Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Uma implementação de Grafo de de Bruijn baseada numa árvore de sufixos comprimida

Vitor Travassos Castelo Branco

Trabalho de Graduação

Recife
06 de Julho de 2018

Universidade Federal de Pernambuco
Centro de Informática

Vitor Travassos Castelo Branco

Uma implementação de Grafo de de Bruijn baseada numa árvore de sufixos comprimida

Trabalho apresentado ao Programa de Graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Paulo Gustavo Soares da Fonseca*

Recife
06 de Julho de 2018

Resumo

Boa parte das ferramentas para o sequenciamento do DNA baseado nas plataformas de alto desempenho ditas de nova geração, especificamente as destinadas à montagem dos fragmentos sequenciados, utilizam estruturas de dados para grafos de subsequencias. Devido ao enorme volume de dados, um dos principais gargalos dessas representações é espaço em memória exigido. Várias representações compactas vêm sendo propostas nos últimos anos para dados de texto, entretanto, existe uma relação inversa entre o espaço de memória e a eficiência/flexibilidade das operações suportadas. Mais recentemente, têm-se explorado a conexão entre os grafos de subsequencias e os índices de texto. Neste trabalho, produzimos uma implementação dos chamados Grafos de de Bruijn baseada numa representação sucinta da árvore de sufixos. Essa implementação foi analisada e comparada a uma representação sucinta disponível na literatura para aferir a sua viabilidade em termos de espaço e tempo.

Palavras-chave: Grafos de de Bruijn, Árvore de sufixos, Estruturas de dados sucintas

Sumário

Lista de Figuras

CAPÍTULO 1

Introdução

1.1 Motivação: montagem de fragmentos de DNA

O DNA, molécula orgânica responsável pela codificação e transmissão das características genéticas, é constituído por duas cadeias complementares formadas a partir de quatro *bases nitrogenadas*, representadas por `a`, `c`, `g` e `t`. Cada `a` de uma cadeia (fita) é emparelhado a um `t` da outra, assim como cada `g` é complementado por um `c`, e vice versa. Essa estrutura molecular torna-o passível de representação por apenas uma sequência de letras no alfabeto dessas quatro letras. Desvendar o genoma de um organismo limita-se com identificar a sequência de caracteres correspondente ao seu DNA.

Atualmente, o processo de sequenciamento de DNA é efetuado principalmente utilizando-se as plataformas de sequenciamento de alto desempenho ditas de "nova geração" (*Next-Generation Sequencing—NGS*) [44]. Essas tecnologias produzem um enorme volume de fragmentos curtos (comprimento abaixo das centenas) que precisam ser *montados*, i.e., alinhados e combinados, para reconstruir sequências originais de bilhões de letras.

1.1.1 Montagem baseada em Grafos de de Bruijn

As ferramentas para montagem de fragmentos NGS são majoritariamente baseadas nos chamados *Grafos de de Bruijn* (GdB) [13]. No GdB de ordem k construído a partir do conjunto de fragmentos S, $G(S)$, os nós correspondem às subsequencias de comprimento k (k-mers) das cadeias em S, e dois k-mers (nós) são unidos por uma aresta desde que haja uma sobreposi-

ção de tamanho $k-1$, de forma que as arestas correspondem aos $k+1$-mers de S. Efetuar a montagem de fragmentos usando GDB envolve problemas como o de encontrar *Caminhos Eulerianos*, que admite solução em tempo polinomial, em contraste com abordagens anteriores que envolvem soluções heurísticas para o problema de *Circuitos Hamiltonianos*, que é um problema NP-completo.

Entretