bucket*;
8 Atualizar *buckets*;

Nos próximos passos (linhas 3 a 8), os *buckets-h* são particionados de acordo com a ordenação do dobro de caracteres utilizados no passo anterior, gerando *buckets-2h*. O número de caracteres comparados no passo i é $h = 2^i$ (linha 4). Note que, conforme h aumenta, o número de *buckets-h* aumenta e seu tamanho diminui. Então, quando a quantidade de *buckets-h* for igual a n, cada *buckets-h* terá apenas um elemento e $SA^h = SA$.

A principal questão envolvida nesse algoritmo é como ordenar de forma direta no passo i os sufixos de cada *bucket-h* para obter os *bucket-2h* em tempo linear no próximo passo. As Observações 1 e 2, originalmente propostas por Karp et al. [1972], são a chave para o algoritmo.

Sejam $T[i, n]$ e $T[j, n]$ dois sufixos de T, tais que seus h primeiros caracteres são iguais, isto é, $T[i, n] =_h T[j, n]$.

Observação 1 *Os caracteres $T[i+h]T[i+h+1]\ldots T[n]$ de $T[i,n]$ são exatamente os primeiros caracteres de $T[i+h,n]$.*

Observação 2 *Para obter a ordem-2h de $T[i,n]$ e $T[j,n]$, pode-se comparar a ordem-h de $T[i+h]$ e $T[j+h]$.*

De forma equivalente, dados dois sufixos $T[i,n]$ e $T[j,n]$, a ordem-2h dos sufixos $T[i-h,n]$ e $T[j-h,n]$, com $T[i-h,n] =_h T[j-h,n]$ ($i-h>0$ e $j-h>0$), pode ser definida a partir da ordem-h de $T[i,n]$ e $T[j,n]$. Isto é:

$$T[i,n] =_h T[j,n] \Longrightarrow T[i-h,n] =_{2h} T[j-h,n] \tag{3.3.1}$$

$$T[i,n] <_h T[j,n] \Longrightarrow T[i-h,n] <_{2h} T[j-h,n] \tag{3.3.2}$$

$$T[i,n] >_h T[j,n] \Longrightarrow T[i-h,n] >_{2h} T[j-h,n] \tag{3.3.3}$$

Com isso, no passo $i+1$ ($i = 1,2,3,\ldots,\lceil \log n \rceil$) obtém-se SA^{2h} a partir de SA^h. Os sufixos são ordenados partindo do primeiro sufixo $SA^h[1]$ (linha 5), o qual representa o menor sufixo de *ordem-h*. Seja $SA^h[1] = i$, o sufixo $T[i-h,n]$ deve ser o primeiro sufixo em seu *bucket-h*, portanto, $T[i-h,n]$ é movido para o início de seu *bucket* (linha 7). Em seguida, a partir do próximo sufixo $SA^h[2] = j$, o sufixo $T[j-h,n]$ é movido para a próxima posição disponível em seu *bucket-h* (linha 7). O passo $i+1$ continua com esses movimentos até que todos os sufixos tenham sido processados (quando $i-h \leq 0$, o sufixo é ignorado). Ao final, os *buckets-h* são atualizados para *buckets-2h* (linha 8). O algoritmo termina em no máximo $\lceil \log n \rceil$ passos (linha 3) ou quando cada *bucket-h* possuir apenas um elemento.

Por exemplo, na Figura 3.4 são ilustrados os passos $i = 1$ *e* $i = 2$ (Figura 3.4(b) e Figura 3.4(c), respectivamente) da construção do vetor de sufixo SA, para $T = GATAGA\$$. No passo inicial ($i = 0$ e $h = 1$) do algoritmo MM (Figura 3.4(a)), os sufixos $T[i,n]$ são ordenados de acordo com seus primeiros caracteres, formando 4 *buckets-1*. Em seguida, no próximo passo ($i = 1$ e $h = 2$) (Figura 3.4(b)), os sufixos são ordenados a partir de SA^1. O sufixo $T[7-1,n] = T[6,n] = A\$$ deve ser posicionado no primeiro espaço de seu *bucket-1*, que é a posição 2. Da mesma forma, o sufixo $T[2-1,n] = T[1,n] = GATAGA\$$ é posicionado na posição 5, e assim sucessivamente para os sufixos $T[4-1,n]$, $T[6-1,n]$, $T[5-1,n]$ e $T[3-1,n]$. Ao final desse passo obtém-se SA^2 e os *buckets* são atualizados. O último passo ($i = 2$ e $h = 4$) (Figura 3.4(c)) é executado da mesma forma, resultando em $SA = SA^h$.

i	$SA^1[i]$	$suff(i)$
1	7	**$**
2	2	**A**TAGA$
4	4	**A**GA$
3	6	**A**$
5	1	**G**ATAGA$
6	5	**G**A$
7	3	**T**AGA$

(a) passo 0, SA^1

i	$SA^2[i]$	$suff(i)$
1	7	**$**
2	6	**A$**
4	4	**AG**A$
3	2	**AT**AGA$
5	1	**GA**TAGA$
6	5	**GA**$
7	3	**TA**GA$

(b) passo 1, SA^2

i	$SA^4[i]$	$suff(i)$
1	7	**$**
2	6	**A$**
4	4	**AGA$**
3	2	**ATAG**A$
5	5	**GA$**
6	1	**GATA**GA$
7	3	**TAGA**$

(c) passo 2, $SA^4 = SA$

Figura 3.4: Algoritmo MM, construção de *SA*.

Note que durante o Algoritmo MM apenas os valores i dos sufixos $T[i, n]$ são movidos, não havendo, portanto, a necessidade de deslocar os sufixos em memória. Então, cada passo requer $O(n)$ em tempo no pior caso, e, com isso, a complexidade de tempo do algoritmo é $O(n \log n)$. No entanto, para cadeias com caracteres distribuídos uniformemente, o algoritmo executa, em média, $O(n \log \log n)$ [Adjeroh et al., 2008].

3.3.2 Algoritmo DC3

O algoritmo DC3 (*Difference Cover modulo-3*), proposto por Kärkkäinen et al. [2003, 2006], realiza a construção de *SA*, para um texto de tamanho n, em tempo linear a n. O algoritmo DC3 é resumido no Algoritmo 2.

Algoritmo 2: Algoritmo DC3

Entrada: texto T, $|T| = n$
Saída: vetor de sufixo *SA*

1 **início**
2 **se** $(i - 1) \mod 3 = 0$ **então**
3 $T[i, n]$ é *Tipo 23*;
4 **senão**
5 $T[i, n]$ é *Tipo 1*;
6 $SA_{23} \leftarrow$ Ordenar os sufixos do *Tipo 23*;
7 $SA_1 \leftarrow$ Ordenar os sufixos do *Tipo 1* a partir de SA_{23};
8 $SA \leftarrow$ Unir SA_{23} e SA_1

O algoritmo DC3 divide os sufixos $T[i, n]$ em dois grupos (linha 2). O primeiro grupo, denominado *Tipo 23*, contém todos os sufixos $T[i, n]$, tal que $(i - 1) \mod 3 \neq 0$ (linha 3). O segundo grupo, denominado *Tipo 1*, contém todos os sufixos $T[j, n]$,

tal que $(j-1) \bmod 3 = 0$ (linha 5). Ou seja, o grupo *Tipo 23* contém os sufixos $\{T[2,n], T[3,n], T[5,n], T[6,n], \ldots\}$, e o grupo *Tipo 1* contém $\{T[1], T[4], T[7], \ldots\}$.

Em seguida, os sufixos do grupo *Tipo 23* são ordenados entre eles (linha 6), em um vetor parcial SA_{23}. Isto é feito considerando (inicialmente) os três primeiros caracteres de cada sufixo $T[i,n]$, isto é, as sub-cadeias $T[i,i+2]$. Por meio do algoritmo *radix sort* obtém-se a ordem relativa r $(1 \leq r \leq \frac{2}{3}n)$ entre as sub-cadeias $T[i,i+2]$. Se não houver empates, SA_{23} é ordenado em um único passo. Caso contrário, os caracteres $T[i+3], T[i+4], \ldots$ devem ser comparados por meio de chamadas recursivas.

Por exemplo, na Figura 3.5 é ilustrado a ordenação dos sufixos *Tipo 23*, para $T = GATAGA\$$. Na Figura 3.5(a) são representadas as sub-cadeias $T[i,i+2]$ dos sufixos *Tipo 23* (quando $i+1 > n \Longrightarrow T[i+1] = \$$, similar para $i+2 > n$). Em seguida, na Figura 3.5(b) são ilustradas as sub-cadeias $T[i,i+2]$ ordenadas por meio do *radix sort*. Por fim, na Figura 3.5(c), é ilustrado SA_{23} com todos os sufixos *Tipo 23* ordenados.

i	$T[i,i+2]$
2	ATA
3	TAG
5	GA$
6	A$$

(a) sub-cadeias

r	i	$T[i,i+2]$
1	6	A$$
2	2	ATA
3	5	GA$
4	3	TAG

(b) ordem r

i	$SA_{23}[i]$	$suff(i)$
1	6	A$
2	2	ATAGA$
3	5	GA$
4	3	TAGA$

(c) SA_{23}

Figura 3.5: Algoritmo DC3, ordenação dos sufixos *Tipo 23*

Em seguida, os sufixos do *Tipo 1* são ordenados (linha 8) a partir dos pares $\langle T[j], pos(SA_{23}, T[j+1,n])\rangle$, os quais contêm o primeiro caractere de $T[j,n]$ e $pos(SA_{23}, T[j+1,n])$, que informa a posição do sufixo $T[j+1,n]$ em SA_{23}. Dessa forma, para comparar dois sufixos $T[j,n]$ e $T[k,n]$ do *Tipo 1*, compara-se os caracteres $T[j]$ e $T[k]$ e, se necessário, a ordem relativa dos sufixos $T[j+1,n]$ e $T[k+1,n]$, disponíveis em SA_{23} ($pos(SA_{23}, T[j+1,n])$ e $pos(SA_{23}, T[k+1,n])$). Esses pares podem ser ordenados em tempo linear a partir das seguintes regras:

$$T[j] < T[k] \Longrightarrow T[j,n] < T[k,n] \quad (3.3.4)$$

$$T[j] = T[k] \text{ e } pos(SA_{23}, T[j+1,n]) < pos(SA_{23}, T[k+1,n]) \Longrightarrow T[j,n] < T[k,n] \quad (3.3.5)$$

Por exemplo, na Figura 3.6 é ilustrada a ordenação dos sufixos *Tipo 1*. Na Figura 3.6(a) são ilustrados os pares $\langle T[j], pos(SA_{23}, T[j+1,n])\rangle$ dos sufixos *Tipo 1*, enquanto que na Figura 3.6(b) é ilustrado SA_1, obtido por meio da ordenação dos elementos $T[j]$ dos pares,

uma vez que todos são diferentes (caso contrário, os valores $pos(SA_{23}, T[j+1,n])$ seriam considerados na ordenação).

j	$\langle T[j], pos(SA_{23}, T[j+1,n])\rangle$
1	$\langle G, 2\rangle$
4	$\langle A, 3\rangle$
7	$\langle \$, -\rangle$

(a) pares dos sufixos *Tipo 1*

j	$SA_1[j]$	$suff(j)$
1	7	\$
2	4	AGA\$
3	1	GATAGA\$

(b) SA_1

Figura 3.6: Algoritmo DC3, ordenação dos sufixos *Tipo 1*

No passo final (linha 8), os vetores SA_{23} e SA_1 são unidos para gerar SA. Os sufixos em SA_{23} devem ser comparados com os sufixos em SA_1 para obter SA. Para comparar um sufixo $T[j,n]$ do *Tipo 1* com um sufixo $T[i,n]$ do *Tipo 23*, são considerados os seguintes casos:

- **Caso 1**: quando $(i-1) \mod 3 = 1$, são comparadas as tuplas $\langle T[j], pos(SA_{23}, T[j+1,n])\rangle$ com $\langle T[i], pos(SA_{23}, T[i+1,n])\rangle$. Para esse caso, as ordens relativas entre $T[j+1,n]$ e $T[i+1,n]$ estão disponíveis em SA_{23}.
- **Caso 2**: quando $(i-1) \mod 3 = 2$, são comparadas as tuplas $\langle T[j], T[j+1], pos(SA_{23}, T[j+2,n])\rangle$ com $\langle T[i], T[i+1], pos(SA_{23}, T[i+2,n])\rangle$. Para esse caso, o empate é resolvido usando a tupla, já que a ordem relativa entre $T[i+2,n]$ e $T[j+2,n]$ está disponível em SA_{23}.

Por exemplo, na Figura 3.7 é ilustrada a construção de SA (Figura 3.7(c)) por meio da união de SA_{23} (Figura 3.7(a)) e SA_1 (Figura 3.7(b)). Na primeira comparação, $j = 7$ e $i = 6$ ($SA_1[1] = 7$ e $SA_{23}[1] = 6$). Como $(6-1) \mod 3 = 2$, para determinar a ordem de $T[j = 7, n]$ e $T[i = 6, n]$ é utilizado o Caso 2. Com isso são comparadas as tuplas $\langle T[7], T[8], pos(SA_{23}, T[9,n])\rangle$ e $\langle T[6], T[7], pos(SA_{23}, T[8,n])\rangle$, isto é, $\langle \$, \$, -\rangle$ e $\langle A, \$, -\rangle$ ($T[j] = \$$ e $pos(SA_{23}, T[j,n]) = -$ caso $j > n$). Como $\langle \$, \$, -\rangle < \langle A, \$, -\rangle$, pois $T[7] < T[6]$, o sufixo $T[j = 7, n] < T[i = 6, n]$. Com isso, $T[j = 7, n]$ é colocado na primeira posição disponível de SA (no caso $SA[1]$).

Em seguida, são comparados $j = 4$ e $i = 6$ ($SA_1[2] = 4$ e $SA_{23}[1] = 6$). Como $i = 6$, tem-se o Caso 2 novamente. Então, compara-se $\langle T[4], T[5], pos(SA_{23}, T[6,n])\rangle$ e $\langle T[6], T[7], pos(SA_{23}, T[8,n])\rangle$, isto é, $\langle A, G, 1\rangle$ e $\langle A, \$, -\rangle$. Como $\langle A, \$, -\rangle < \langle A, G, 1\rangle$ (segundo componente $\$ < G$), então $T[j = 6, n] < T[i = 4, n]$, e $T[i = 6, n]$ é colocado em $SA[2]$. O próximo passo compara $j = 4$ e $i = 2$, sendo o menor sufixo colocado em $SA[3]$.

i	$SA_{23}[i]$	$suff(i)$
1	6	A$
2	2	ATAGA$
3	5	GA$
4	3	TAGA$

(a) SA_{23}

j	$SA_1[j]$	$suff(j)$
1	7	$
2	4	AGA$
3	1	GATAGA$

(b) SA_1

i	$SA[i]$	$suff(i)$
1	7	$
2	6	A$
3	4	AGA$
4	2	ATAGA$
5	5	GA$
6	1	GATAGA$
7	3	TAGA$

(c) $SA \leftarrow SA_{23} \cup SA_1$

Figura 3.7: Algoritmo DC3, etapa de união.

Esse processo é repetido sucessivamente até que todos os sufixos em SA_1 e SA_{23} tenham sido analisados. Na Figura 3.7(c) é ilustrado o resultado final com SA construído.

O número de sufixos do *Tipo 1* é igual a $n/3$, e o número de sufixos do *Tipo 23* é $2n/3$. Isto é importante pois as chamadas recursivas são somente sobre sufixos do *Tipo 23*. Então, o tempo de execução do algoritmo DC3 é dado pela regra de recorrência: $\varphi(n) = \varphi(\lceil \frac{2n}{3} \rceil) + O(n)$, a qual é igual à $O(n)$ [Adjeroh et al., 2008].

3.4 Etapa de Consulta

A etapa de consulta consiste em encontrar todas as ocorrências de um padrão P no texto T utilizando SA, com $|P| = m$. Caso o padrão P ocorra em alguma posição i de T, então $P = T[i, i + m - 1]$ e P é um prefixo do sufixo $T[i, n]$. Em SA os sufixos $T[i, n]$ estão ordenados, com isso, a consulta pode ser realizada por meio de uma busca binária em SA. Dessa forma, uma consulta em SA realiza no máximo $\lceil \log n \rceil + occ$ passos, onde occ é o número de ocorrências de P em T. A cada passo, são realizados no máximo m comparações. Portanto, a complexidade de tempo da consulta é $O(m \log n + occ)$.

Por exemplo, considere o vetor de sufixo SA ilustrado na Figura 3.8. A busca pelo padrão $P = AGA$ em SA é realizada em 3 passos ($\lceil \log 7 \rceil = 3$), representados pelas setas ao lado direito na Figura 3.8. No passo $i = 1$, compara-se P com $SA[4]$, e no caso, $P < T[SA[4], n] = ATAGA\$$. Com isso, no passo $i = 2$, compara-se P com $SA[2]$ e, como $P > T[SA[2]] = A\$$, no passo $i = 3$ compara-se P com $SA[3]$ e $P =_3 T[SA[3], n] = AGA\$$, encontrando-se a ocorrência do padrão P em $SA[3]$, isto é, P ocorre em $T[4]$.

A etapa de consulta pode ser melhorada utilizando SA aumentado com estruturas auxiliar [Kim et al., 2003b; Abouelhoda et al., 2004]. Duas importantes melhorias para a

i	$SA[i]$	$suff(i)$	
1	7	\$	
2	6	A\$	$\leftarrow 2$
3	4	AGA\$	$\leftarrow 3$
4	2	ATAGA\$	$\leftarrow 1$
5	5	GA\$	
6	1	GATAGA\$	
7	3	TAGA\$	

Figura 3.8: Busca por $P = AGA$ em SA para $T = GATAGA\$$

busca binária foram propostas por [Manber e Myers, 1993]. Essas melhorias são descritas nesta dissertação, nas Seções 3.4.1 e 3.4.2, respectivamente, por serem base para trabalhos voltados à consulta em vetores de sufixo em memória externa, os quais são descritos no Capítulo 4. Além disso, na Seção 3.4.3 são descritas outros tipos de consultas que podem ser feitas utilizando vetores de sufixo generalizados.

3.4.1 Otimização simples

Nessa seção é descrita uma otimização simples proposta por Manber e Myers [1993] para o algoritmo de busca em SA. Essa otimização utiliza os valores de lcp para reduzir comparações repetidas durante cada consulta. Sejam $[L, R]$ os limites do intervalo de uma consulta no passo i da busca binária. No passo $i = 1$ temos $[L, R] = [1, n]$. Em cada passo i, uma consulta é feita na posição $M = (L + R)/2$ de SA. O intervalo $[L, R]$ do primeiro passo da busca por $P = AGA$ em SA é ilustrado na Figura 3.9.

i	$SA[i]$	$suff(i)$	
1	7	\$	$\leftarrow$ L
2	6	A\$	
3	4	AGA\$	
4	2	ATAGA\$	$\leftarrow$ M
5	5	GA\$	
6	1	GATAGA\$	
7	3	TAGA\$	$\leftarrow$ R

Figura 3.9: Primeiro passo da busca por $P = AGA$ em SA

Sejam os valores $l = lcp(T[SA[L], n], P)$, $r = lcp(T[SA[R], n], P)$ e $mlr = min(l, r)$. O valor mlr pode ser usado para acelerar a comparação de P e $T[SA[M], n]$, já que todos os sufixos no intervalo $[L, R]$ compartilham o mesmo prefixo de tamanho mlr. Dessa forma,

no passo i a comparação do padrão P com $T[SA[M], n]$ pode começar a partir do caractere $mlr + 1$.

Note que o valor de mlr deve ser atualizado a cada passo. No primeiro passo, quando $[L, R] = [1, n]$, as comparações entre P e os sufixos $T[SA[1], n]$ e $T[SA[n], n]$ são feitas explicitamente. Nos próximos passos, os valores l, r e mlr são atualizados de acordo com $lcp(T[SA[M], n], P)$, o qual é obtido durante a comparação de P com $T[SA[M], n]$. Se $P < T[SA[M], n]$, então $r \leftarrow lcp(T[SA[M], n], P)$, caso contrário, $l \leftarrow lcp(T[SA[M], n], P)$. Atualizar e manter o valor mlr adiciona uma pequena sobrecarga ao algoritmo, porém reduz um grande número de comparações redundantes. Entretanto, quando $l \neq r$, todos os caracteres de $P[mlr+1, max(l, r)]$ são comparados novamente. Com isso, no pior caso a complexidade da busca binária utilizando a otimização simples continua $O(m \log n + occ)$.

3.4.2 Otimização complexa

Nessa seção é descrita uma otimização complexa proposta por Manber e Myers [1993] para o algoritmo de busca em SA. Essa otimização utiliza todos os valores de lcp entre as possíveis tuplas (L, R, M) de uma busca em SA para evitar comparações repetidas durante a consulta. Sejam $lcp(L, M) = lcp(T[SA[L], n], T[SA[M], n])$ e $lcp(M, R) = lcp(T[SA[M], n], T[SA[R], n])$. Em cada passo i, a comparação do padrão P envolve três possíveis casos (assume-se que $l > r$, caso contrário basta inverter as regras):

- **Caso 1** ($lcp(L, M) > l$): nesse caso, $T[SA[M], n] =_{l+1} T[SA[L], n] \neq_{l+1} P$, de outra forma, $P =_l T[SA[M], n]$ e os caracteres $T[SA[M]+l+1] = T[SA[L]+l+1] < P[l+1]$. Então não é possível encontrar ocorrências de P no intervalo $[L, M]$. Nesse caso, nenhuma comparação em P é necessária, e $L \leftarrow M$, e l e r não mudam.

- **Caso 2** ($lcp(L, M) < l$): nesse caso, $P =_{lcp(L,M)} T[SA[M], n]$, e $P =_{lcp(L,M)+1} T[SA[L], n] <_{lcp(L,M)+1} T[SA[M], n]$. Então as ocorrências de P devem ocorrer (se houver) no intervalo $[L, M]$. Nesse caso, nenhuma comparação em P é necessária, e $R \leftarrow M$ e $r \leftarrow lcp(L, M)$, e l não muda.

- **Caso 3** ($lcp(L, M) = l$): nesse caso, $P =_l T[SA[M], n]$. O algoritmo compara P com $T[SA[M], n]$ a partir dos caracteres na posição $l + 1$. O resultado dessa comparação determina em qual intervalo a busca deve continuar (ou seja, $[L, M]$ ou $[M, R]$), e o valor correspondente l ou r é atualizado.

A quantidade de tuplas (L, M, R) possíveis em um vetor de sufixo com n posições é $n-2$, cada uma com um único ponto médio $M \in [2, n-1]$ e $1 \leq L < M < R < n$. Os valores $lcp(L, M)$ e $lcp(M, R)$ das tuplas (L, M, R) podem ser calculados durante a etapa de construção do Algoritmo MM (Seção 3.3.1), ou obtidos posteriormente a partir de *SA*, o que adiciona um custo de tempo e espaço ao algoritmo. Com isso, apesar do tempo da busca binária utilizando a otimização complexa ser reduzido para $O(m + \log n)$, na prática o algoritmo não é tão eficiente. Manber e Myers [1993] reportaram que o uso do *mlr* sozinho (otimização simples) permite que a busca seja tão rápida quanto o método $O(m + \log n)$ (otimização complexa).

3.4.3 Outras Consultas

Consultas envolvendo conjuntos de cadeias podem ser realizadas utilizando vetores de sufixo generalizados. Uma vez construído *GSA* para um conjunto de k cadeias $\mathcal{T}$, com tamanho $N = \sum_{i=1}^{k} |T_i|$, é possível encontrar todas as ocorrências de um padrão P, $|P| = m$, em todas as cadeias $T_i \in \mathcal{T}$ utilizando os mesmos algoritmos desenvolvidos para *SA* em tempo $O(m \log N + occ)$ (otimização simples) ou $O(m + \log N + occ)$ (otimização complexa).

Além disso, outros problemas podem ser resolvidos comparando todas as cadeias indexadas em um vetor de sufixo generalizado aumentado. Alguns desses problemas são especificados nas Definições 3.21 e 3.22.

Definição 3.21 *Sejam T_a e T_b duas cadeias em $\mathcal{T}$, a sub-cadeia comum mais longa entre T_a e T_b corresponde à maior cadeia S, $|S| = s$, tal que: $T_a[i_a, i_a + s] = T_b[i_b, i_b + s]$.*

Para encontrar a sub-cadeia em comum mais longa entre duas cadeias T_a e T_b utilizando *GESA* é necessário encontrar um intervalo $[l, r]$, tal que sufixos de T_a e T_b ocorram em $[l, r]$ e que $rmq_{LCP}(l, r)$ seja máximo. Essa consulta pode ser feita por meio de uma busca sequencial em *GESA* em tempo $O(N)$, comparando apenas valores de *lcp*. Por exemplo, na Figura 3.3, os sufixos que estão nas posições $[13, 14]$ possuem sufixos das cadeias T_1 e T_2 e possuem os valores $rmq_{LCP}(l, r)$ máximos, isto é, igual a 4. Então, a sub-cadeia comum mais longa entre T_1 e T_2 é "TAGA".

Esse problema pode ser generalizado para o problema de encontrar a sub-cadeia comum mais longa em todo o conjunto $\mathcal{T}$.

Definição 3.22 *Seja um conjunto de k cadeias $\mathcal{T}$, a sub-cadeia comum mais longa entre todas as cadeias de $\mathcal{T}$ corresponde à maior cadeia S, $|S| = s$, tal que: $T_1[i_1, i_1 + s] = T_2[i_2, i_2 + s] = \cdots = T_k[i_k, i_k + s]$.*

Para resolver o problema generalizado, é necessário estender a solução anterior e encontrar intervalos aonde ocorram sufixos de todas as cadeias de $\mathcal{T}$.

Na literatura, vetores de sufixo generalizados são utilizados para resolver diferentes problemas, como montagem de genomas [Gonnella e Kurtz, 2012a], identificação de padrões que não ocorrem em um conjunto (*word absent*) [Pinho et al., 2009], casamento de todos os pares sufixo-prefixo [Ohlebusch e Gog, 2010], identificação de repetições [Arnold e Ohlebusch, 2011], recuperação de informação em documentos [Välimäki e Mäkinen, 2007] e processamento de linguagem natural [Hui et al., 2009], dentre outros.

3.5 Considerações Finais

Nesse capítulo foram descritos conceitos relacionados aos vetores de sufixo. Na Seção 3.1 foram descritas terminologias utilizadas nesta dissertação. Em seguida, na Seção 3.2 foram descritos os vetores de sufixo. Nas Seções 3.3 e 3.4 foram discutidas as etapas de construção e consulta para vetores de sufixo em memória interna, respectivamente. Por fim, na Seção 3.4.3 foram discutidas consultas específicas que podem ser realizadas utilizando vetores de sufixo generalizados.

No contexto de bioinformática, devido ao enorme volume de dados envolvidos nos BDBs, é importante desenvolver métodos para vetores de sufixo generalizados em memória externa. Entretanto, na literatura não há trabalhos voltados especificamente para a construção de vetores de sufixo generalizados em memória externa. Portanto, no Capítulo 4 são descritos trabalhos correlatos para a construção de vetores de sufixo em memória externa.

Capítulo

4

Trabalhos Correlatos

Nesse capítulo são resumidos trabalhos correlatos voltados à construção de vetores de sufixo em memória externa. Na Seção 4.1 é descrito o primeiro trabalho proposto para construção em memória externa, descrito em [Gonnet et al., 1992]. Em seguida, na Seção 4.2 é descrito o trabalho de Crauser e Ferragina [1999, 2002], no qual são apresentadas adaptações e comparações de soluções já existentes em memória interna para memória externa. Na Seção 4.3 é descrito o trabalho de Dementiev et al. [2005, 2008], no qual é apresentada uma metodologia para desenvolver, analisar, e implementar algoritmos em memória externa. Na Seção 4.4 é descrito o trabalho de Bingmann et al. [2013], no qual é apresentado o primeiro algoritmo para construção do vetor de sufixo aumentado com o vetor *LCP* em memória externa. Por fim, na Seção 4.5 são discutidas as considerações finais.

4.1 Gonnet et al.(1992)

Gonnet et al. [1992] desenvolveram, durante a proposta de um índice para o *New Oxford English Dictionary*, uma estrutura de dados similar ao vetor de sufixo, que foi chamada de *PAT-array*. O algoritmo proposto para a construção do *PAT-array*, chamado de GBS, considera a estrutura em memória externa. Com isso, o algoritmo GBS pode ser considerado o primeiro trabalho proposto para construção de vetores de sufixo em memória externa.

O algoritmo GBS constrói incrementalmente SA, para um texto T de tamanho n com capacidade de memória interna M, em n/M passos. Seja $l < 1$ uma constante positiva, tal que $m = l \times M$. O texto T é dividido em n/m partições T^i de tamanho m, isto é, $T^i = T[(i-1) \times m + 1, i \times m]$. Por conveniência, assume-se que m divide n. O algoritmo preserva a seguinte invariante: *No começo de cada passo i ($i = 1, 2, \ldots, n/m$), o algoritmo possui armazenado em memória externa um vetor de sufixo parcial SA_{ext}, o qual contém todos os primeiros $(i-1) \times m$ sufixos ordenados, isto é, todos os sufixos* $T[j, n], \forall j \in [1, (h-1) \times m]$. Durante o passo i, o algoritmo atualiza SA_{ext} com os sufixos de T^i, mantendo-os em ordem. Com isso, a invariante é preservada. O passo i do algoritmo GBS é resumido a seguir.

Inicialmente no passo i, a partição T^i é carregada em memória interna e o vetor de sufixo correspondente SA_i é construído, isto é, SA_i fornece a ordem de todos os sufixos de T^i. Em seguida, a união dos vetores SA_i e SA_{ext} é feita da seguinte forma. Um vetor contator C de tamanho $m + 1$ é calculado, sendo que cada entrada $C[l]$ corresponde à quantidade de sufixos em SA_{ext} que são maiores que $SA_i[l-1]$ ou menores que $SA_i[l]$. Em particular, o vetor C é obtido por meio da leitura dos sufixos $T[j, n]$ em memória externa ($\forall j \in [1, (h-1) \times m]$). Para cada $T[j, n]$, realiza-se uma busca binária em SA_i e C é incrementado em 1 na posição l encontrada na busca, ou seja, $C[l]$ indica quantos sufixos em SA_{ext} estão entre $SA_i[j-1]$ e $SA_i[j]$. Com isso, a informação em C é utilizada para unir SA_i e SA_{ext}. No final, depois de todos os n/m passos, $SA_{ext} = SA$.

O algoritmo GBS executa dois acessos à memória externa principais: o primeiro para carregar a partição T^i em memória interna e o segundo para unir SA_{ext} e SA_i. Adicionalmente, alguns acessos aleatórios podem ser necessários em duas situações: durante a construção de SA_i ou durante o cálculo de $C[j]$. Em ambos os casos, a comparação de dois sufixos pode requerer acessos às sub-cadeias que não estão em memória interna, isto é, fora da partição T^i. Portanto, a complexidade teórica do algoritmo GBS é $O(n^3)$. Entretanto, na prática as duas situações citadas anteriormente raramente acontecem, isto é, os caracteres necessários para determinar a ordem dos sufixos normalmente estão em memória interna. Portanto, apesar da complexidade teórica $O(n^3)$, na prática o comportamento do algoritmo GBS é $O(n^2)$ [Crauser e Ferragina, 2002].

Por fim, na proposta original, Gonnet et al. [1992] não desenvolveram um algoritmo específico para construção dos vetores SA_i. Eles apenas consideram que SA_i pode ser construído por um algoritmo de ordenação padrão e indicam o uso do *quick-sort*.

4.2 Crauser e Ferragina (1999, 2002)

Crauser e Ferragina [1999, 2002] realizaram um estudo teórico e experimental com os seguintes algoritmos: (i) MM [Manber e Myers, 1993], (ii) GBS [Gonnet et al., 1992], (iii) *Doubling* [Arge et al., 1997], e três novos algoritmos propostos pelos autores: (iv) *Doubling+Discarding*, (v) *Doubling+Discarding and Radix Heaps* e (vi) *L-pieces*. Os algoritmos (i) e (iii) foram originalmente propostos para construção de vetores de sufixo em memória interna. Os algoritmos (iv) e (v) são otimizações de (iii), enquanto que o algoritmo (vi) propõe a construção do vetor de sufixo em L vetores de sufixo parciais, de forma que cada vetor parcial caiba em memória interna.

Todos os algoritmos foram implementados em *C++* utilizando a biblioteca *LEDA-SM* [Crauser e Mehlhorn, 1999]. Nos testes realizados, o algoritmo mais eficiente foi o algoritmo GBS. Apesar de sua complexidade ser $O(n^3)$, devido ao pequeno tamanho de memória auxiliar utilizada e ao seu padrão de acessos sequenciais à memória externa, o algoritmo GBS demonstrou um comportamento quadrático durante os testes.

Utilizando como base os resultados obtidos, Crauser e Ferragina [2002] propuseram um novo algoritmo, baseado nas ideias do algoritmo GBS. Esse novo algoritmo, chamado de GBS-new, trabalha da seguinte forma. O algoritmo GBS-new divide o texto T em k partições T^i, e constrói SA em memória externa incrementalmente em k passos, da mesma forma que o algoritmo GBS. No entanto, as partições T^i são consideradas do final para o começo de T, isto é, $T = T^k \cdot T^{k-1} \ldots T^2 \cdot T^1$ (T^i é numerada de acordo com o passo i, no qual é processada). O objetivo dessa alteração é, a cada passo i, evitar acessos aleatórios à memória externa durante a construção de SA_i e atualização de SA_{ext}. Para isso, o novo algoritmo utiliza estruturas de dados auxiliares para que as comparações entre os sufixos utilizem apenas informações em memória interna.

O algoritmo preserva a seguinte invariante: *Seja $S = T^{i-1} \cdot T^{i-2} \ldots T^2 \cdot T^1$ a parte do texto T processada até o passo $(i-1)$. No começo de cada passo i, o algoritmo possui armazenado duas estruturas de dados: (i) o vetor de sufixo parcial SA_{ext}, que contém todos os sufixos $T[j,n]$ ordenados, $\forall T[j,n] \in S$, e (ii) o vetor $POS_{SA_{ext}}$, que armazena as posições dos sufixos $T[j,n]$ em SA_{ext} ($pos(SA_{ext}, T[j,n])$).* No passo i, cada sufixo $T[j,n]$ pertencente a T_i pode ser visto como $T[j,n] = T[j, j+m-1] \cdot T[j+m, n]$. Então $T[j,n]$ é representado pelo par $\langle T[j, j+m-1], POS_{SA_{ext}}[(j+m)]\rangle$. Dessa forma, comparações entre dois sufixos de T^i que precisem acessar as partições $T^{i-1}T^{i-2}\ldots$ podem utilizar o vetor $POS_{SA_{ext}}$, o qual indica a ordem das sub-cadeias (sufixos) $T[j+m, n]$ em SA_{ext}. Depois de todos os k passos, $S = T$ e $SA_{ext} = SA$.

O algoritmo GBS-new reduz a complexidade do tempo de construção para $O(n^2/M^2)$. Além disso, os autores indicam o uso do algoritmo MM [Manber e Myers, 1993] para construção de SA_i, diferentemente da proposta original do algoritmo GBS, na qual os autores sugeriam o uso do *quick-sort*.

4.3 Dementiev et al. (2005, 2008)

Dementiev et al. [2005, 2008] propuseram uma metodologia para desenvolver, analisar, e implementar algoritmos em memória externa. Essa metodologia foi empregada para construção de vetores de sufixo. A ideia principal é prover uma interface na qual a saída de uma etapa do algoritmo seja fornecida diretamente, como um fluxo de dados, para a entrada de outra etapa do algoritmo (*pipelining*), sem a necessidade de escrever esses dados em memória externa.

O *pipelining* proposto representa o fluxo de dados em um grafo direcionado acíclico $G = (V = F \cup S \cup R, E)$. Os nós de arquivos $f \in F$ representam os dados que devem ser armazenados em memória externa. Quando um nó f é acessado, é preciso um *buffer* de tamanho $b(f)$. Os nós de *streaming* $s \in S$ representam a leitura de zero ou mais cadeias ou a saída de zero ou mais novas cadeias usando *buffers* de tamanho $b(s)$. Os nós de ordenação $r \in R$ representam a leitura de uma cadeia S e a saída de S ordenada. As arestas $e \in E$ são rotuladas com o número de palavras $w(e)$ trocadas entre os nós.

Para demonstrar a metodologia proposta, foram adaptados cinco algoritmos seguindo a técnica de *pipelining*: (i) *Doubling* [Arge et al., 1997], (ii) *Doubling+Discarding* [Crauser e Ferragina, 2002]; (iii) *Quadrupling*; (iv) *Quadrupling+Discarding*; e (v) DC3 [Kärkkäinen et al., 2003, 2006]. Os algoritmos (iii) e (iv) são modificações propostas sobre (ii). Os autores descartaram do estudo o algoritmo GBS [Gonnet et al., 1992] (Seção 4.1), pois o consideraram ineficiente e aplicável apenas para cadeias pequenas.

Todos os algoritmos foram implementados em *C++* utilizando a biblioteca *STXXL* [Dementiev, 2005], a qual foi projetada para tratar algoritmos para memória externa considerando a técnica de *pipelining*. Na biblioteca STXXL os nós de fluxo de dados são implementados como objetos com interface similar aos iteradores da biblioteca STL [Stepanov e Lee, 1994].

Nos testes realizados, entre todos os algoritmos investigados o algoritmo DC3 para memória externa demonstrou o melhor desempenho. Foram consideradas cadeias de até 4 GB de diferentes tipos de dados nos testes. O fluxo de dados do algoritmo DC3 é

ilustrado na Figura 4.1, onde cada nó representa uma etapa do algoritmo DC3 para memória externa.

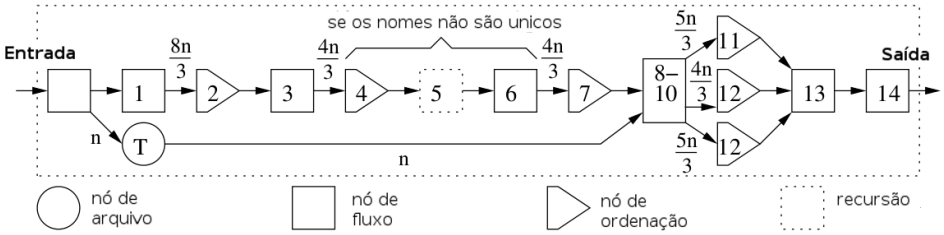

Figura 4.1: Fluxo de dados: Algoritmo DC3, adaptado de [Dementiev et al., 2008]

Em particular, os nós na Figura 4.1 são numerados a partir das linhas do algoritmo DC3 estendido descrito em [Dementiev et al., 2008]. De forma simplificada, o nó 1 representa a leitura do texto T. Em seguida, os nós 2 a 6 representam a classificação dos sufixos em *Tipo 1* e *Tipo 23* e ordenação dos sufixos do *Tipo 23*. Caso haja empate nessa ordenação, o algoritmo realiza uma chamada recursiva para ele mesmo (nós 4 e 6). O algoritmo realiza a ordenação dos sufixos do *Tipo 1* (nó 7) utilizando a ordenação dos sufixos do *Tipo 23*, caso necessário. Por fim, é realizada a união dos sufixos *Tipo 23* e *Tipo 1* (nós 8 a 13) formando o vetor de sufixo *SA* (nó 14). A complexidade de tempo do algoritmo DC3 para memória externa é $O(sort(n))$ I/Os, onde $sort(n)$ é o número de I/Os necessários para ordenar todos os sufixos do texto T.

4.4 Bingmann et al. (2013)

Bingmann et al. [2013] propuseram o primeiro algoritmo para a construção de vetores de sufixo aumentado com o vetor *LCP* em memória externa. O algoritmo proposto, chamado de eSAIS, estende o algoritmo de construção de vetores de sufixo em memória interna SAIS [Nong et al., 2011]. De forma resumida, o algoritmo original SAIS seleciona e ordena um subconjunto dos sufixos de T, e utiliza a ordem desses sufixos já ordenados para induzir a ordem dos sufixos ainda não ordenados [Dhaliwal et al., 2012].

Bingmann et al. [2013] reformularam o algoritmo SAIS utilizando apenas leituras sequenciais, ordenação, união e filas de prioridades, os quais podem ser implementados eficientemente por meio da biblioteca *STXXL* [Dementiev, 2005]. De forma semelhante ao algoritmo DC3 para memória externa (descrito na Seção 4.3), o fluxo de dados do eSAIS é ilustrado na Figura 4.2.

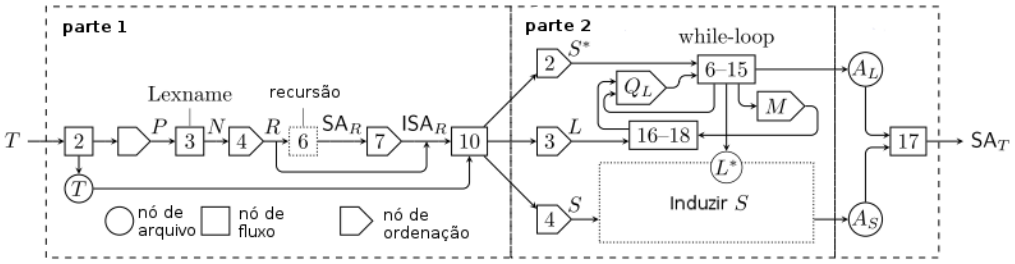

Figura 4.2: Fluxo de dados: Algoritmo eSAIS, adaptado de [Bingmann et al., 2013]

Em particular, os nós na Figura 4.2 são numerados a partir das linhas do algoritmo em [Bingmann et al., 2013]. De forma simplificada, os nós 2 a 10 (parte 1) representam a leitura do texto T, classificação dos sufixos em tipo S^*, L e S, e ordenação dos sufixos do tipo S^*. Caso haja empate nessa ordenação, o algoritmo realiza uma chamada recursiva para ele mesmo para resolver os empates (nós 6 e 7 da parte 1). Em seguida, cada tipo de sufixo é encaminhado (nó 10 da parte 1) para seu respectivo nó de ordenação, um para cada tipo de sufixo (nós 2, 3 e 4 da parte 2). Os sufixos do tipo S^* já estão ordenados e são utilizados para induzir os sufixos do tipo L (nós 6 a 15 da parte 2), que por sua vez são utilizados para induzir os sufixos do tipo S (nós 16 a 18 da parte 2). Conforme os sufixos são ordenados, eles são escritos em um arquivo de saída, os sufixos do tipo L são escritos em A_L e os do tipo S e S^* em A_S. Por fim, o vetor de sufixo final é construído por meio da união (nó 17 da parte 2) dos vetores parciais A_L e A_S.

O algoritmo eSAIS foi comparado com o algoritmo DC3 para memória externa [Dementiev, 2005] (Seção 4.3) utilizando cadeias de caracteres de até 80 GB de diferentes tipos de dados. Os resultados mostraram que o algoritmo eSAIS reduziu o tempo de construção do vetor de sufixo por um fator de 2 à 3 vezes. Além disso, os autores também introduziram uma extensão do algoritmo DC3 para memória externa para a calcular o vetor *LCP* durante a construção do vetor de sufixo. Nesse caso, o eSAIS manteve o mesmo ganho sobre o DC3 para memória externa estendido. Por fim, a complexidade de tempo do algoritmo eSAIS é $O(n)$ além de $O(n/B)$ I/Os, onde B corresponde ao número de partições necessárias para armazenar T em memória interna.

O algoritmo eSAIS é o mais eficiente para construção de vetores de sufixo aumentados com *LCP* em memória externa, e é o trabalho correlato utilizado como comparativo com o trabalho proposto nesta dissertação (apresentado no Capítulo 5) para construção de vetores de sufixo generalizados aumentado com o vetor *LCP* em memória externa.

4.5 Considerações Finais

Nesse capítulo foram descritos trabalhos correlatos voltados à construção de vetores de sufixo em memória externa. O trabalho de Gonnet et al. [1992], descrito na Seção 4.1, apresentou a primeira proposta para construção de vetores de sufixo em memória externa. Em seguida, o trabalho de Crauser e Ferragina [1999, 2002], descrito na Seção 4.2, apresentou adaptações e comparações de soluções já existentes em memória interna para memória externa. Entretanto, essas soluções não são eficientes e os testes realizados consideraram apenas pequenas cadeias de caracteres. Em contrapartida, o trabalho de Dementiev et al. [2005, 2008], descrito na Seção 4.3, apresentou algoritmos eficientes, e testes com cadeias de até 4 GB. Por fim, o trabalho de Bingmann et al. [2013], descrito na Seção 4.4, apresentou o primeiro algoritmo para construção do vetor de sufixo aumentado com o vetor *LCP* em memória externa.

Uma limitação dos trabalhos descritos neste capítulo é que eles não são voltados à indexação de conjuntos de cadeias, isto é, eles não consideram a construção de vetores de sufixo generalizados. Em especial, no melhor de nosso conhecimento, não existem trabalhos correlatos na literatura que enfoquem a construção de vetores de sufixo generalizados em memória externa. Esta dissertação tem como objetivo suprir essa limitação, por meio da proposta do algoritmo eGSA, voltado a esse fim. O algoritmo proposto, bem como testes de desempenho que comprovam a viabilidade de sua utilização em cadeias de caracteres grandes e pequenas são apresentados no Capítulo 5.

Capítulo

5

Algoritmo eGSA

Nesse capítulo é apresentado o algoritmo eGSA, o primeiro algoritmo para construção de vetores de sufixo generalizados aumentados com os vetores LCP e BWT em memória externa. Na Seção 5.1 é descrito o algoritmo proposto, o qual é detalhado nas Seções 5.1.1 e 5.1.2. Em seguida, na Seção 5.2 são descritos testes de desempenho com conjuntos de cadeias grandes e de cadeias pequenas. Na Seção 5.3 é feita uma análise teórica da complexidade de tempo e espaço do algoritmo. Por fim, na Seção 5.4 são feitas as considerações finais.

5.1 Proposta

O algoritmo proposto, chamado de eGSA (acrônimo para *External Generalized Enhanced Suffix Array Construction Algorithm*) publicado em [Louza et al., 2013], recebe como entrada um conjunto de k cadeias de caracteres $\mathcal{T} = \{T_1, \ldots, T_k\}$, as quais são armazenadas em memória externa, e possuem tamanhos $n_1, \ldots, n_k$ (ou seja, tamanho total do conjunto $\mathcal{T}$ é igual a $N = \sum_{i=1}^{k} n_i$). O algoritmo gera como saída o vetor de sufixo generalizado aumentado com os vetores *LCP* e *BWT* (*GESA*) para o conjunto $\mathcal{T}$, o qual é armazenado em memória externa.

O algoritmo eGSA é baseado na técnica de divisão e conquista para ordenação em memória externa *Two-Phase, Multiway Merge-Sort* proposta por Garcia-Molina et al.

[1999]. Em resumo, o algoritmo eGSA funciona da seguinte forma. Na primeira fase, para cada cadeia $T_i \in \mathcal{T}$, os vetores SA_i, LCP_i e BWT_i referentes à T_i são construídos em memória interna e armazenados em memória externa. Em seguida, na segunda fase esses vetores são unidos utilizando *buffers* em memória interna em conjunto com um método melhorado para comparação de sufixos, obtendo o *GESA* para o conjunto $\mathcal{T}$ em memória externa.

As duas fases do algoritmo eGSA são detalhadas a seguir. Durante esse capítulo, são considerados exemplos baseados no conjunto de textos $\mathcal{T} = \{T_1, T_2\}$, sendo que $T_1 = GATAGA\$$ e $T_2 = TAGAGA\$$. Esse mesmo conjunto de textos foi utilizado nos exemplos do Capítulo 3.

5.1.1 Fase 1: Ordenação Interna

Na primeira fase do algoritmo eGSA, para cada cadeia $T_i \in \mathcal{T}$, T_i é carregada (da memória externa) para a memória interna e os vetores SA_i e LCP_i são construídos. Em seguida, são obtidos o vetor BWT_i e o vetor de prefixos PRE_i, os quais são utilizados para melhorar a segunda fase do algoritmo. No final, esses vetores são armazenados em memória externa na forma de um vetor composto R_i de forma que:

Definição 5.1 $R_i[j] = \langle SA_i[j], LCP_i[j], BWT_i[j], PRE_i[j] \rangle$.

Os vetores SA_i e LCP_i podem ser obtidos utilizando qualquer algoritmo de construção em memória interna (*e.g.* [Nong, 2013; Gog e Ohlebusch, 2011; Fischer, 2011]). Caso não haja memória interna suficiente, podem ser utilizados algoritmos de construção em memória externa [Bingmann et al., 2013]. No caso de grandes conjuntos de cadeias pequenas, o conjunto $\mathcal{T}$ pode ser dividido em r partições $\mathcal{T}^1, \ldots, \mathcal{T}^r$, de forma que seja possível construir um *GSA* para cada partição utilizando algum algoritmo em memória interna (*e.g.* [Shi, 1996; Simpson e Durbin, 2010]). Nesse caso, ao invés de construir um vetor SA_i é construído um vetor GSA_i para cada partição, e o vetor LCP_i é calculado da mesma forma.

O vetor BWT_i é construído diretamente de SA_i (Definição 3.14) sem custo adicional de memória, isto é, o elemento $BWT_i[j]$ é obtido durante a escrita do valor de $R_i[j]$ em memória externa.

O vetor de prefixos armazena o início de cada sufixo em SA_i. Definido inicialmente por Barsky et al. [2009], o vetor de prefixo armazena para cada sufixo $T_i[SA_i[j], n_i]$ a subcadeia $T_i[SA_i[j], SA_i[j] + p]$, para um tamanho p constante. Entretanto, a probabilidade

de que dois valores consecutivos sejam iguais é alta, já que os sufixos $T_i[SA_i[j-1], n_i]$ e $T_i[SA_i[j], n_i]$ estão ordenados em SA_i. Então, para evitar redundâncias, foi proposta para o algoritmo eGSA uma nova definição para o vetor de prefixos de SA_i, chamado de PRE_i. Essa estratégia é similar à apresentada pelo algoritmo de consulta na estrutura LOF-SA [Sinha et al., 2008].

Definição 5.2 $PRE_i[j] = T_i[SA_i[j]+h_j, SA_i[j]+h_j+p]$, *para* $h_j = min(LCP_i[j], h_{j-1}+p)$ *e* $h_0 = 0$.

No vetor PRE_i, o elemento $PRE_i[j]$ armazena os p primeiros caracteres não comuns aos sufixos $T_i[SA_i[j-1], n_i]$ e $T_i[SA_i[j], n_i]$ (caso $h_j = LCP_i[j]$) ou os que são comuns mas não foram armazenados nos elementos anteriores à $PRE_i[j]$ (caso $h_j = h_{j-1}+p$). O vetor PRE_i também pode ser obtido diretamente durante a escrita dos valores de SA_i e LCP_i em memória externa (semelhante ao vetor BWT_i).

Na Figura 5.1 são ilustrados os vetores de prefixo PRE_1 e PRE_2 para o conjunto $\mathcal{T}$ (referentes aos vetores ilustrados na Figura 3.2), com valor de $p = 3$. Por conveniência, quando $SA_i[j] + h_j + p > n_i = 7$, $T_i[SA_i[j] + h_j + p]$ é igual a \$.

j	$PRE_1[j]$	$suff_1(j)$
1	\$\$\$	**\$**
2	A\$\$	**A\$**
3	GA\$	**AGA\$**
4	TAG	**ATAG**A\$
5	GA\$	**GA\$**
6	TAG	**GATAG**A\$
7	TAG	**TAG**A\$

(a) PRE_1 para $T_1 = GATAGA\$$

j	$PRE_2[j]$	$suff_2(j)$
1	\$\$\$	**\$**
2	A\$\$	**A\$**
3	GA\$	**AGA\$**
4	GA\$	**AGA**GA\$
5	GA\$	**GA\$**
6	GA\$	**GAGA\$**
7	TAG	**TAG**AGA\$

(b) PRE_2 para $T_2 = TAGAGA\$$

Figura 5.1: Vetores PRE_i para T_1 e T_2, com $p = 3$.

No final, o tamanho total de cada vetor R_i é igual à soma dos tamanhos dos vetores $SA_i, LCP_i, BWT_i, PRE_i$, que é igual a $4 + 4 + 1 + p$ *bytes*.

5.1.2 Fase 2: União Externa

Na segunda fase do algoritmo eGSA é realizada a união de todos os vetores R_i construídos na primeira fase, obtendo *GESA* para $\mathcal{T}$.

Cada vetor R_i é particionado em r_i blocos consecutivos $R_i^1, \ldots, R_i^{r_i}$ de tamanho b, exceto talvez para $R_i^{r_i}$, que pode ser menor que b. Para cada texto T_i, o algoritmo utiliza

dois *buffers* em memória interna: um *buffer* de cadeia S_i, no qual são armazenadas subcadeias de até s caracteres de T_i, e um *buffer* de partição B_i, no qual são armazenados blocos R_i^j, tal que $B_i[j] = R_i[m]$, para $j = m \bmod b$. Também são utilizados dois *buffers* gerais: (i) o *buffer* de saída D, que armazena d elementos de *GESA*, tal que $D[j] = \langle GSA[m], LCP[m], BWT[m] \rangle$, para $j = m \bmod d$; (ii) o *buffer* de sufixos induzidos I, de tamanho $|\Sigma| \times c$, que armazena informações necessárias para a estratégia de indução. Os valores de s, b, d e c determinam a quantidade de memória interna necessária nessa fase. Na Figura 5.2 são ilustrados todos os *buffers* utilizados.

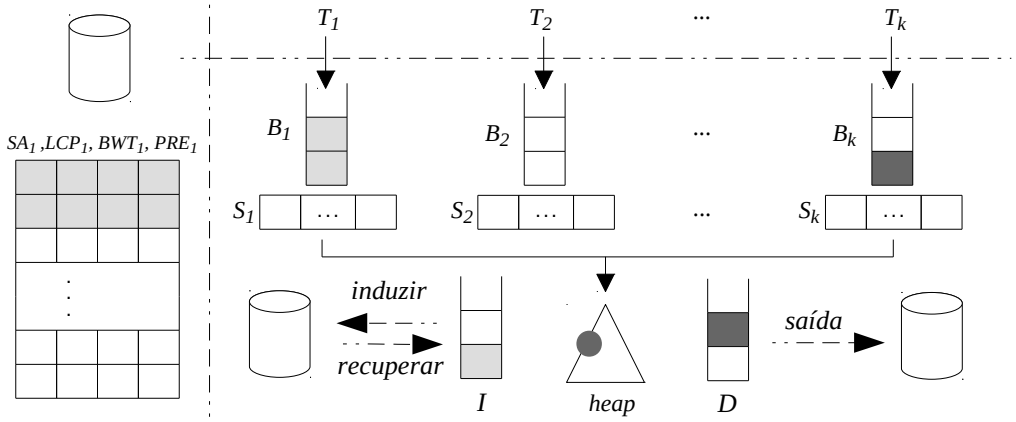

Figura 5.2: Algoritmo eGSA: *buffers* em memória interna

Para construir o vetor *GESA*, inicialmente, cada bloco R_i^1 é carregado no *buffer* B_i. Em seguida, o primeiro elemento (topo) de cada B_i é inserido em uma árvore binária de comparação (*heap*). O menor elemento na *heap* (nó raiz) é inserido na primeira posição vazia de D. Então, o nó raiz é substituído pelo próximo elemento do mesmo *buffer* B_i. Quando um bloco R_i^x é processado, o próximo bloco R_i^{x+1} é carregado em B_i. Quando D fica cheio, todos seus elementos são gravados em memória externa sequencialmente. Essa operação é repetida até que todos os elementos sejam comparados.

De forma geral, no passo m, quando o nó raiz armazena $B_i[j]$ e é inserido em D, os componentes de $D[m]$ são preenchidos com $\langle GSA[m] = (i, SA_i[j]), LCP[m] = l_d, BWT[m] = BWT_i[j] \rangle$. No primeiro passo $l_d = 0$, enquanto que nos outros passos l_d deve ser calculado durante a comparação dos sufixos na *heap*.

A comparação entre os elementos na *heap* constitui a operação mais sensível nessa fase do algoritmo. Essas comparações podem exigir muitos acessos aleatórios à memória externa, já que os vetores SA_i armazenam apenas inteiros e todas as cadeias T_i estão em memória externa. Portanto, para comparar dois sufixos, é necessário acessá-los (conforme necessário) em memória externa.

Para reduzir o número de acessos à memória externa, é proposta nesta dissertação um método melhorado para comparação de sufixos na *heap*, o qual é composto por três estratégias: (i) montagem de prefixo; (ii) comparações de LCP; e (iii) indução de sufixos. Essas estratégias são detalhadas a seguir.

Montagem de Prefixo

Seja j o índice do menor elemento no *buffer* B_i. O vetor PRE_i pode ser utilizado para carregar o prefixo inicial de $T_i[SA_i[j], n_i]$ em S_i sem acessar a cadeia T_i que está em memória externa. Essa operação é feita por meio da concatenação de $PRE_i[j]$ com os prefixos dos sufixos anteriores à $SA_i[j]$, armazenados em $PRE_i[m]$, para $m = 1, 2, \ldots, j-1$. O *buffer* S_i é atualizado conforme $B_i[j]$ é enviado para D e j é incrementado seguindo a Definição 5.3.

Definição 5.3 $S_i[1, h_j + p + 1] = S_i[1, h_j] \cdot PRE_i[j] \cdot \#$, $h_j = min(LCP_i[j], h_{j-1} + p)$ *e* $h_0 = 0$.

O símbolo # é um marcador de fim-de-*buffer*, $\# \notin \Sigma$. Caso a comparação com o sufixo $T_i[SA_i[j], n_i]$ não envolva mais do que $h_j + p$ caracteres, então não é necessário acessar T_i em memória externa. Caso contrário, # é alcançado e T_i é acessado em memória externa na posição $SA_i[j] + h_j + p$.

Por exemplo, na coluna $suff_i$ da Figura 5.1 são destacados em negrito os prefixos recuperados pela estratégia de montagem de prefixos. Na Figura 5.3 é ilustrado a construção de R_1 para $j = 6$. Quando $j = 5$, $h_5 = LCP_1[5] = 0$ e S_1 armazena $GA\$$. Então, em seguida, quando $j = 6$, $h_6 = min(LCP_i[6], h_5 + p) = 2$, e $S_1 = S_1[1, 2] \cdot PRE_1[5] \cdot \# = GA \cdot TAG \cdot \# = GATAG\#$.

j	$SA_1[j]$	$LCP_1[j]$	$BWT_i[j]$	$PRE_1[j]$	$suff_1(j)$
...	...	...	...	...	...
5	5	0	A	GA$	GA$
6	1	2	$	TAG	GATAGA$

S_1	G	A	T	A	G	#	...

Figura 5.3: Estratégia de Montagem de Prefixos

Note que o tamanho do prefixo recuperado é sempre maior ou igual a p, diferente da proposta de Barsky et al. [2009] que sempre recupera prefixos de tamanho p. Caso o tamanho do prefixo recuperado seja maior do que s, o prefixo é truncado na posição s. Entretanto, em geral $s >> h_j + p$.

Comparações de LCP

Sejam X, Y e Z nós da *heap* que armazenam os elementos topos $B_1[x]$, $B_2[y]$ e $B_3[z]$, respectivamente. Conforme ilustrado na Figura 5.4, suponha que o nó X é pai dos nós Y e Z. Então, $X < Y$ e $X < Z$, isto é, $T_1[SA_1[x], n_1] < T_2[SA_2[y], n_2]$ e $T_1[SA_1[x], n_1] < T_3[SA_3[z], n_3]$.

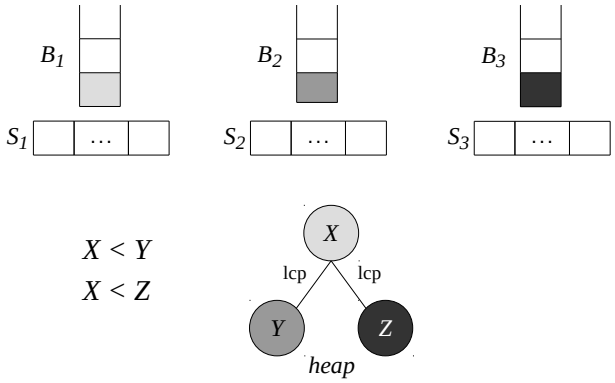

Figura 5.4: Estratégia de Comparação de LCP

Os valores de *lcp* entre nós na *heap* podem ser utilizados para calcular o valor de l_d e otimizar a comparação na *heap* [Ng e Kakehi, 2008]. Essa otimização é baseado no Lema 5.1. A prova do lema é discutida em seguida e ilustrada na Figura 5.5.

Lema 5.1 *Sejam as cadeias S_1, S_2 e S_3, tais que $S_1 < S_2$ e $S_1 < S_3$. Se $lcp(S_1, S_2) > lcp(S_1, S_3)$ então $S_2 < S_3$ (caso 1). Se $lcp(S_1, S_2) < lcp(S_1, S_3)$ então $S_2 > S_3$ (caso 2). Caso contrário, se $lcp(S_1, S_2) = lcp(S_1, S_3) = l$ então $lcp(S_2, S_3) \geq l$ (caso 3).*

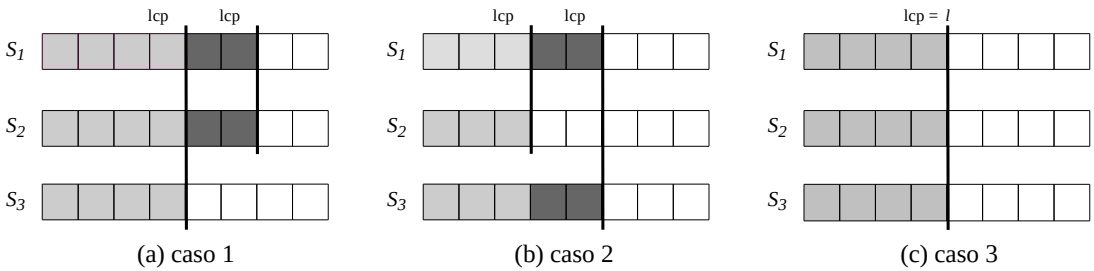

Figura 5.5: Ilustração do Lema 5.1

Prova 5.1 *Dado que S_1 é a menor cadeia, se o valor de lcp $l = lcp(S_1, S_i)$ é máximo e único (casos 1 e 2), então qualquer outra cadeia S_j possui um caractere $S_j[m] > S_i[m]$, para $1 \leq m \leq l$ e, portanto, $S_i < S_j$. No caso 3, o valor l é o mesmo para S_i e S_j e nada pode-se afirmar sobre a ordem de S_i e S_j.* $\square$

A ordem dos sufixos em Y e Z pode ser determinada utilizando o Lema 5.1 (casos 1 e 2). No caso 3, $lcp(X, Y) = lcp(X, Z)$, então $lcp(Y, Z) \geq lcp(X, Y) = l$ e os sufixos em Y e Z podem ser comparados diretamente começando da posição l. No próximo passo do algoritmo de união, o elemento no nó X é removido da *heap* e $B_1[x]$ é movido para D. Então, X é substituído por outro nó W que contém o próximo elemento $B_1[x + 1]$. Essa operação é ilustrada na Figura 5.6.

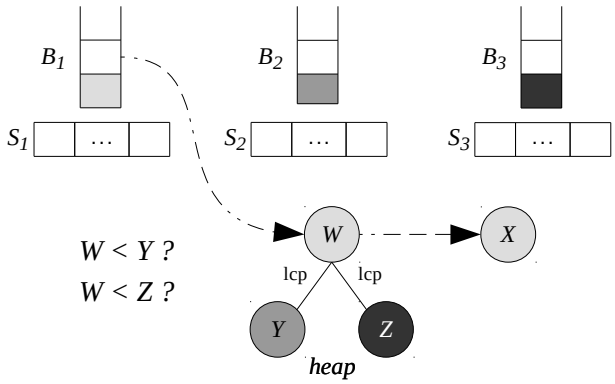

Figura 5.6: Troca de nós na *heap*

A comparação de W com seus filhos pode ser realizada seguindo o Lema 5.1, utilizando o valor de $LCP_1[i + 1] = lcp(X, W)$. Por conveniência, suponha que $Y < Z$, então caso $lcp(X, W) > lcp(X, Y)$ tem-se que $W < Y$ e $l_d = lcp(X, W)$, ou caso $lcp(X, W) < lcp(X, Y)$ tem-se que $W > Y$ e $l_d = lcp(X, Y)$. Nesses casos, nenhuma comparação precisa ser feita entre os sufixos em W e Y e o valor de l_d é atualizado para a próxima iteração. Por fim, caso $lcp(X, W) = lcp(X, Y) = l$, a comparação entre W e Y pode começar a partir do caractere l, atualizando l_d. Dessa forma, muitas comparações de caracteres são evitadas, e por consequência, são evitados acessos à memória externa. No caso de $Y < W$, o nó W deve ser trocado com o nó Y e os valores de *lcp* entre os nós devem ser atualizados. A operação de troca na *heap* continua até que W seja menor que um nó Y'.

Por exemplo, considere a união dos vetores ESA_1 e ESA_2 ilustrados na Figura 3.2, durante a comparação de $ESA_1[4]$ e $ESA_2[3]$. Primeiramente, tem-se que $suff_1(4) = ATAGA\$$ é maior que $suff_2(3) = AGA\$$ e, portanto, $ESA_1[4] > ESA_2[3]$. No próximo passo, são comparados $ESA_1[4]$ e $ESA_2[4]$. Nesse caso, pode-se utilizar a estratégia de comparações de LCP e determinar que, como $LCP_2[4] = lcp(suff_2(3), suff_2(4)) > lcp(suff_2(3), suff_1(4))$, então $suff_2(4) = AGAGA\$$ é menor que $suff_1(4) = ATAGA\$$. Isso pode ser feito, portanto, sem a necessidade de se comparar explicitamente esses sufixos.

Indução de sufixos

A ordenação por indução consiste em determinar a ordem de sufixos não ordenados por meio dos sufixos já ordenados. Essa técnica tem sido empregada por diferentes algoritmos de construção em memória interna [Puglisi et al., 2007] e em memória externa [Bingmann et al., 2013], podendo variar de acordo com o conceito de indução utilizado.

A estratégia de indução de sufixos proposta nesta dissertação é baseada no Lema 5.2, ilustrado na Figura 5.7. A prova do lema é feita por contradição e discutida em seguida.

Lema 5.2 *Seja Π o conjunto de todos os sufixos de $\mathcal{T}$, $T_i \in \mathcal{T}$, $1 \leq j \leq n_i$ e $\alpha \in \Sigma$. Se $T_i[j, n_i]$ é o menor sufixo de Π e $T_i[j-1, n_i] \in \Pi$ então $T_i[j-1, n_i] = \alpha \cdot T_i[j, n_i]$ é o menor α-sufixo em $\Pi \setminus \{T_i[j, n_i]\}$.*

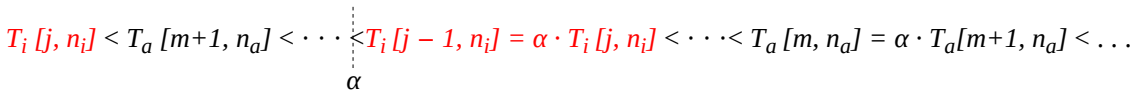

Figura 5.7: Ilustração do Lema 5.2

Prova 5.2 *Suponha que exista um α-sufixo $T_a[m, n_a]$ em Π que seja menor que $T_i[j-1, n_i]$. Então $T_a[m+1, n_a]$ deve ser menor que $T_i[j, n_i]$, e portanto $T_i[j, n_i]$ não é o menor sufixo de Π.* $\square$

O sufixo $T_i[j, n_i]$ pode induzir a ordem do sufixo $T_i[j-1, n_i]$, isto é, determinar a ordem de $T_i[j-1, n_i]$ em Π sem compará-lo. Essa estratégia pode ser utilizada na ordenação dos sufixos de T_i. No Algoritmo 3 é detalhada essa ideia. Inicialmente, é calculado o início de cada α-*bucket* (linhas 2 a 6) por meio de uma leitura sequencial em T_i. Na sequencia, Π recebe todos os sufixos de T_i (linha 7) e conforme o menor sufixo $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$ é encontrado (linha 9), $T_i[j, n_i]$ é removido de Π (linha 10) e inserido na primeira posição disponível do α-*bucket* de SA_i (linha 11). Caso o sufixo $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ ainda não foi ordenado, isto é, se $\alpha < \beta$ (linha 12), o sufixo $T_i[j-1, n_i]$ é induzido para a primeira posição disponível do β-*bucket* de SA_i (linha 13), $\alpha, \beta \in \Sigma$. Esse procedimento é realizado até que Π fique vazio (linha 8).

Note que, os sufixos induzidos $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ não podem ser removidos de Π, pois esses sufixos podem também induzir outros sufixos $T_i[j-2, n_i]$. Entretanto, é possível evitar que os sufixos induzidos sejam comparados durante a escolha do menor

Algoritmo 3: Algoritmo de Ordenação Induzida

Entrada: cadeia T_i de tamanho n_i
Saída: vetor de sufixo SA_i

1 **início**
2 **para** $j \leftarrow 1$ *até* n_i **faça**
3 $freq[T_i[j]] \leftarrow freq[T_i[j]] + 1$; `// frequência de cada` α-$sufixo$ `em` T_i
4 $inicio[\alpha_0] \leftarrow 0$; `// início de cada` α-*bucket* `em` SA_i
5 **para** $j \leftarrow 1$ *até* $|\Sigma|$ **faça**
6 $inicio[\alpha_j] \leftarrow inicio[\alpha_{j-1}] + freq[\alpha_j]$;
7 $\Pi \leftarrow T_i[1, n_1], \ldots, T_i[n_i, n_i]$; `// todos os sufixos de` T_i
8 **enquanto** $\Pi \neq \varnothing$ **faça**
9 $T_i[j, n_i] \leftarrow menor\_sufixo(\Pi)$; `//` $T_i[j, n_i] = \alpha \cdot T_i[j+1, n_i]$
10 $remove(T_i[j, n_i])$;
11 $insere(T_i[j, n_i], \alpha\text{-}bucket)$;
12 **se** $\alpha < \beta$ **então**
13 $insere(T_i[j-1, n_i], \beta\text{-}bucket)$; `// indução de` $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$

sufixo (linha 9). Para isso, quando o primeiro β-$sufixo$ $T_i[j-1, n_i]$ é o menor sufixo de Π, todos os sufixos no β-*bucket* são lidos sequencialmente. Conforme os sufixos $T_i[j-2, n_i]$ são analisados para a indução, os sufixos $T_i[j-1, n_i]$ são removidos de Π. Além disso, se $\alpha = \beta$ então o próximo menor sufixo de Π é $T_i[j-1, n_i]$.

Apesar do Algoritmo 3 evitar a comparação dos sufixos induzidos, o algoritmo não é eficiente para ordenar uma única cadeia T_i, pois sempre precisa encontrar o menor sufixo em Π, o que pode envolver a comparação de todos os sufixos não ordenados de T_i. Entretanto, essa estratégia de indução pode ser bem aplicada na fase de união dos vetores R_i, já que o menor sufixo é sempre um dos k elementos topo dos *buffers* B_i, que estão armazenados na *heap* e podem ser comparados eficientemente a cada iteração.

Então, a estratégia de indução de sufixos aplicada na segunda fase do algoritmo eGSA é a seguinte. Seja Π o conjunto de todos os sufixos de todas as cadeias $T_i \in \mathcal{T}$. Suponha que o nó X está na raiz da *heap* e contém o α-$sufixo$ $T_i[j, n_i]$. Portanto, o sufixo $T_i[j, n_i]$ é o menor sufixo em Π, e o sufixo $T_i[j-1, n_i] = \beta \cdot T_i[j, n_i]$ é induzido se $\alpha < \beta$. Para determinar se $\alpha < \beta$, isto é, se $T_i[j] < T_i[j-1]$, pode-se utilizar o primeiro caractere em S_i, $S_i[1] = T_i[j]$ e o elemento $BWT_i[j] = T_i[j-1]$. Assim, se $S_i[1] < BWT_i[j]$, o sufixo $T_i[j-1, n_i]$ é induzido.

Os sufixos induzidos nesse processo são inseridos no *buffer* de sufixos induzidos I, composto por $|\Sigma|$ *buffers* I_α, um para cada $\alpha \in \Sigma$. Quando um α-$sufixo$ $T_i[j, n_i]$ é

induzido, o valor da cadeia i é inserido na primeira posição disponível do *buffer* I_α, isto é, I_α é um vetor de inteiros que armazena a ordem das cadeias que tiveram α-$sufixos$ induzidos, I_α tem tamanho c e é movido para um arquivo F_α em memória externa conforme fica cheio.

Para evitar que os sufixos induzidos sejam comparados na *heap*, quando o primeiro α-$sufixo$ é inserido em D, o *buffer* I_α é escrito em F_α. A partir do segundo α-$sufixo$, todos os elementos de F_α são carregados, c elementos por vez, em I_α e lidos sequencialmente. A ordem dos valores em I_α determina em qual cadeia i pode se encontrado o próximo menor sufixo $T_i[j-1, n_i]$ de Π. A comparação desses sufixos é ignorada na *heap*, isto é, no passo m o menor sufixo é retirado diretamente do *buffer* B_i, caso $I_\alpha[m \mod c] = i$. Além disso, durante a leitura de I_α, os sufixos $T_i[j-2, n_i] = \beta \cdot T_i[j-1, n_i]$ são induzidos, caso $\alpha < \beta$. Esse procedimento é ilustrado nas operações de induzir e recuperar na Figura 5.2.

Os valores de *lcp* l_d entre os sufixos induzidos enviados para o *buffer* de saída D também devem ser induzidos, já que esses valores não são calculados quando os sufixos induzidos são ignorados na *heap*. A indução de *lcp* é ilustrada na Figura 5.8 e descrita a seguir. Seja $T_a[i, n_a]$ um sufixo que induz um α-$sufixo$ e seja $T_b[j, n_b]$ o sufixo que induz exatamente o próximo α-$sufixo$. O *lcp* entre os dois sufixos induzidos é igual ao *lcp* entre os sufixos que os induziram $T_a[i, n_a]$ e $T_b[j, n_b]$ mais 1, desde que $T_a[i-1] = T_b[j-1] = \alpha$, isto é, $lcp(T_a[i-1, n_a], T_b[j-1, n_b]) = lcp(T_a[i, n_a], T_b[j, n_b]) + 1$. Suponha que $T_a[i, n_a]$ e $T_b[j, n_b]$ estão nas posições x e y de *GESA*, respectivamente, então $lcp(T_a[i, n_a], T_b[j, n_b]) = rmq_{LCP}(x, y)$ (Definição 3.13).

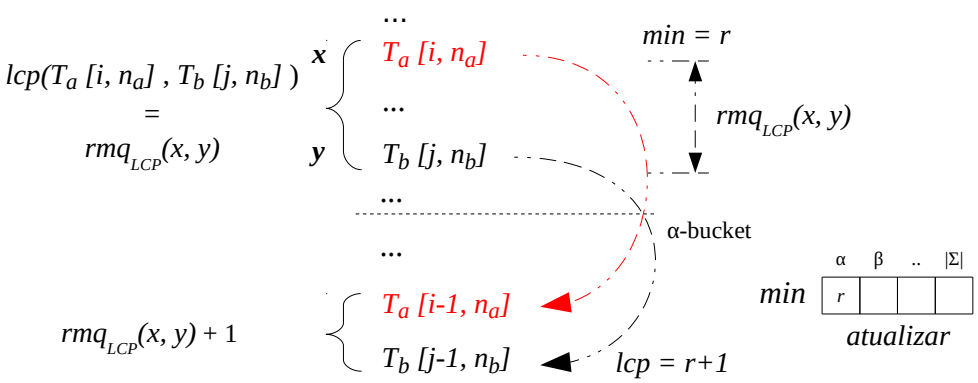

Figura 5.8: Indução de valores de *lcp*

É possível resolver a função $rmq_{LCP}(x, y)$ *online* durante a indução dos sufixos, isto é, sem ter que calcular $min_{x<k\leq y}\{LCP[k]\}$ sempre que um sufixo é induzido. Para isso, é preciso armazenar em um conjunto de $|\Sigma|$ variáveis min_α, uma para cada $\alpha \in \Sigma$, o valor mínimo l_d entre os sufixos inseridos no *buffer* D sempre que dois α-$sufixo$ são induzidos,

da seguinte forma. Quando um α-$sufixo$ $T_a[i-1, n_a]$ é induzido, $min_\alpha \leftarrow max\_int$, ou seja, min_α recebe o valor máximo atribuído para uma variável inteira). Em seguida, nos próximos passos da união, a cada sufixo inserido em D, min_α é atualizado, $min_\alpha \leftarrow min(min_\alpha, l_d)$, até que o próximo α-$sufixo$ $T_b[j-1, n_b]$ seja induzido. Como resultado, $lcp = min_\alpha + 1$ é induzido junto com $T_b[j-1, n_b]$ em I_α e $min_\alpha \leftarrow max\_int$. Portanto, o *buffer* de sufixos induzidos I deve armazenar também um componente para o *lcp*, nesse caso $I_\alpha[m] = (b, min_\alpha)$. Quando o sufixo induzido for recuperado de F_α, o valor de *lcp* l_d, que é enviado para D, recebe o componente *lcp* de $I_\alpha[m]$.

Por exemplo, considere a união dos vetores ESA_1 e ESA_2 ilustrados na Figura 3.2, durante a comparação dos elementos $ESA_1[2]$ e $ESA_2[2]$. Tem-se que $suff_1(2) = A\$$ é menor que $suff_2(2) = A\$$ (conforme a Definição 3.4). Então, o sufixo $T_1[6-1=5, n_1] = GA\$$ é induzido como o menor α-$sufixo$, $\alpha = G$, em Π pois $BWT_1[2] = G > A$. O *buffer* I_α recebe (em sua primeira posição disponível) o par $(1, lcp = 0)$, desde que $l_d = 0$. Em seguida, é calculado o valor de min_α até que o próximo α-$sufixo$ seja induzido, o que acontece no próximo passo, quando $ESA_2[2]$ é o menor elemento na *heap* e $T_1[5, n_1]$ é induzido, então *buffer* I_α recebe o par $(1, lcp = 1 + 1 = 2)$, desde que $l_d = 1$. No final, quando o sufixo $T_2[5, n_2] = GA\$$ for recuperado em I_α, o *lcp* entre $T_1[5, n_1]$ e $T_2[5, n_2]$ já está armazenado em I_α e será igual a 2.

Por fim, a estratégia de montagem de prefixos deve considerar a indução de sufixos, já que os sufixos ignorados na *heap* não participam da montagem no *buffer* S_i (Definição 5.3). Então, para utilizar essas duas estratégias, a montagem deve iniciar exatamente após o primeiro α-$sufixo$ não induzido de T_i. Em seguida, a montagem deve seguir normalmente. Um sufixo $T_i[SA[j], n_i]$ é induzido se $T_i[SA[j]] > T_i[SA[j]+1]$. Então, na primeira fase, durante a construção de R_i, $LCP_i[j+1]$ recebe 0 se $T_i[SA[j]] > T_i[SA[j]+1]$ ($T_i[j, n_i]$ é induzido), de forma que na construção de $PRE_i[j+1]$, h_{j+1} é igual a 0 (Definição 5.2). Isso faz com que $PRE_i[j+1]$ armazene exatamente os p primeiros caracteres de $T_i[SA[j+1], n_i]$, e dessa forma, o prefixo do primeiro α-$sufixo$ é recuperado desde o início durante a montagem.

Por exemplo, na Figura 5.9 é ilustrada a construção de R_1, para $j = 2$, considerando a estratégia de indução em conjunto com a estratégia de montagem. Quando $j = 2$, $SA[j=2] = 6$ e $T_1[6] > T_1[6+1]$, então o sufixo $T_1[6, n_1]$ é induzido e $LCP_1[2+1=3] \leftarrow 0$. Quando $j = 3$, $SA[j=3] = 4$ e $T_1[4] < T_1[4+1]$, então o sufixo $T_1[4, n_1]$ não é induzido. Isso significa que, na segunda fase, durante a montagem de prefixos, o sufixo $T_1[6, n_1]$ será induzido, e portanto ignorado na *heap*, e o sufixo $T_1[4, n_1]$ será montado a partir de seu início em S_1. A partir desse ponto a montagem continua normalmente.

j	$SA_1[j]$	$LCP_1[j]$	$BWT_i[j]$	$PRE_1[j]$	$suff_1(j)$
...	...	...	...	...	
2	6	0	G	A$$	A$
3	4	0	$	AGA	AGA$

S_1	A	G	A	#	#	#	...

Figura 5.9: Montagem de Prefixos considerando Indução de Sufixos

Note que, os valores de $LCP_i[j+1] = 0$ não interferem na construção do vetor *LCP* generalizado, já que os valores de *lcp* também são induzidos. O valor de *lcp* entre o último α-$sufixo$ induzido com o primeiro não induzido é sempre igual a 1. Então, após a recuperação de todos os sufixos em F_α, o próximo α-$sufixo$ a ser inserido em D (primeiro α-$sufixo$ não induzido) tem *lcp* igual a $l_d = 1$.

A comparação dos α-$sufixo$ não induzidos continua por meio da *heap*, até que o primeiro β-$sufixo$ seja encontrado, $\beta > \alpha$. O processo de recuperação dos sufixos induzidos é feito novamente para os β-$sufixos$.

5.2 Testes de Desempenho

O algoritmo eGSA foi analisado por meio testes de desempenho utilizando dados biológicos reais. Na Seção 5.2.1 são descritos experimentos comparativos utilizando o algoritmo eSAIS [Bingmann et al., 2013] com conjuntos de cadeias grandes de DNA. Nesses experimentos a entrada do eSAIS foi adaptada para a construção de vetores generalizados. Em seguida, na Seção 5.2.2 são descritos experimentos com conjuntos de cadeias pequenas de proteínas. Esses experimentos foram realizados apenas com o eGSA, já que no melhor do nosso conhecimento, não existem trabalhos correlatos que possam ser adaptados para conjuntos com muitas cadeias. Na Seção 5.2.3 são investigadas características específicas do eGSA tanto para cadeias de DNA como para cadeias de proteínas.

O algoritmo eGSA foi implementado em ANSI/C e compilado em GNU gcc, versão 4.6.3, com o parâmetro de otimização -O3. Seu código fonte encontra-se disponível em `http://code.google.com/p/egsa/`. Todos os experimentos realizados foram conduzidos em um computador rodando o sistema operacionalDebian GNU/Linux 6.0.3/64 bits, com processador Intel Core i7 3.40 GHz, 8MB L3 cache, 16 GB de memória interna e um disco SATA de 1,5 TB, 7.200 RPM e 32MB cache. O total de memória interna utilizada nos experimentos foi restrito a 4 GB.

5.2.1 Cadeias grandes: DNA

Nessa seção são realizados experimentos com conjuntos de cadeias grandes de caracteres. Foram utilizadas sequências de DNA de diferentes genomas, descritos na Tabela 5.1, obtidos no repositório *Ensembl genome database*[1]. O tamanho em bp (coluna 4) e em GB (coluna 5) de cada genoma corresponde à soma dos tamanhos de todos os cromossomos do organismo. Cada bp referente a um nucleotídeo é representado por um caractere que ocupa 1 *byte* em memória. Nas cadeias, foram removidos todos os nucleotídeos desconhecidos (caractere N).

Tabela 5.1: Genomas utilizados nos experimentos.

Genoma	Organismo	nº cromossomos	Tamanho (bp)	Total (GB)
1	*Homo sapiens*	24	2.855.343.793	2,66
2	*Oryzias latipes*	24	582.126.416	0,54
3	*Danio rerio*	26	1.354.652.750	1,26
4	*Bos taurus*	30	2.640.166.205	2,46
5	*Mus musculus*	21	2.647.521.452	2,47
6	*Gallus gallus*	30	984.855.240	0,92

Os genomas da Tabela 5.1 foram combinados em 5 conjuntos de testes (BDBs) de tamanhos entre 0,54 GB e 8,50 GB com 24 à 105 cadeias de caracteres. Na Tabela 5.2 são detalhados esses conjuntos. Cada cromossomo de um genoma (coluna 2) corresponde a uma cadeia de caracteres no BDB (coluna 3). A maior cadeia de um BDB (coluna 4) corresponde ao tamanho do maior cromossomo dos genomas que compõem o BDB. Os valores de *lcp-médio* e *lcp-máximo* (colunas 5 e 6) indicam a dificuldade da ordenação do conjunto (conforme discutido na Seção 3.3). O tamanho do BDB em GB (coluna 7) corresponde ao tamanho total ocupado em memória por todas as cadeias do conjunto.

Tabela 5.2: Conjuntos de testes - cadeias grandes.

BDB	Genomas	nº cadeias	maior (MB)	*lcp-médio*	*lcp-máximo*	Total (GB)
D_1	2	24	29,37	19	2.573	0,54
D_2	6	30	186,14	17	5.476	0,92
D_3	3, 6	56	186,14	58	71.314	2,18
D_4	2, 3, 4	80	129,90	44	71.314	4,26
D_5	1, 4, 5, 6	105	226,69	59	168.246	8,50

Na primeira fase do algoritmo eGSA, a construção dos vetores SA_i e LCP_i foi realizada utilizando o algoritmo *inducing+sais-lite* [Fischer, 2011], que representa o estado da arte

[1] `http://www.ensembl.org/`

para construção de vetores de sufixo aumentados com *LCP* em memória interna. O tamanho do prefixo utilizado para a construção do vetor PRE_i foi $p = 10$. Na segunda fase, os *buffers* S_i, B_i, D e I foram configurados para utilizar 200 KB, 10 MB, 16 MB e 64 MB de memória interna, respectivamente. Por fim, a saída do algoritmo foi validada utilizando um algoritmo de verificação trivial (*e.g.* [Burkhardt e Kärkkäinen, 2003]).

eGSA vs. eSAIS

Nessa seção são descritos experimentos que comparam o desempenho do eGSA com o algoritmo eSAIS [Bingmann et al., 2013] (descrito na Seção 4.4), que representa o estado da arte para construção de vetores de sufixo em memória externa. Entretanto, o algoritmo eSAIS foi projetado para a construção do vetor de sufixo aumentado com *LCP* em memória externa para uma única cadeia T_i. Para utilizar o eSAIS em conjuntos de cadeias foi adotada nesta dissertação a seguinte estratégia. Todas as cadeias $T_i \in \mathcal{T}$ foram concatenadas em uma única cadeia $T = T_1\$_1 \cdot T_2\$_2 \cdot \ldots \cdot T_k\$_k$, e os símbolos terminais de cada T_i foram substituídos por novos símbolos $\$_i$, de forma que $\$_i < \$_j$ se $i < j$ e $\$_i < \alpha, \forall \alpha \in \Sigma$. Entretanto, conforme discutido na Seção 3.3, essa estratégia limita o número k de cadeias que podem ser indexadas. No caso de sequências de DNA, utilizando variáveis de 1 *byte* para cada caractere, k é limitado a $256 - |\{A, C, G, T\}| = 252$.

Tempo de execução

Na Tabela 5.3 são ilustrados os resultados da execução dos algoritmos eGSA e eSAIS com os conjuntos de teste da Tabela 5.2. Para cada algoritmo são ilustrados os tempos médios de execução em microssegundos por *byte* (coluna 2), isto é, o tempo total de execução dividido pelo tamanho do BDB. Na coluna 3 são ilustrados os tempos de execução totais (*wallclock*) em segundos, e na coluna 4 os tempos de processamento em CPU (*cputime*), isto é, tempo de execução total menos o tempo de espera de I/O. A eficiência de cada algoritmo (coluna 5) corresponde à proporção de tempo de processamento em CPU no tempo de execução total. Por fim, na coluna 6 é calculada a razão do tempo em CPU do algoritmo eGSA pelo eSAIS. Na Tabela 5.3 são destacados em negrito os melhores tempos médio de execução.

Pode-se observar que o algoritmo eGSA foi muito mais rápido em todos os testes (coluna 2), obtendo um tempo médio de 3,2 a 8,3 vezes menor que o tempo do algoritmo eSAIS. Então, pode-se concluir que o eGSA é um algoritmo eficiente na prática para cons-

Tabela 5.3: Resultados para a comparação do eGSA com o eSAIS.

BDB	μs/byte		*wallclock* (seg.)		*cputime* (seg.)		eficiência		razão *cputime*
	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS	eGSA	eSAIS/eGSA
D_1	9,23	**1,12**	5.378	655	1.156	437	0,21	0,66	2,64
D_2	8,23	**1,22**	8.109	1.206	2.032	690	0,25	0,57	2,94
D_3	6,80	**1,51**	15.926	3.533	5.053	2.273	0,32	0,64	2,22
D_4	6,67	**1,58**	30.562	7.264	10.357	4.662	0,34	0,64	2,21
D_5	7,03	**2,23**	64.134	20.365	21.660	10.019	0,33	0,49	2,16

trução de vetores de sufixo generalizados aumentados em memória externa para conjuntos de cadeias grandes. Além disso, diferentemente do eSAIS, o eGSA realiza a construção do vetor *BWT* generalizado sem custo adicional, e pode ser aplicado para a construção de vetores generalizados para conjuntos com mais de 252 cadeias, no caso de sequências de DNA.

A eficiência dos algoritmos (coluna 5) indica o proporção do *cputime* no tempo total. O algoritmo eGSA foi mais eficiente do que o eSAIS em todos os casos de teste. Pode-se observar que a eficiência do eGSA diminui no BDB D_5. Isso ocorre não somente pelo tamanho do BDB mas também pelos valores de *lcp-médio* e *lcp-máximo* do conjunto, que aumentam, respectivamente, 1,34 e 2,35 vezes do BDB D_4 para o BDB D_5. Entretanto, mesmo apresentando essa redução, o algoritmo proposto ainda é mais eficiente do que o eSAIS quando considerado o BDB D_5.

Pode-se observar que a razão do tempo de CPU para os dois algoritmos nos conjuntos D_4 e D_5 é próxima (coluna 6), o que indica que o tempo de espera de I/O está provavelmente relacionado com essa perda de eficiência.

Memória interna utilizada

Na Tabela 5.4 é descrita a quantidade de memória utilizada em cada fase do algoritmo eGSA. Na primeira fase, a memória interna máxima utilizada (coluna 2) é determinada pelo algoritmo utilizado para a construção de SA_i e LCP_i para a maior cadeia do BDB (valor máximo de $|T_i|$). Na segunda fase, o total de memória (coluna 7) corresponde à soma de todos os *buffers*: $S = \sum_{i=1}^{k} S_i$, $B = \sum_{i=1}^{k} B_i$, $I = \sum_{\alpha_j=1}^{|\Sigma|} I_{\alpha_j}$ e D.

A quantidade máxima de memória utilizada na primeira fase foi de 1,99 GB para a construção do vetor de sufixo da maior cadeia do BDB D_5, a qual possui tamanho de 226,69 MB. Na segunda fase, a quantidade de memória utilizada depende do número de cadeias do BDB (*buffers* S e B) e do tamanho do alfabeto (*buffer* I). Nos experimentos

Tabela 5.4: Memória utilizada pelo eGSA.

BDB	primeira fase Total (GB)	segunda fase (MB)				
		S	B	I	D	Total
D_1	0,26	2,29	240,00	16,00	64,00	320,00
D_2	1,72	2,86	300,00	16,00	64,00	380,00
D_3	1,72	5,34	560,00	16,00	64,00	640,00
D_4	1,14	7,63	800,00	16,00	64,00	880,00
D_5	1,99	10,01	1.050,00	16,00	64,00	1.130,00

realizados, a quantidade máxima de memória utilizada foi de 1,1 GB para o BDB D_5, o qual contém $k = 105$ cadeias.

A quantidade de memória interna utilizada pelo algoritmo eSAIS foi de 4 GB em todos os testes. Em particular, esse valor é um parâmetro utilizado pelo eSAIS no início da execução do algoritmo. Portanto, pode-se perceber que o algoritmo proposto eGSA consome muito menos memória do que o algoritmo eSAIS.

5.2.2 Cadeias pequenas: proteínas

Nessa seção são realizados experimentos com conjuntos de cadeias de caracteres pequenas. Foram utilizadas sequências de proteínas de diferentes organismos, obtidas no repositório *Uniprot database*[2]. Foram gerados 5 conjuntos de teste, de tamanhos entre 0,22 GB e 3,64 GB com 0,7 a 11 milhões de cadeias. Cada bp referente a um aminoácido é representado por um caractere que ocupa 1 *byte* em memória. Nas cadeias, foram considerados todos os aminoácidos listados na Tabela 2.2 (isto é, $|\Sigma| = 20$).

Na Tabela 5.5 são detalhados os conjuntos de teste utilizados. A coluna 2 indica a quantidade de cadeias de cada BDB. O tamanho da menor cadeia, da maior cadeia, e o tamanho médio das cadeias em um BDB são indicados nas colunas 3, 4 e 5, respectivamente. Os valores de *lcp-médio* e *lcp-máximo* (colunas 6 e 7) indicam a dificuldade da ordenação do conjunto (conforme discutido na Seção 3.3). O tamanho de um BDB em GB (coluna 8) corresponde ao tamanho total ocupado em memória por todas as cadeias do conjunto.

Na primeira fase do algoritmo eGSA, devido ao grande número k de cadeias em $\mathcal{T}$ (conforme discutido na Seção 5.1.1), o conjunto $\mathcal{T}$ foi dividido em r partições de tamanhos iguais. Em seguida, para cada partição $\mathcal{T}^i$ de $\mathcal{T}$, a construção do vetor de sufixo

[2] `http://www.uniprot.org/`

Tabela 5.5: Conjuntos de testes - cadeias pequenas.

BDB	nº cadeias	menor (bp)	maior (bp)	média (bp)	*lcp-médio*	*lcp-máximo*	Total (GB)
A_1	0,7M	78	12.220	344	26	72	0,22
A_2	1,1M	78	12.220	378	42	72	0,38
A_3	2,3M	70	36.686	398	48	88	0,85
A_4	4,8M	62	36.686	403	63	107	1,80
A_5	11,2M	62	36.686	348	65	167	3,64

generalizado GSA_i foi realizada utilizando o algoritmo SAIS [Nong et al., 2011] adaptado para a construção de vetores de sufixo generalizados[3] em memória interna [Simpson e Durbin, 2010]. Por fim, o vetor LCP_i generalizado foi construído utilizando o algoritmo de Kasai et al. [2001]. Em conjunto, esses algoritmos utilizam $13 \times n$ *bytes* de memória interna, onde n é o tamanho da partição $\mathcal{T}^i$. A segunda fase do algoritmo segue de forma considerando que o vetor composto R_i (Definição 5.1) contém o vetor generalizado GSA_i ao invés de SA_i.

O tamanho do prefixo utilizado para a construção do vetor PRE_i foi $p = 5$. Na segunda fase, os *buffers* S_i, B_i, D e I foram configurados para utilizar 200 *bytes*, 10 MB, 16 MB e 64 MB de memória interna, respectivamente. O tamanho do *buffer* S_i foi reduzido devido ao tamanho médio das cadeias nos BDB. Por fim, a saída do algoritmo foi validada utilizando um algoritmo de verificação trivial (*e.g.* [Burkhardt e Kärkkäinen, 2003]).

Eficiência do algoritmo eGSA

No melhor de nosso conhecimento, não existem trabalhos correlatos na literatura que possam ser adaptados para a construção de vetores de sufixo generalizados para conjuntos com muitas cadeias de caracteres. A estratégia utilizada na Seção 5.2.1 para adaptar a entrada do algoritmo eSAIS não pode ser utilizada nessa seção, devido ao número k de cadeias em $\mathcal{T}$ ser muito grande (conforme discutido na Seção 3.3). Dessa forma, os experimentos de construção de vetores de sufixo generalizados para conjuntos de cadeias pequenas consideram apenas o tempo de execução e a memória interna utilizada pelo algoritmo eGSA.

[3]disponível em `https://github.com/jts/sga/blob/master/src/SuffixTools/`

Tempo de execução

Na Tabela 5.6 são descritos os resultados da execução do algoritmo eGSA com os conjuntos de teste da Tabela 5.5. Na coluna 2 são descritos o número de partições utilizadas em cada BDB. Para possibilitar a comparação entre os tempos de construção para BDBs de cadeias pequenas com os tempos para BDBs de cadeias grandes, cada conjunto de teste foi dividido com o mesmo número de partições referentes ao número de cadeias dos BDBs da Tabela 5.2. Os tempos médios de execução em microssegundos por *byte* são ilustrados na coluna 3. Nas colunas 4 e 5 são ilustrados os tempos em segundos de execução totais (*wallclock*) e de processamento em CPU (*cputime*), respectivamente. Por fim, a coluna 6 mostra eficiência do algoritmo.

Tabela 5.6: Resultados experimentais do eGSA.

BDB	nº partições	μs/byte	*wallclock*	*cputime*	eficiência
A_1	24	**0,55**	134,00	114,39	0,86
A_2	30	**0,62**	260,00	228,92	0,88
A_3	56	**0,82**	757,00	576,79	0,76
A_4	80	**1,02**	1.986,00	1.331,22	0,67
A_5	105	**1,42**	5.617,00	2.931,65	0,52

Pode-se observar que os tempos para BDBs de cadeias pequenas são competitivos. Então, além do eGSA ser o único algoritmo que pode ser aplicado para esses BDBs, o eGSA é um algoritmo eficiente para a construção de vetores de sufixo generalizados aumentados em memória externa para conjuntos de cadeias pequenas.

Além disso, assim como nos experimentos com BDBs de DNA (Seção 5.2.1), conforme aumenta o tamanho do BDB, a eficiência do eGSA diminui. Isso ocorre não somente pelo tamanho do BDB mas também pelos valores de *lcp-médio* e *lcp-máximo* do conjunto, que aumentam, respectivamente, 1,03 e 1,57 vezes do BDB A_4 para o BDB A_5.

Memória interna utilizada

Na Tabela 5.7 é descrita a quantidade de memória utilizada em cada fase do algoritmo eGSA. Na primeira fase, a memória interna máxima utilizada (coluna 2) é determinada pelos algoritmos utilizados para a construção de GSA_i e LCP_i para a maior partição do BDB. Na segunda fase, o total de memória (coluna 7) corresponde à soma de todos os *buffers*: $S = \sum_{i=1}^{r} S_i$, $B = \sum_{i=1}^{r} B_i$, $I = \sum_{\alpha_j=1}^{|\Sigma|} I_{\alpha_j}$ e D, onde r é o número de partições. Nesse caso, o alfabeto utilizado Σ possui tamanho 20.

Tabela 5.7: Memória utilizada pelo eGSA.

BDB	primeira fase Total (GB)	segunda fase (MB)				
		S	B	I	D	Total
A_1	0,27	0,004	240,00	16,00	64,00	320,00
A_2	0,43	0,005	300,00	16,00	64,00	380,00
A_3	0,54	0,010	560,00	16,00	64,00	640,00
A_4	0,68	0,015	800,00	16,00	64,00	880,00
A_5	1,03	0,020	1.050,00	16,00	64,00	1.130,00

A quantidade máxima de memória utilizada na primeira fase foi de 1,03 GB para a construção da maior partição do BDB A_5, a qual possui tamanho de 85,89 MB. Na segunda fase, a quantidade de memória utilizada depende do número de partições r (*buffers* S e B) e do tamanho do alfabeto (*buffer* I). Nos experimentos realizados, a quantidade máxima de memória utilizada foi de 1,1 GB para o BDB A_5 o qual contém $r = 105$ partições. Note que esse valor foi o mesmo do teste com cadeias grandes de DNA (Subseção 5.2.1), já que o particionamento seguiu o mesmo número de cadeias na Tabela 5.2.

5.2.3 Características específicas do eGSA

Nessa seção são discutidos experimentos que avaliam características específicas do algoritmo eGSA utilizando conjuntos de teste de cadeias grandes de DNA, descritos na Tabela 5.2, e conjuntos de cadeias pequenas de proteínas, descritos na Tabela 5.5. No primeiro teste é investigado o melhor valor para o parâmetro p para cada conjunto de teste. Em seguida, é investigada a quantidade de sufixos que são induzidos em cada experimento. Por fim, são avaliados os efeitos de cada estratégia utilizada na *heap*, separadamente e em conjunto, no tempo de execução do algoritmo.

Tamanho do vetor de prefixos

Para determinar o melhor valor para o parâmetro p nos BDBs de DNA e de proteínas, foram avaliados diferentes valores para p considerando a mesma quantidade de memória interna utilizada, isto é, conforme p aumenta o número de elementos em B_i diminui, de forma a manter o mesmo valor de memória ocupada por $B = \sum_{i=1}^{k} B_i$, descritos nas Tabelas 5.4 e 5.7 para os BDBs de DNA e proteínas, respectivamente.

Nas Tabelas 5.8 e 5.9 são descritos o efeito do tamanho do valor de p no tempo médio do algoritmo para os BDBs de DNA e proteínas, respectivamente. O tamanho de $p = 0$

(coluna 2) significa que a estratégia de montagem não foi utilizada pelo algoritmo. Nas Tabelas 5.8 são destacados em negrito os melhores tempos médio de execução.

Tabela 5.8: Tamanho do vetor de prefixos - DNA.

BDB	tempo em μs/byte					
	$p = 0$	$p = 5$	$p = 10$	$p = 15$	$p = 20$	$p = 25$
D_1	2,12	1,22	**1,12**	1,23	1,33	1,47
D_2	2,05	1,26	**1,22**	1,51	1,71	2,13
D_3	2,37	1,65	**1,51**	1,80	1,98	2,18
D_4	2,63	1,88	**1,58**	1,72	2,00	2,24
D_5	3,02	2,56	**2,23**	2,95	3,49	3,56

Tabela 5.9: Tamanho do vetor de prefixos - proteínas.

BDB	tempo em μs/byte					
	$p = 0$	$p = 5$	$p = 10$	$p = 15$	$p = 20$	$p = 25$
A_1	0,77	**0,55**	0,59	0,58	0,58	0,60
A_2	0,87	**0,62**	0,66	0,76	0,89	0,97
A_3	1,00	**0,82**	1,03	1,15	1,55	1,21
A_4	1,11	**1,02**	1,34	1,45	1,59	1,76
A_5	**1,30**	1,42	1,81	1,96	2,11	2,46

Para os BDBs de DNA, o valor de $p = 10$ foi o melhor em todos os experimentos, enquanto que para os BDBs de proteínas o valor de $p = 5$ foi o melhor em quase todos os experimentos. Apenas para o BDB A_5 o algoritmo sem a montagem de prefixos obteve um melhor desempenho (9% mais rápido do que com o valor $p = 5$).

Para os BDBs de DNA, pode-se concluir que, conforme o tamanho de p aumenta, o prefixo montado aumenta e melhora o desempenho do algoritmo até certo ponto, no qual a quantidade de elementos em B_i diminui fazendo com que o número de acessos à memória externa para carregar as partições de R_i tenha um impacto no desempenho do algoritmo. Por exemplo, para o BDB D_5 quando $p = 10$ o número de elementos em cada B_i é igual a 551.882 e quando $p = 20$ esse número é reduzido para 361.577 elementos. No caso dos BDBs de proteínas (para o BDB A_5), devido ao tamanho médio das cadeias, o custo da estratégia de montagem de prefixos é superior à quantidade de comparações que são evitadas.

Sufixos induzidos

Para avaliar o efeito da estratégia de indução de sufixos, foi calculada a quantidade de sufixos induzidos durante a execução do algoritmo eGSA para os BDBs de DNA e proteínas. Nas Tabelas 5.10 e 5.11 são descritas a porcentagem dos α-$sufixos$, $\forall\alpha \in \Sigma$, induzidos em relação ao total de α-$sufixos$ nos BDBs.

Na Tabela 5.10 a porcentagem dos sufixos que começam com "A" é muito pequena (coluna 2), e foi arredondada para zero. Esses sufixos podem ser induzidos apenas por sufixos que começam com \$. Nas Tabelas 5.10 e 5.11, o total de sufixos induzidos (coluna 6) corresponde à soma de todas as porcentagens de cada conjunto.

Tabela 5.10: Sufixos induzidos - DNA.

BDB	porcentagem de sufixos induzidos				
	A	C	G	T	Total
D_1	0,00	37,24	50,44	64,18	36,80
D_2	0,00	37,18	52,15	67,14	38,16
D_3	0,00	38,73	51,03	65,93	37,56
D_4	0,00	36,77	50,07	66,51	37,34
D_5	0,00	35,66	49,99	67,69	37,58

Tabela 5.11: Sufixos induzidos - proteínas.

BDB	porcentagem de sufixos induzidos				
	{A, C, D, E, F}	{G, H, I, J, L}	{M, N, P, Q, R}	{S, T, V, W, Y}	Total
A_1	9,68	36,96	61,66	81,42	46,31
A_2	9,66	36,35	61,13	83,14	46,23
A_3	9,64	35,86	60,91	83,06	46,15
A_4	9,43	35,36	60,88	82,83	46,01
A_5	9,52	36,46	62,19	83,66	46,20

Pode-se observar que muitos de sufixos são induzidos, em média $37,49\%$ dos sufixos nos BDBs de DNA, e $46,18\%$ dos sufixos nos BDBs de proteínas. No caso de BDBs de proteínas, a quantidade de sufixos induzidos foi maior devido ao tamanho do alfabeto ser 5 vezes maior que o alfabeto de DNA. Isso significa que essa melhoria proposta para o algoritmo eGSA faz com que muitas comparações sejam evitadas. A contagem de sufixos induzidos pode ser feita em um preprocessamento, utilizando a regra de que um sufixo $T_i[SA[j], n_i]$ é induzido se $T_i[SA[j]] > T_i[SA[j] + 1]$. Esses valores podem ser utilizados, por exemplo, para determinar tamanhos variáveis para cada *buffer* I_α.

Efeito de cada estratégia da *heap*

Para avaliar o efeito de cada estratégia de comparação utilizada na *heap* nos BDBs de DNA e de proteínas, foram desenvolvidas 8 versões do algoritmo eGSA, uma para cada possível combinação das estratégias (a) montagem de prefixo, (b) comparações de LCP e (c) indução de sufixos.

Nas Tabelas 5.12 e 5.13 são descritos os tempos médios de execução (em μs/byte) para cada versão do algoritmo para os BDBs de DNA e de proteínas, respectivamente. Na coluna 2 são indicados os tempos para a versão sem nenhuma estratégia. Nas colunas 3, 4 e 5 são indicados os tempos das versões com apenas uma estratégia, e nas colunas 6, 7 e 8 os tempos das versões com duas estratégias combinadas. Por fim, na coluna 9 são indicados os tempos para a versão completa do eGSA, com todas as estratégias. Nas Tabelas 5.12 e 5.13 são destacados em negrito os melhores tempos médio de execução.

Tabela 5.12: Efeito de cada estratégia da *heap* - DNA.

BDB	μs/byte							
	$\emptyset$	{a}	{b}	{c}	{a, b}	{a, c}	{b, c}	{a, b, c}
D_1	3,04	1,29	2,80	2,14	1,22	1,28	2,12	**1,12**
D_2	3,05	1,28	2,90	2,18	1,31	1,43	2,05	**1,22**
D_3	3,52	1,87	3,17	2,53	1,70	1,78	2,37	**1,51**
D_4	3,65	2,63	3,36	2,73	1,80	1,79	2,63	**1,58**
D_5	3,71	3,00	3,78	3,19	2,45	2,52	3,02	**2,23**

Tabela 5.13: Efeito de cada estratégia da *heap* - proteínas.

BDB	μs/byte							
	$\emptyset$	{a}	{b}	{c}	{a, b}	{a, c}	{b, c}	{a, b, c}
A_1	1,16	0.72	1,08	0.82	0,63	0,60	0,77	**0,55**
A_2	1,33	0,91	1,25	0,91	0,79	0,73	0,87	**0,62**
A_3	1,55	1,19	1,42	1,09	1,04	0,92	1,00	**0,82**
A_4	1,73	1,54	1,55	1,28	1,39	1,33	1,11	**1,02**
A_5	1,98	1,96	1,80	1,40	1,95	1,96	**1,30**	1,42

Para os BDBs de DNA (Tabela 5.12), a versão completa do algoritmo (coluna 8) foi a melhor em todos os experimentos, enquanto que para os BDBs de proteínas (Tabela 5.13), apenas no BDB A_5, a versão completa não foi a melhor. Comparada com a versão sem estratégia (coluna 2), a versão completa foi de 1,6 à 3,7 vezes mais rápida nos BDBs de DNA, e 1,3 à 2,1 vezes mais rápida nos BDBs de proteínas. Além disso, todas as

estratégias, individualmente, melhoraram o desempenho do algoritmo com relação ao algoritmo sem nenhuma estratégia em todos os casos.

A estratégia de montagem de prefixo (presente nas colunas 3, 6, 7 e 9) foi a que proporcionou a maior redução no tempo de execução para os BDBs de DNA (Tabela 5.12). Enquanto que para os BDBs de proteínas (Tabela 5.13), a estratégia de indução de sufixos (presente nas colunas 5, 7, 8 e 9) foi a que, no geral, proporcionou a maior redução no tempo de execução.

Pode-se observar que, em geral, a redução do tempo nos BDBs de proteínas foi menor do que nos BDBs de DNA. Acredita-se que esse fato esteja relacionado com o pequeno tamanho das cadeias de proteínas, o que faz com que poucas comparações de caracteres sejam feitas durante a união dos vetores de sufixo, fazendo com que o efeito das estratégias da *heap* sejam minimizados.

Por fim, os resultados descritos nas Tabelas 5.12 e 5.13 comprovam que, em geral, o uso de diferentes estratégias na *heap* contribui para melhorar o desempenho do algoritmo proposto. Uma observação importante a ser feita é que, para os BDBs de DNA (Tabela 5.12), mesmo que o algoritmo proposto seja executado sem a adição de nenhuma estratégia (coluna 2), ele ainda apresenta desempenho superior ao desempenho do algoritmo correlato eSAIS (Tabela 5.3).

5.3 Análise de Complexidade

Nessa seção são realizadas análises teóricas de tempo e espaço e o volume de operações de I/O das duas fases do algoritmo eGSA. As análises são feitas considerando conjuntos de cadeias grandes, isto é, no caso aonde o conjunto $\mathcal{T}$ não é particionado. Para analisar o caso em que o conjunto de cadeias é particionado, deve-se considerar cada partição T^i como uma cadeia de $\mathcal{T}$ e a quantidade de cadeias k igual ao número de partições r.

Na primeira fase do eGSA, o custo teórico para construir o vetor composto R_i é determinado pelo algoritmo utilizado na construção dos vetores SA_i e LCP_i, já que os valores de $BWT[j]$ e $PRE[j]$ são obtidos durante a escrita de $R_i[j]$ em memória externa, isto é, em tempo linear e com memória constante. Nos experimentos realizados, o algoritmo utilizado para a construção de SA_i e LCP_i foi o *inducing+sais-lite*, que consome tempo $O(|T_i|)$ e utiliza $O(9 \times |T_i|)$ de espaço em memória interna [Fischer, 2011]. Portanto, considerando T_i a maior cadeia do conjunto $\mathcal{T}$, a complexidade assintótica de tempo da primeira fase é $O(k \times |T_i|)$ mais o tempo relacionado às operações de I/O, que possui

complexidade $O(N)$, já que cada vetor R_i é escrito sequencialmente na memória externa. A complexidade assintótica de espaço é $O(|T_i|)$, dado que cada R_i é construído por vez.

Na segunda fase do eGSA, o custo teórico para unir todos os vetores R_i é dominado pelo número de comparações feitas na *heap*. A quantidade de operações de troca na *heap*, necessárias para manter a *heap* consistente, é limitado por $N \log k$, onde N é a soma do tamanho das k cadeias indexadas. Cada operação de troca na *heap* exige a comparação de, no máximo, *lcp-máximo* caracteres (*maxlcp*). Entretanto, no caso médio, o número de comparações é próximo do valor de *lcp-médio*. Portanto, a complexidade assintótica de tempo dessa fase é $O((N \log k) \times maxlcp)$, além das operações de I/O.

O volume de operações de I/O na segunda fase é dominado pelo número de operações de leitura das r_i partições de R_i nos *buffers* B_i, para toda cadeia $T_i \in \mathcal{T}$, mais as operações de escrita do *buffer* de saída D. Considerando que cada cadeia T_i tem tamanho médio igual a $m = N/k$, e dado que o tamanho de cada partição R_i^j é igual a b, então o número r_i de partições de um vetor R_i é igual a m/b. Dessa forma, o número total de partições de todos os vetores $R_1, R_2, \ldots R_k$ é igual à $k \times (m/b) = k \times (N/(k \times b)) = N/b$. A quantidade de operações de escrita em memória externa é proporcional ao número de sufixos ordenados pelo tamanho do *buffer* D, isto é, N/d. Portanto, a complexidade de I/Os da segunda fase é $O(N/b + N/d)$.

A complexidade de espaço da segunda fase é determinada pelo número de cadeias do conjunto que determina o número de *buffers* utilizados e a quantidade de nós na *heap*. Para cada cadeia $T_i \in \mathcal{T}$, são utilizados dois *buffers* S_i e B_i, de tamanhos s e b, respectivamente. Além disso, são utilizados dois *buffers* gerais I e D, de tamanhos $c \times |\Sigma|$ e d, respectivamente. Então o espaço utilizado pelos *buffers* é igual a $O(k \times (s + b) + c \times |\Sigma| + d)$. Considerando a quantidade de nós na *heap*, que é da ordem de $O(k)$, a complexidade final de espaço é $O(k \times (s + b + 1) + c \times |\Sigma| + d)$.

5.4 Considerações Finais

Nesse capítulo foi apresentado o algoritmo eGSA, o primeiro algoritmo proposto para construção de vetores de sufixo generalizados aumentados com os vetores LCP e BWT em memória externa. Inicialmente, na Seção 5.1 foi descrita a proposta geral do algoritmo, e em seguida foram detalhadas suas fases de ordenação em memória interna (Seção 5.1.1) e união em memória externa (Seção 5.1.2). Foram descritos testes de desempenho com o algoritmo eGSA (Seção 5.2), nos quais foram consideradas cadeias grandes de DNA

(Seção 5.2.1) e cadeias pequenas de proteínas (Seção 5.2.2). O algoritmo correlato eSAIS foi utilizado como base de comparação nos BDBs de cadeias grandes de DNA. O algoritmo proposto eGSA obteve um melhor desempenho em todos os testes realizados. O ganho de desempenho variou de 3,2 a 8,3 vezes. Para os conjuntos de cadeias pequenas, foram realizados testes apenas com o eGSA. Comparado com o tempo médio para conjuntos de cadeias grandes, o eGSA obteve tempos competitivos para conjuntos de cadeias pequenas. Dessa forma, os resultados dos testes demonstraram que o algoritmo proposto pode ser aplicado eficientemente para indexar tanto conjuntos de cadeias grandes como conjuntos de cadeias pequenas. Em seguida, foram realizados testes para avaliar características específicas do eGSA (Seção 5.2.3). Por fim, na Seção 5.3 foram feitas análises teóricas de tempo e espaço e o volume de operações de I/O das duas fases do algoritmo eGSA.

No Capítulo 6 são discutidas as principais contribuições desta dissertação, listadas as publicações resultantes do trabalho realizado e são listadas sugestões para trabalhos futuros.

Capítulo

6

Conclusões

O vetor de sufixo é uma estrutura de dados importante utilizada em diferentes problemas que envolvem cadeias de caracteres. Na literatura, trabalhos que consideram a construção de vetores de sufixo voltados à indexação de conjuntos de cadeias são direcionados apenas à memória interna (Capítulo 3), enquanto que trabalhos voltados à construção em memória externa consideram a indexação de apenas uma cadeia (Capítulo 4). Em outras palavras, não existem trabalhos para construção de vetores de sufixos generalizados em memória externa. Esta dissertação supre essa limitação existente na literatura, avançando o estado da arte por meio da proposta do algoritmo eGSA (Capítulo 5), o primeiro algoritmo voltado à construção de vetores de sufixo generalizados aumentado com os vetores LCP e BWT em memória externa.

O algoritmo eGSA realiza a construção do vetor de sufixo generalizado aumentado com os vetores LCP e BWT para um conjunto de cadeias de caracteres. Na primeira fase do algoritmo, para conjuntos de cadeias grandes, são construídos vetores individuais para cada cadeia, os quais são armazenados em memória externa. No caso de conjuntos de cadeias pequenas, o conjunto de cadeias é particionado e são construídos vetores generalizados para cada partição. Da mesma forma, esses vetores são armazenados em memória externa. Em seguida, na segunda fase, todos os vetores construídos são unidos utilizando um método melhorado de comparação de sufixos em memória externa composto por três estratégias: (i) montagem de prefixo; (ii) comparações de LCP; e (iii) indução de su-

fixos. No final, o vetor de sufixo generalizado aumentado com os vetores LCP e BWT é armazenado em memória externa.

A validação do algoritmo eGSA foi realizada por meio de testes de desempenho com conjuntos de cadeias grandes (DNA) e conjuntos de cadeias pequenas (proteínas). Foram utilizados BDBs reais de tamanhos de até 8,5 GB com 105 sequências de DNA e 11 milhões de sequências de proteínas. Para os conjuntos de cadeias grandes, foram realizados testes comparativos com o algoritmo eSAIS [Bingmann et al., 2013], o qual representa o trabalho correlato mais eficiente disponível na literatura para construção de vetores de sufixo. Nesses experimentos a entrada do algoritmo eSAIS foi adaptada para a construção de vetores generalizados. O algoritmo eGSA obteve um tempo médio de 3,2 a 8,3 vezes menor do que o eSAIS e um consumo de memória 50% menor. Para os conjuntos de cadeias pequenas, foram realizados testes apenas com o eGSA, desde que no melhor do nosso conhecimento, não existem trabalhos correlatos que possam ser adaptados para conjuntos com muitas cadeias. Comparado com o tempo médio para conjuntos de cadeias grandes, o eGSA obteve tempos competitivos para conjuntos de cadeias pequenas. Dessa forma, os resultados dos testes demonstraram que o algoritmo proposto pode ser aplicado eficientemente para indexar tanto conjuntos de cadeias grandes quanto conjuntos de cadeias pequenas. Adicionalmente, testes realizados com cada estratégia comprovaram que o uso conjunto dessas estratégias melhora o desempenho do algoritmo como um todo.

Outra vantagem do algoritmo eGSA é que ele pode ser utilizado para a construção de vetores generalizados a partir de vetores de sufixo (individuais) construídos anteriormente para cada cadeia de um conjunto. Em adicional, o eGSA pode ser utilizado para construir diferentes estruturas de dados, como o LOF-SA [Sinha et al., 2008; Moffat et al., 2009], a árvore de sufixo generalizada [Barsky et al., 2008] e o vetor LPF (*Longest Previous Factor array*) [Crochemore et al., 2013].

Além das contribuições apresentadas nesta dissertação com a proposta do algoritmo eGSA, no Apêndice A é investigado o uso de vetores de sufixo no problema de montagem de genomas *de-novo*. Nesse apêndice é apresentado um algoritmo preliminar, chamado de SG2L, voltado à construção de grafos de cadeias utilizando vetores de sufixo generalizados. O algoritmo é proposto apenas teoricamente e sua fase de implementação e testes de desempenho estão listadas como trabalhos futuros no Apêndice A.

6.1 Contribuições

As principais contribuições decorrentes desta dissertação são:

- Proposta do algoritmo eGSA para construção de vetores de sufixo generalizados aumentados em memória externa.
- Validação do algoritmo eGSA com conjuntos de cadeias pequenas e cadeias grandes, evidenciando um avanço no estado da arte.
- Revisão bibliográfica de algoritmos para construção de vetores de sufixo em memória externa.

Adicionalmente, considerando-se o período de desenvolvimento do projeto, foram realizadas as seguintes publicações:

1. Louza, F. A., Telles, G. P. e Ciferri, C. D. A. . External Memory Generalized Suffix and LCP Arrays Construction. In: Proceedings of 24th Annual Symposium on Combinatorial Pattern Matching (CPM), 2013, Bad Herrenalb, Germany, p. 201-210. Classificação QUALIS Ciência da Computação B1.
2. Louza, F. A., Hoffmann, S., Stadler, P. F., Telles, G. P. e Ciferri, C. D. A. . Suffix Arrays and Genome Assembly. In: Digital Proceedings of the 8th Brazilian Symposium on Bioinformatics (BSB) 2013, Recife, Brazil, p. 1-1.
3. Louza, F. A. e Ciferri, C. D. A.. Indexação de Dados Biológicos baseada em Vetores de Sufixo Generalizados para Disco. In: Anais do Workshop de Teses e Dissertações em Banco de Dados do 27º Simpósio Brasileiro de Banco de Dados (SBBD), 2012, São Paulo, Brasil. p. 87-92.
4. Chino, D. Y. T., Louza, F. A., Traina, A. J. M., Ciferri, C. D. A. e Traina Jr., C.. Time Series Indexing Taking Advantage of the Generalized Suffix Tree. Journal of Information and Data Management - JIDM, v. 3, p. 101-109, 2012. Classificação QUALIS Ciência da Computação B3.

O primeiro trabalho refere-se a um artigo completo em evento com os resultados obtidos no desenvolvimento do algoritmo eGSA (Capítulo 5), enquanto que o segundo trabalho refere-se a um resumo com a investigação do uso de vetores de sufixo generalizados no

problema de montagem *de-novo* (Apêndice A). O terceiro trabalho é um resumo expandido escrito com a revisão bibliográfica realizada para o Exame de Qualificação. Por fim, o quarto trabalho refere-se a um artigo em revista realizado em parceria com outros pesquisadores do grupo de pesquisa, no qual empregou-se a árvore de sufixo para indexar séries temporais.

6.2 Trabalhos Futuros

Trabalhos futuros relacionados aos algoritmos desenvolvidos nesta dissertação incluem:

- Adaptação do algoritmo eGSA para trabalhar com múltiplos discos. Nesse desenvolvimento, pretende-se considerar um disco somente para operações de leitura e outro somente para operações de escrita.
- Desenvolvimento de um método para particionar automaticamente o conjunto de cadeias na primeira fase no caso de conjuntos de cadeias pequenas. Esse método deve considerar o tamanho da memória interna disponível e a quantidade de memória utilizada pelos algoritmos de construção de GSA_i e LCP_i.
- Validação do algoritmo eGSA em outros domínios de dados, como os disponíveis em `http://pizzachili.dcc.uchile.cl/` e `http://www.cas.mcmaster.ca/~bill/strings/`.

Referências Bibliográficas

Abouelhoda, M. I., Kurtz, S., e Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86.

Adjeroh, D., Bell, T., e Amar Mukherjee (2008). *The burrows-wheeler transform: data compression, suffix arrays, and pattern matching*. Springer Publishing Company, Incorporated, Boston, MA.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., e Lipman, D. J. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410.

Altschul, S. F., Madden, T. L., Schäffer, A. A., Zhang, J., Zhang, Z., Miller, W., e Lipman, D. J. (1997). Gapped Blast and PsiBlast: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389—-3402.

Aluru, S. (2005). *Handbook of Computational Molecular Biology (Chapman & All/Crc Computer and Information Science Series)*. Chapman & Hall/CRC.

Arge, L., Ferragina, P., Grossi, R., e Vitter, J. S. (1997). On sorting strings in external memory (extended abstract). In *Proc. ACM symposium on Theory of computing*, STOC '97, pags. 540–548, New York, NY, USA. ACM.

Arnold, M. e Ohlebusch, E. (2011). Linear Time Algorithms for Generalizations of the Longest Common Substring Problem. *Algorithmica*, 60(4):806–818.

Baets, B. D., Fack, V., e Dawyndt, P. (2012). Prospects and limitations of full-text index structures in genome analysis. *Nucleic Acids Research*, pags. 1–23.

Baker, M. (2012). De novo genome assembly: what every biologist should know. *Nature Methods*, 9(4):333–337.

Barh, D., Miyoshi, A., e Azevedo, V. (2013). Next-Generation Sequencing and Assembly of Bacterial Genomes. In *OMICS: Applications in Biomedical, Agricultural, and Environmental Sciences*, pags. 367–384.

Barsky, M., Stege, U., Thomo, A., e Upton, C. (2008). A new method for indexing genomes using on-disk suffix trees. In *Proc. CIKM*, pag. 649, New York, New York, USA. ACM Press.

Barsky, M., Stege, U., Thomo, A., e Upton, C. (2009). Suffix trees for very large genomic sequences. In *Proc. CIKM*, volume 26, pag. 1417, New York, New York, USA. ACM Press.

Bauer, M. J., Cox, A. J., Rosone, G., e Sciortino, M. (2012). Lightweight LCP Construction for Next-Generation Sequencing Datasets. In *Proc. WABI*, pags. 326–337.

Benson, D. A., Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J., e Sayers, E. W. (2013). GenBank. *Nucleic acids research*, 41(Database issue):D36–42.

Bingmann, T., Fischer, J., e Osipov, V. (2013). Inducing Suffix and LCP Arrays in External Memory. In *Proc. ALENEX*, pags. 88–103.

Bolsover, S. R. (2004). *Cell biology: a short course*. Wiley-Liss.

Burkhardt, S. e Kärkkäinen, J. (2003). Fast lightweight suffix array construction and checking. In *Proc. CPM*, pags. 55–69.

Burrows, M. e Wheeler, D. (1994). A block-sorting lossless data compression algorithm. *Systems Research*.

Calladine, C. R. (2004). *Understanding DNA: the molecule & how it works*. Elsevier Academic Press.

Chaisson, M. J. e Pevzner, P. A. (2008). Short read fragment assembly of bacterial genomes. *Genome research*, 18(2):324–30.

Cheung, D. W. (2003). Approximate string matching in DNA sequences. In *Proc. Conference on Database Systems for Advanced Applications*, pags. 303–310. IEEE Comput. Soc.

Compeau, P. E. C., Pevzner, P. A., e Tesler, G. (2011). How to apply de Bruijn graphs to genome assembly. *Nature biotechnology*, 29(11):987–91.

Crauser, A. e Ferragina, P. (1999). On Constructing Suffix Arrays in External Memory. In *Proc. European Symposium on Algorithms*, pags. 224–235.

Crauser, A. e Ferragina, P. (2002). A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory. *Algorithmica*, 32(1):1–35.

Crauser, A. e Mehlhorn, K. (1999). LEDA-SM: Extending LEDA to Secondary Memory. In *Proc. International Workshop on Algorithm Engineering*, pags. 228–242.

Crick, F. (1970). Central Dogma of Molecular Biology. *Nature*, 227(5258):561–563.

Crochemore, M., Grossi, R., Kärkkäinen, J., e M. Landau, G. (2013). A Constant-Space Comparison-Based Algorithm for Computing the BurrowsâWheeler Transform. In *Proc. CPM*, volume 118653, pags. 74–82.

Dementiev, R. (2005). The STXXL library. Technical report.

Dementiev, R., Kärkkäinen, J., Mehnert, J., e Sanders, P. (2005). Better External Memory Suffix Array Construction. In Demetrescu, C., Sedgewick, R., e Tamassia, R., editors, *Proc. Workshop on Algorithm Engineering and Experiments*, pags. 86–97. SIAM.

Dementiev, R., Kärkkäinen, J., Mehnert, J., e Sanders, P. (2008). Better external memory suffix array construction. *ACM Journal of Experimental Algorithmics*, 12:3.4:1—-3.4:24.

Dhaliwal, J., Puglisi, S. J., e Turpin., A. (2012). Trends in Suffix Sorting : A Survey of Low Memory Algorithms. In *Proc. Australasian Computer Science Conference*, number Acsc, pags. 81–98, Melbourne, Australia.

Ferragina, P. e Manzini, G. (2000). Opportunistic Data Structures with Applications. In *Proc. FOCS*, pags. 390—-398.

Fischer, J. (2011). Inducing the LCP-Array. *Proc. Algorithms and Data Structures Symp.*, pags. 374–385.

Garcia-Molina, H., Widom, J., e Ullman, J. D. (1999). *Database System Implementation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Gog, S. e Ohlebusch, E. (2011). Fast and Lightweight LCP-Array Construction Algorithms. In Müller-Hannemann, M. e Werneck, R. F. F., editors, *Proc. ALENEX*, pags. 25–34, Francisco, California, USA. SIAM.

Gonnella, G. e Kurtz, S. (2012a). Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC bioinformatics*, 13(1):82.

Gonnella, G. e Kurtz, S. (2012b). Readjoiner: a fast and memory efficient string graph-based sequence assembler. In *Proc. of German Conference on Bioinformatics - Highlight Abstracts*, pags. 35–38.

Gonnet, G. H., Baeza-yates, R., e Snider, T. (1992). *New indices for text: PAT Trees and PAT arrays*, pags. 66–82. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Grossi, R. (2011). A quick tour on suffix arrays and compressed suffix arrays. *Theoretical Computer Science*, 412(27):2964–2973.

Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA.

Gusfield, D., M. Landau, G., e Schieber, B. (1992). An efficient algorithm for the All Pairs Suffix-Prefix Problem. *Information Processing Letters*, 41(4):181–185.

Hernandez, D., François, P., Farinelli, L., Osterå s, M., e Schrenzel, J. (2008). De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome research*, 18(5):802–9.

Hui, Z., Han, W., Gao, Y., e Jingmin, Z. (2009). An online clustering algorithm for Chinese web snippets based on Generalized Suffix Array. In *Proc. International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pags. 148–154.

Itoh, H. e Tanaka, H. (1999). An Efficient Method for in Memory Construction of Suffix Arrays. In *Proc. SPIRE*, SPIRE '99, pags. 81–88, Washington, DC, USA. IEEE Computer Society.

Kärkkäinen, J., Sanders, P., e Burkhardt, S. (2003). Simple Linear Work Suffix Array Construction. In *Proc. Conference on Automata, Languages and Programming*, volume 14186, pags. 943–955. Springer.

Kärkkäinen, J., Sanders, P., e Burkhardt, S. (2006). Linear work suffix array construction. *ACM Journal*, 53(6):918–936.

Karp, R. M., Miller, R. E., e Rosenberg, A. L. (1972). Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. ACM symposium on Theory of computing*, pags. 125–136, New York, NY, USA. ACM.

Kasai, T., Lee, G., Arimura, H., Arikawa, S., e Park, K. (2001). Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proc. CPM*, pags. 181–192.

Kim, D., Sim, J., Park, H., e Park, K. (2003a). Linear-time construction of suffix arrays. In *Proc. CPM*, pags. 186–199.

Kim, D., Sim, J., Park, H., e Park, K. (2003b). Linear-time Search in Suffix Arrays. In *Proc. Australasian Workshop on Combinatorial Algorithms*, pags. 186–199. Springer.

Knuth, D. E., Morris, Jr., J. H., e Pratt, V. R. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350.

Ko, P. e Aluru, S. (2003). Space efficient linear time construction of suffix arrays. In *Proc. CPM*, pags. 200–210.

Korf, I., Yandell, M., e Bedell, J. (2003). *BLAST*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Larsson, N. e Sadakane, K. (2007). Faster suffix sorting. *Theoretical Computer Science*, 387:258–272.

Lesk, A. M. (2002). *Introduction to bioinformatics*. Oxford University Press.

Louza, F. A., Telles, G. P., e Ciferri, C. D. A. (2013). External Memory Generalized Suffix and LCP Arrays Construction. In Fischer, J. e Sanders, P., editors, *Proc. CPM*, pags. 201–210, Bad Herrenalb. Springer.

Manber, U. e Myers, E. W. (1990). Suffix Arrays: A New Method for On-Line String Searches. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pags. 319–327.

Manber, U. e Myers, E. W. (1993). Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948.

Maniscalco, M. A. e Puglisi, S. J. (2006). Faster lightweight suffix array construction. In *Proc. Australasian Workshop on Combinatorial Algorithms*, pags. 122–133.

Manzini, G. e Ferragina, P. (2004). Engineering a Lightweight Suffix Array Construction Algorithm. *Algorithmica*, 40(1):33–50.

Moffat, A., Puglisi, S. J., e Sinha, R. (2009). Reducing Space Requirements for Disk Resident Suffix Arrays. In *Proc. International Conference Database Systems for Advanced Applications*, pags. 730–744.

Mount, D. W. (2004). *Bioinformatics: Sequence and Genome Analysis, Second Edition*. Cold Spring Harbor Laboratory Press, 2nd edition.

Myers, E. W. (2005). The fragment assembly string graph. *Bioinformatics (Oxford, England)*, 21 Suppl 2:ii79–85.

Myers, E. W., Sutton, G. G., Delcher, A. L., Dew, I. M., Fasulo, D. P., Flanigan, M. J., Kravitz, S. A., Mobarry, C. M., Reinert, K. H., Remington, K. A., Anson, E. L., Bolanos, R. A., Chou, H. H., Jordan, C. M., Halpern, A. L., Lonardi, S., Beasley, E. M., Brandon, R. C., Chen, L., Dunn, P. J., Lai, Z., Liang, Y., Nusskern, D. R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G. M., Adams, M. D., e Venter, J. C. (2000). A whole-genome assembly of drosophila. *Science*, 287(5461):2196–204.

Ng, W. e Kakehi, K. (2008). Merging string sequences by longest common prefixes. *Information Processing Society of Japan Digital Courier*, 4:69–78.

Nong, G. (2013). Practical linear-time O (1)-workspace suffix sorting for constant alphabets. *ACM Transactions on Information Systems*, 31(3):1–15.

Nong, G., Zhang, S., e Chan, W. H. (2009). Linear Time Suffix Array Construction Using D-Critical Substrings. In *Proc. CPM*, number 60873056, pags. 54–67.

Nong, G., Zhang, S., e Chan, W. H. (2011). Two Efficient Algorithms for Linear Time Suffix Array Construction. *IEEE Transactions on Computers*, 60(10):1471–1484.

Ohlebusch, E. e Gog, S. (2010). Efficient algorithms for the all-pairs suffix-prefix problem and the all-pairs substring-prefix problem. *Information Processing Letters*, 110(3):123–128.

Ouellette, A. D. e Baxevanis, F. B. F. (2005). *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. Wiley-interscience, Hoboken, New Jersey, U.S.A.

Paszkiewicz, K. e Studholme, D. J. (2010). De novo assembly of short sequence reads. *Briefings in Bioinformatics*, 11(5):457–472.

Pearson, W. R. e Lipman, D. J. (1988). Improved tools for biological sequence comparison. *Proc. National Academy of Sciences*, 85(8):2444–2448.

Pevzner, P. a., Tang, H., e Waterman, M. S. (2001). An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences of the United States of America*, 98(17):9748–53.

Pinho, A., Ferreira, P., Garcia, S., e Rodrigues, J. (2009). On finding minimal absent words. *BMC bioinformatics*, 10:137.

Puglisi, S. J., Smyth, W. F., e Turpin, A. H. (2007). A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):1–31.

Sadakane, K. (1998). A Fast Algorithms for Making Suffix Arrays and for Burrows-Wheeler Transformation. In *Proc. DCC*, pags. 129–138.

Schbath, S., Martin, V., Zytnicki, M., Fayolle, J., Loux, V., e Gibrat, J.-F. (2012). Mapping reads on a genomic sequence: an algorithmic overview and a practical comparative analysis. *Journal of computational biology : a journal of computational molecular cell biology*, 19(6):796–813.

Scheibye-Alsing, K., Hoffmann, S., Frankel, A., Jensen, P., Stadler, P. F., Mang, Y., Tommerup, N., Gilchrist, M. J., Nygå rd, A.-B., Cirera, S., Jø rgensen, C. B., Fredholm, M., e Gorodkin, J. (2009). Sequence assembly. *Computational Biology and Chemistry*, 33(2):121–36.

Selzer, P. M., Marhfer, R. J., e Rohwer, A. (2008). *Applied Bioinformatics: An Introduction*. Springer Publishing Company, Incorporated.

Setubal, J. a. C. e Meidanis, J. a. (1997). *Introduction to computational molecular biology*. PWS Publishing Company.

Seward, J. (2000). On the Performance of BWT Sorting Algorithms. In *Proc. DCC*, pags. 173–182, Washington, DC, USA. IEEE Computer Society.

Shi, F. (1996). Suffix arrays for multiple strings: A method for on-line multiple string searches. In Jaffar, J. e Yap, R., editors, *Proc. ASIAN*, volume 1179 of *Lecture Notes in Computer Science*, pags. 11–22. Springer Berlin / Heidelberg.

Simpson, J. T. e Durbin, R. (2010). Efficient construction of an assembly string graph using the FM-index. *Bioinformatics (Oxford, England)*, 26(12):i367–i373.

Simpson, J. T. e Durbin, R. (2012). Efficient de novo assembly of large genomes using compressed data structures. *Genome research*, 22(3):549–56.

Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Jones, S. J. M., e Birol, I. (2009). ABySS: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–23.

Sinha, R., Puglisi, S. J., Moffat, A., e Turpin, A. (2008). Improving suffix array locality for fast pattern matching on disk. In *Proc. SIGMOD*, pags. 661–672.

Stepanov, A. e Lee, M. (1994). The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project.

Välimäki, N. e Mäkinen, V. (2007). Space-efficient algorithms for document retrieval. In *Proc. CPM*, number i, pags. 205–215. Springer.

Weiner, P. (1973). Linear pattern matching algorithms. In *Proc. Annual Symposium on Switching and Automata Theory*, pags. 1–11, Washington, DC, USA. IEEE Computer Society.

Xiong, J. (2006). *Essential bioinformatics*. Cambridge University Press.

Zerbino, D. R. e Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–9.

Apêndice

A

Vetores de Sufixo e Montagem de Genomas

Nesse apêndice é investigado o uso de vetores de sufixo no problema de montagem de genomas. Em particular, é apresentado um algoritmo preliminar, chamado de SG2L, para a construção de grafos de cadeias utilizando vetores de sufixo. Esse trabalho foi desenvolvido durante um estágio de pesquisa no exterior realizado com os pesquisadores Steve Hoffmann [1] e Peter F. Stadler [2] no laboratório *LIFE - Leipzig Research Center for Civilization Diseases* da Universidade de Leipzig - Alemanha, com bolsa BEPE/FAPESP (processo número 2013/01752-6).

Esse apêndice está estruturado da seguinte forma. Na Seção A.1 são resumidos conceitos de montagem de genomas *de-novo* e na Seção A.2 são descritos os grafos de sobreposições e de cadeias. Em seguida, na Seção A.3 é apresentada a proposta inicial do algoritmo SG2L para a construção de grafos de cadeias utilizando apenas os vetores de sufixo generalizados aumentados com o vetor *LCP*. Por fim, são feitas considerações finais e são discutidos trabalhos futuros.

[1] `http://hoffmann.bioinf.uni-leipzig.de/LIFE/Steve.html`
[2] `http://www.bioinf.uni-leipzig.de/~studla/`

A.1 Montagem *de-novo*

A montagem de genomas *de-novo* (descrita na Seção 2.1.3), ilustrada na Figura A.1, realiza a reconstrução da sequência original de DNA utilizando apenas informações contidas nas *reads* obtidas durante o sequenciamento do DNA [Compeau et al., 2011]. A montagem é feita por meio da identificação de sobreposições entre as *reads* e geração de cadeias maiores em *contigs* e posterior ligação dos *contigs* em *scaffolds* [Baker, 2012].

A geração dos *contigs* pode ser realizada por meio de caminhos em um grafo (de montagem) que represente todas as sobreposições entre as *reads* sequenciadas. No final, os *contigs* gerados são mapeados em *super-contigs*, chamados de *scaffolds* (porções da cadeia contendo *contigs* e espaços não montados). Esse procedimento é conhecido como *scaffolding*, e fornece a orientação e ordem corretas dos *contigs* gerados. Em geral, são utilizadas informações adicionais sobre o genoma, como *reads* pareadas e/ou a ordem de genes em espécies relacionadas [Paszkiewicz e Studholme, 2010].

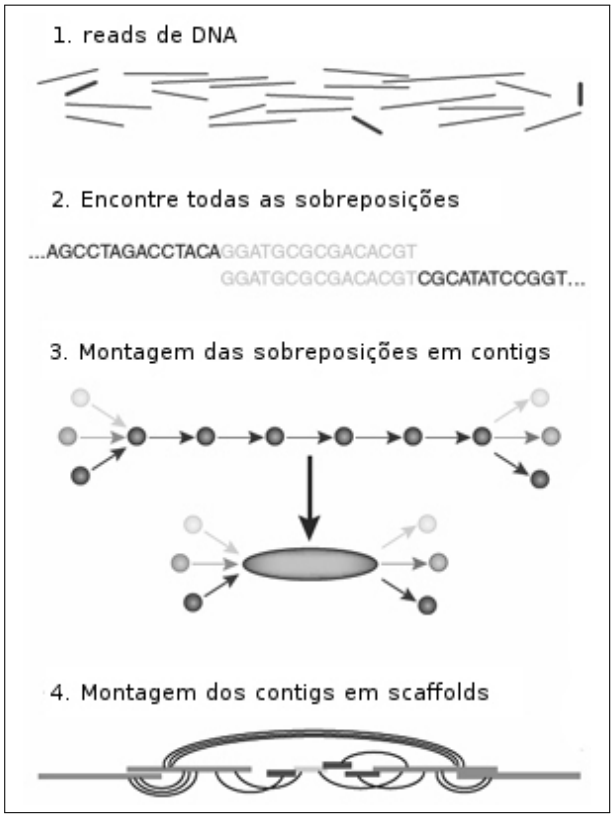

Figura A.1: Montagem *de-novo*, adaptado de [Baker, 2012].

O grafo *de Bruijn*, reformulado por Pevzner et al. [2001], tem sido muito utilizado por ferramentas de montagem *de-novo* para a identificação de sobreposições entre as *reads* e geração dos *contigs* (*e.g.* Velvet [Zerbino e Birney, 2008], EULER-SR [Chaisson e Pevzner, 2008] e ABySS [Simpson et al., 2009]). Entretanto, o volume da dados gerados pelos sequenciadores de nova geração e o tamanho das *reads* (menores do que anteriormente) têm dificultado o seu uso [Gonnella e Kurtz, 2012b].

Uma alternativa que tem se destacado é o grafo de cadeias, proposto por Myers [2005] e recentemente utilizado em eficientes ferramentas de montagem *de-novo* (*e.g.* Edena [Hernandez et al., 2008], SGA [Simpson e Durbin, 2012] e Readjoiner [Gonnella e Kurtz, 2012a]). Comparado como o grafo *de Bruijn*, o grafo de cadeias possui a vantagem de não particionar as *reads* em cadeias de tamanho q (q-mers), o que facilita a detecção de pequenas repetições [Simpson e Durbin, 2010], e cada caminho no grafo representa uma montagem válida das *reads* [Gonnella e Kurtz, 2012a].

A.2 Grafos de Montagem

Seja $\mathcal{R} = \{r_1, r_2, \ldots, r_k\}$ o conjunto de k cadeias em $\Sigma^\$$, com tamanhos $n_1, \ldots, n_k$ e tamanho total $N = \sum_{x=1}^{k} n_x$, sendo que cada cadeia r_x representa uma *read* gerada durante o sequenciamento de um genoma. É assumido que o conjunto $\mathcal{R}$ não contém *reads* duplicadas ou contidas em outras *reads*. Sejam as cadeias $r_x, r_y \in \mathcal{R}$ e $l_{min} > 0$ um parâmetro de tamanho mínimo.

Definição A.1 *A cadeia $r_x = x_1 \cdot x_2 \cdot \$$ sobrepõe $r_y = y_1 \cdot y_2 \cdot \$$ caso $x_2 = y_1$ e $|y1| > l_{min}$.*

Definição A.2 *A tupla $\langle r_x, r_y, l_{x,y} \rangle$ denota a maior sobreposição de r_x em r_y, de tamanho $l = |y1| > l_{min}$, em $\mathcal{R}$.*

As Definições A.1 e A.2 são ilustradas na Figura A.2.

Em outras palavras, $\langle r_x, r_y, l_{x,y} \rangle$ é o maior casamento do tipo sufixo-prefixo de r_x em r_y, isto é, o sufixo $r_x[n_x - l_{x,y} - 1, n_x - 1]$ é igual ao prefixo $r_y[1, l_{x,y}]$.

Definição A.3 *$SPM(\mathcal{R})$ é o conjunto de todas as sobreposições de tamanho máximo $\langle r_x, r_y, l_{x,y} \rangle$ entre todos os pares r_x, r_y de $\mathcal{R}$, com $l_{x,y} > l_{min}$.*

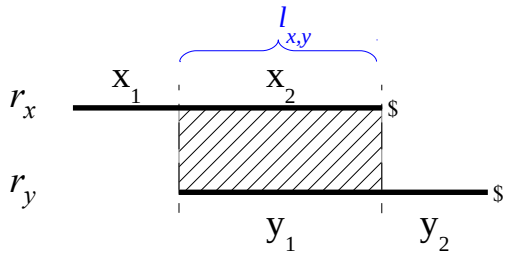

Figura A.2: Sobreposição $\langle r_x, r_y, l_{x,y} \rangle$

Grafo de sobreposições

Um grafo de sobreposições (*overlap graph*) para o conjunto $\mathcal{R}$, denotado por $OG(\mathcal{R}) = \{V_o, E_o\}$, é um grafo onde cada *read* $r_x \in \mathcal{R}$ é representada por um nó $X \in V_o$, e para cada sobreposição $\langle r_x, r_y, l_{x,y} \rangle \in SPM(\mathcal{R})$, o nó X é ligado ao nó Y por uma aresta $(X, Y) \in E_o$. O caminho que conecta o nó X ao Y é denotado por $X \rightarrow Y$. A cadeia resultante de um caminho $X \rightarrow Y \rightarrow Z$ corresponde à concatenação das cadeias sobrepostas nos nós X, Y e Z, isto é, $r_x[1, n_1 - 1] \cdot r_y[l_{x,y}, n_y - 1] \cdot r_z[l_{y,z}, n_z]$.

Na Figura A.3 é ilustrado um grafo de sobreposições para as *reads* $r_x = x_1 \cdot x_2 \cdot x_3 \cdot \$$, $r_y = y_1 \cdot y_2 \cdot y_3 \cdot \$$ e $r_z = z_1 \cdot z_2 \cdot z_3 \cdot \$$. As arestas (X, Y), (X, Z) e (Y, Z) correspondem às tuplas $\langle r_x, r_y, l_{x,y} \rangle$, $\langle r_x, r_z, l_{x,z} \rangle$ e $\langle r_y, r_z, l_{y,z} \rangle$, respectivamente. A cadeia gerada pelo caminho $X \rightarrow Y$ é igual à $x_1 \cdot x_2 \cdot x_3 \cdot y3 \cdot \$$. O mesmo se aplica para os outros caminhos.

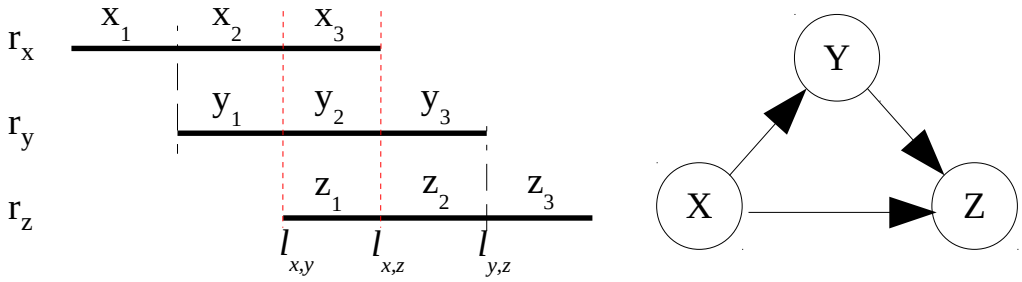

Figura A.3: Grafo de sobreposições

A construção do grafo de sobreposições consiste em encontrar o conjunto de todas as sobreposições $\langle r_x, r_y, l_{x,y} \rangle \in SPM(\mathcal{R})$. Esse problema, conhecido como o casamento sufixo-prefixo de todos os pares [Gusfield, 1997], pode ser resolvido em tempo $O(N + k^2)$ utilizando árvores de sufixo generalizadas Gusfield et al. [1992] ou vetores de sufixo generalizados aumentados com *LCP* Ohlebusch e Gog [2010]. Entretanto, para grandes

conjuntos de *reads*, com valor de k muito grande, essa solução pode gastar muito tempo e consumir muita memória.

Grafo de cadeias

Myers [2005] propôs o grafo de cadeias (*string graph*), uma versão melhorada do grafo de sobreposições. O grafo de cadeias para o conjunto $\mathcal{R}$, denotado por $SG(\mathcal{R}) = \{V_s, E_s\}$, é um subgrafo de $OG(\mathcal{R})$, com $V = V_o$ e $E \subseteq E_o$. Em $SG(\mathcal{R})$, cada aresta (X, Y) no grafo é rotulada com um par (a, b), que representa o prefixo de r_x e o sufixo de r_y, respectivamente, que não foram casados em $\langle r_x, r_y, l_{x,y}\rangle$. Na Figura A.2 esse rótulos seriam (x_1, y_2). A cadeia resultante de um caminho $X \to Y \to Z$ corresponde à concatenação da cadeia r_x com os componentes b das arestas Y e Z. Por fim, o grafo $SG(\mathcal{R})$ contém apenas arestas não-transitivas de $OG(\mathcal{R})$. Uma aresta $(X, Y) \in E_o$ é transitiva se a cadeia resultante no caminho $X \to Y$ estiver contida em outro caminho do grafo, esse caminho é chamado de redundante, e $(X, Y) \notin E_s$.

Na Figura A.4 é ilustrado o grafo de cadeias $SG(\mathcal{R})$ para as cadeias $r_x = x_1 \cdot x_2 \cdot x_3 \cdot \$$, $r_y = y_1 \cdot y_2 \cdot y_3 \cdot \$$ e $r_z = z_1 \cdot z_2 \cdot z_3 \cdot \$$. As arestas (X, Y) e (Y, Z) correspondem às tuplas $\langle r_x, r_y, l_{x,y}\rangle$ e $\langle r_y, r_z, l_{y,z}\rangle$, respectivamente. A aresta (X, Y) é rotulada com (x_1, y_3), já que $x_2 \cdot x_3 = y_1 \cdot y_2$, sendo que o primeiro componente é ilustrado abaixo da aresta e o segundo acima. O caminho $X \to Y$ resulta na cadeia $r_x \cdot y_3 = x_1 \cdot r_y$. O mesmo se aplica para a aresta (Y, Z). Note que, a aresta $(X, Z) \in E_o$ (Figura A.3), referente à tupla $\langle r_x, r_z, l\rangle$ não está presente em $SG(\mathcal{R})$. Essa é uma aresta transitiva pois a cadeia resultante do caminho $X \to Z = r_x \cdot z_2 \cdot z_3$ está contida em $X \to Y \to Y = r_x \cdot y_3 \cdot z_3$. No caso, $y_3 = z_2$.

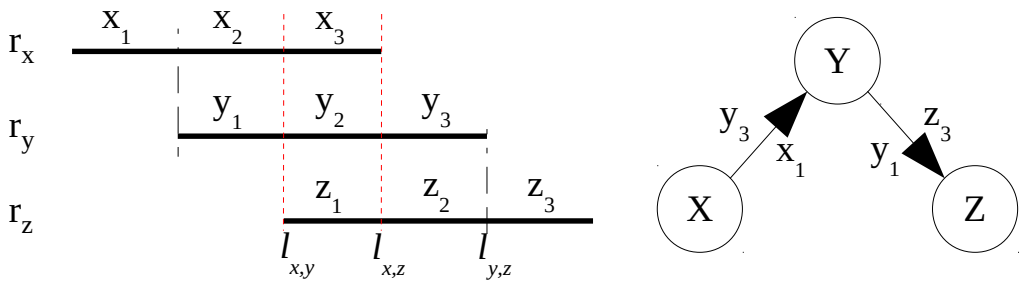

Figura A.4: Grafo de cadeias

As arestas transitivas podem ser removidas sem perda de informação no grafo de cadeias, isto é, no final cada caminho no grafo de cadeias representa uma montagem válida entre as *reads* [Gonnella e Kurtz, 2012a].

O grafo de cadeias pode ser construído indiretamente, primeiro construindo o grafo de sobreposições e depois removendo todas as arestas transitivas. Myers [2005] propôs um algoritmo para a remoção das arestas transitivas do grafo de sobreposições em tempo $O(|E_o|)$, onde $|E_o|$ é o número esperado de arestas no grafo. Entretanto, a construção do grafo de sobreposições torna essa abordagem inviável para grandes volumes de dados.

Recentemente, Simpson e Durbin [2012] e Gonnella e Kurtz [2012a] propuseram algoritmos para construção direta de grafos de cadeias (sem a construção do grafo de sobreposições). Simpson e Durbin [2012] propuseram o montador SGA que utiliza o método FM-index [Ferragina e Manzini, 2000] para a construção de $SG(\mathcal{R})$, e Gonnella e Kurtz [2012a] propuseram o montador Readjoiner que utiliza o vetor de sufixo aumentado com o vetor *LCP* para a construção de $SG(\mathcal{R})$.

Nos testes de desempenho comparativos realizados em [Gonnella e Kurtz, 2012a], o algoritmo de construção do Readjoiner foi 20 vezes mais rápido e apesar do SGA utilizar uma estrutura de dados compacta o algoritmo no Readjoiner utilizou aproximadamente 30% menos memória primária.

O algoritmo de construção utilizado no Readjoiner percorre o vetor de sufixo para encontrar o conjunto $SPM(\mathcal{R})$ e identifica arestas transitivas comparando os caracteres não casados (prefixos) de duas cadeias que sobrepõem uma terceira. No exemplo ilustrado na Figura A.4, durante o processamento da cadeia r_z é encontrado $\langle r_x, r_z, l_{x,z}\rangle$ e $\langle r_y, r_z, l_{y,z}\rangle$. O algoritmo proposto por [Gonnella e Kurtz, 2012a] compara os caracteres em x_2 com os em y_1, caso sejam iguais $X \rightarrow Z$ é transitiva. Essa comparação é feita utilizando uma árvore *trie*, que é construída durante o processamento de r_z para armazenar todos os prefixos das cadeias r_y que sobrepõem r_z.

Entretanto, o custo para construir e manter em memória a árvore *trie* para cada conjunto de cadeias que sobrepõe uma cadeia r_z pode ser evitado. Além disso, a comparação de caracteres faz com que além das estruturas de dados, todas as cadeias de $\mathcal{R}$ devam estar carregadas em memória interna. Essas melhorias são descritas na Seção A.3, na qual é proposto o algoritmo SG2L.

A.3 Algoritmo SG2L

Nessa seção é proposto o algoritmo SG2L (*String Graph construction using Generalized LCP-array*) para a construção direta do grafo de cadeias utilizando vetores de sufixo generalizados aumentados. O algoritmo SG2L utiliza a mesma estratégia para encontrar o

conjunto $SPM(\mathcal{R})$ proposta no Readjoiner [Gonnella e Kurtz, 2012a]. Entretanto, a identificação das arestas não-transitivas é melhorada utilizando apenas informações presentes no vetor *LCP* sem a comparação de caracteres.

Considere o vetor de sufixo generalizado aumentado com o vetor *LCP* (*GESA*) construído para o conjunto de cadeias $\mathcal{R}$ (Seção 3.2).

Definição A.4 *Seja $\Delta(r_y)$ o conjunto de todas as cadeias r_x que sobrepõe r_y em $l_{x,y} > l_{min}$ caracteres, ou seja, $r_x \in \Delta(r_y) \iff \langle r_x, r_y, l_{x,y} \rangle \in SPM(\mathcal{R})$.*

No grafo de sobreposições $OG(\mathcal{R})$, todos os nós conectados (incidentes) ao nó que representa a cadeia $r_y \in \mathcal{R}$ representam cadeias em $\Delta(r_y)$. O conjunto $\Delta(r_x)$ pode ser encontrado utilizando o *GESA* conforme formaliza o Lema A.1.

Lema A.1 *Todos os sufixos que sobrepõem r_y estão em posições anteriores ao sufixo $r_y[1, n_y]$ em GSA.*

Prova A.1 *Se o sufixo $r_x[i, n_x]$ sobrepõe $r_y[1, n_y]$ em $l_{x,y}$ caracteres ($i = n_x - l_{x,y} - 1$), então $r_x[i, n_x - 1] = r_y[1, l_{x,y}]$, e já que $r_x[n_x] = \$ < r_x[l_{x,y} + 1]$, então $r_x[i, n_x] < r_y[1, n_y]$, ou seja, $pos(GSA, r_x[i, n_x]) < pos(GSA, r_y[1, n_y])$.*

O Lema A.1 pode ser utilizado para encontrar um intervalo $[l+1, r]$ em *GESA* para cada cadeia r_y, no qual estão todos sufixos que sobrepõem r_y em pelo menos l_{min} caracteres, isto é, todas as tuplas $\langle r_x, r_y, l_{x,y} \rangle \in SPM(\mathcal{R})$, e consequentemente pode ser utilizado para encontrar o conjunto $\Delta(r_y)$. No Algoritmo 4 é detalhada essa ideia. Partindo da posição $r = pos(GESA, r_y[1, n_y])$ em *GESA* (Linha 2), o algoritmo, por meio de uma busca sequencial no sentido decrescente em *GESA* (Linhas 4 à 8), encontra a posição l ($l < r$), quando $LCP[l] < l_{min}$ (Linha 6). Por fim, para encontrar o conjunto $\Delta(r_y)$ é preciso verificar (Linha 6) se $LCP[l] = |r_x[i, n_x]| - 1$ ($l < x \leq r$) e se $GSA[l].text \neq y$, respectivamente.

O Algoritmo 4 pode ser utilizado para encontrar os conjuntos $\Delta(r_y)$, para todas as cadeias $r_y \in \mathcal{R}$, e consequentemente obter $SPM(\mathcal{R})$ e construir o grafo de sobreposições $SG(\mathcal{R})$. Essa estratégia é similar à proposta por Gonnella e Kurtz [2012a] no algoritmo para identificação do conjunto $SPM(\mathcal{R})$. A seguir, é descrito como identificar todas as sobreposições em $SPM(\mathcal{R})$ que geram arestas não-transitivas para a construção do grafo de cadeias $SG(\mathcal{R})$.

Algoritmo 4: Calcular $\Delta(r_y)$

Entrada: cadeia r_y e l_{min}
Saída: $[l+1, r]$ e $\Delta(r_y)$

```
1 início
2     r ⟵ pos(GESA, r_y[1, n_y]);
3     l ⟵ r − 1;
4     enquanto LCP[l] > l_min faça
5         // pos(GESA, r_x[i, n_x]) = l
6         se LCP[l] = |r_x[i, n_x]| − 1 e GSA[l].text ≠ y então
7             Δ(r_y) ⟵ r_x;
8         l ⟵ r − 1;
```

Identificando sobreposições não-transitivas

As sobreposições em $SPM(\mathcal{R})$ que geram arestas não-transitivas podem ser encontradas utilizando os conjuntos $\Delta(r_y)$, $r_y \in \mathcal{R}$. Essa relação é formalizada no Lema A.2.

Lema A.2 *Se $r_x \in \Delta(r_z) \cap \Delta(r_y)$ e $r_y \in \Delta(r_z)$, $\forall r_y \in \Delta(r_z) \setminus r_x$, $X \rightarrow Z$ é transitiva, caso contrário, $X \rightarrow Z$ é não-transitiva.*

Prova A.2 *Se r_x sobrepõe as cadeias r_z e r_y ($r_x \in \Delta(r_z) \cap \Delta(r_y)$), e r_y sobrepõe a cadeia r_z ($r_y \in \Delta(r_z)$) então $X \rightarrow Y \rightarrow Z = X \rightarrow Z$, portanto $X \rightarrow Z$ é transitiva.*

Utilizando o Lema A.2 para identificar arestas não-transitivas. Primeiro calcula-se $\Delta(r_z)$, e para cada $r_y \in \Delta(r_z)$, calcula-se $\Delta(r_y)$. Em seguida, é obtido o conjunto $\Delta(r_y) \cap \Delta(r_z)$ e, para cada $r_x \in \Delta(r_z) \setminus \bigcup_{r_y \in \Delta(r_z)} (\Delta(r_y) \cap \Delta(r_z))$, tem-se que $X \rightarrow Z$ é não-transitiva. Em outras palavras, para cada $r_x \in \Delta(r_z)$, se $r_x \notin \Delta(r_y) \cap \Delta(r_z)$, $\forall r_y \in \Delta(r_z) \setminus r_x$ então $X \rightarrow Z$ é não-transitiva.

O conjunto $\Delta(r_y)$ pode ser obtido utilizando o Algoritmo 4. Entretanto, essa abordagem não é eficiente, já que é preciso encontrar e manter em memória todos os conjuntos $\Delta(r_y)$ e calcular todas as interseções $\Delta(r_y) \cap \Delta(r_z)$. A seguir, é descrito como resolver esse problema utilizando apenas o vetor *GESA*.

Construção direta do grafo de cadeias

As sobreposições em $SPM(\mathcal{R})$ que geram arestas não-transitivas podem ser encontradas diretamente utilizando apenas as informações contidas no vetor *GESA*. Essa relação é formalizada no Lema A.3.

Lema A.3 *Se* $\langle r_x, r_y, l_{x,y}\rangle$ *e* $\langle r_y, r_z, l_{y,z}\rangle \in SPM(\mathcal{R})$*, e* $l_{x,y} > (n_y - l_{y,z} - 1) + l_{min}$ *então* $\langle r_x, r_z, l_{x,z}\rangle \in SPM(\mathcal{R})$.

Prova A.3 *Se* $\langle r_x, r_y, l_{x,y}\rangle \in SPM(\mathcal{R})$*, o sufixo* $r_x[i, n_x]$ *sobrepõe a cadeia* r_y *em* $l_{x,y} > l_{min}$ *caracteres* $(i = n_x - l_{x,y} - 1)$*, tem-se que* $r_x[i, n_x - 1] = r_y[1, l_{x,y}]$*, e se* $\langle r_y, r_z, l_{y,z}\rangle \in SPM(\mathcal{R})$*, o sufixo* $r_y[j, n_y]$ *sobrepõe a cadeia* r_z *em* $l_{y,z} > l_{min}$ *caracteres* $(j = n_y - l_{y,z} - 1)$*, tem-se que* $r_y[j, n_y - 1] = r_z[1, l_{y,z}]$*, então se* $l_{x,y} > j + l_{min}$*, tem-se que* $r_x[j + i, n_x - 1] = r_y[j, n_x - (j + i)] = r_z[1, n_x - (j + i)]$*, portanto* r_x *sobrepõe* r_z *em* $n_x - (j + i) = l_{x,z}$ *caracteres e a sobreposição* $\langle r_x, r_z, l_{x,z}\rangle \in SPM(\mathcal{R})$.

A prova do Lema A.3 é ilustrada na Figura A.5.

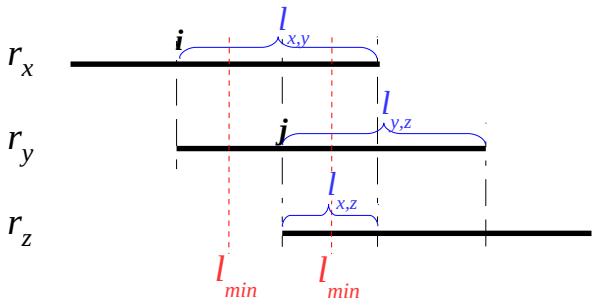

Figura A.5: Ilustração do Lema A.3

Um resultado direto do Lema A.3 é que se $\langle r_x, r_z, l_{x,z}\rangle \in SPM(\mathcal{R})$ então a aresta $X \rightarrow Z$ é transitiva. Dessa forma, para cada cadeia $r_z \in \mathcal{R}$ é possível identificar as cadeias $r_x, r_y \in \Delta(r_z)$ que geram arestas transitivas comparando os valores de $l_{x,y}$ e $l_{y,z}$. Esses valores correspondem ao $lcp(r_y[1, n_y], r_x[i, n_x])$ e $lcp(r_z[1, n_z], r_y[j, n_y])$, respectivamente, os quais podem ser obtidos eficientemente utilizando o vetor *GESA* por meio da função *rmq* (Definição 3.13). A função *rmq* pode ser calculada durante a leitura sequencial em *GESA* para a obtenção de cada $\Delta(r_y)$. No Algoritmo 5 é detalhada essa ideia.

Para cada cadeia $r_z \in R$, primeiro é criado o nó Z no grafo (Linhas 3 e 4). Então, o conjunto $\Delta(r_z)$ é calculado utilizando o Algoritmo 4 (Linha 5). Em seguida, para cada $r_y \in \Delta(r_z)$ é feita uma busca sequencial em *GESA* de $r = pos(GESA, r_y[1, n_y])$ até l

enquanto $LCP[l] < i + l_{min}$ (Linhas 8 à 11). Todas as cadeias r_x que sobrepõe r_y e r_z estão no intervalo $[l+1, r]$, então se $LCP[l] = |r_x[i, n_x]| - 1$ e $GSA[l].text \neq y$, r_x sobrepõe r_y (Linha 9) e a aresta $X \rightarrow Z$ é transitiva e é marcado em $\Delta(r_z)$ que a aresta $X \rightarrow Z$ não deve ser inserida (Linha 10). No final do processamento de cada cadeia $r_y \in \Delta(r_z)$ (Linha 12), caso essa cadeia não tenha sido marcada durante o processamento de outra cadeia $r_w \in \Delta(r_w)$, a aresta $Y \rightarrow Z$ é não-transitiva e pode ser inserida em $SG(\mathcal{R})$ (Linhas 12 e 13). No final, todas as arestas são obtidas e o grafo de cadeias $SG(\mathcal{R})$ para $\mathcal{R}$ é construído.

Algoritmo 5: Algoritmo SG2L

Entrada: $R = \{r_1, \ldots, r_k\}$
Saída: Grafo de cadeias $SG(\mathcal{R})$

1 **início**
2 **para** *cada* $r_z \in \mathcal{R}$ **faça**
3 **se** $Z \notin E_s$ **então**
4 $V_s \longleftarrow Z$; `// Criar nó` Z
5 Calcular $\Delta(r_z)$; `// Algoritmo 4`
6 **para** *cada* $r_y \in \Delta(r_z)$ **faça**
7 $l \longleftarrow pos(GESA, r_y[1, n_y])$; `// ` r_y `sobrepõe` r_z `no sufixo` $r_y[j, n_y]$
8 **enquanto** $LCP[l] > j + l_{min}$ **faça**
9 **se** r_x *sobrepõe* r_y **então**
10 Marcar r_x em $\Delta(r_z)$ que a aresta $X \rightarrow Z$ é transitiva;
11 $l \longleftarrow l - 1$;
12 **se** r_y *não está marcada como transitiva em* $\Delta(r_z)$ **então**
13 $E_s \longleftarrow (Y, Z)$ com rótulo $(a, b) = (r_y[1, j], r_z[n_y - j, n_z])$

Note que, a construção do grafo $SG(\mathcal{R})$ é realizada no sentido contrário das arestas, isto é, para um dado nó Z, encontra-se as arestas (Y, Z) que alcançam (incidentes) Z em $SG(\mathcal{R})$. Além disso, as arestas transitivas (X, Z) são identificadas durante o cálculo do intervalo $[l, r]$ das cadeias $r_y \in \Delta(r_z) \setminus r_x$.

A principal vantagem do algoritmo SG2L é que nenhuma comparação de caracteres é feita nos Algoritmos 4 e 5, isto é, uma vez construído o vetor de sufixo generalizado aumentado como vetor *LCP* para o conjunto $\mathcal{R}$, todas as cadeias de $\mathcal{R}$ podem ser removidas da memória primária.

Considerações Finais

Nesse apêndice foi investigado o uso de vetores de sufixo no problema de montagem de genomas. Inicialmente, na Seção A.1 foi discutido a abordagem de montagem *de-novo*. Na Seção A.2 foram descritos os grafos de sobreposição e de cadeias. Por fim, na Seção A.3 foi apresentado a proposta preliminar do algoritmo SG2L para a construção direta do grafo de cadeias utilizando apenas vetores de sufixo generalizados aumentados com o vetor LCP.

Como trabalhos futuros tem-se a implementação do algoritmo SG2L e a comparação com trabalhos correlatos como os apresentados em [Simpson e Durbin, 2012; Gonnella e Kurtz, 2012a]. Além disso, pretende-se estender o algoritmo de construção de grafos de cadeia para tratar repetições. Essas repetições podem ser identificadas em intervalos no vetor LCP aonde haja grandes valores de *lcp*.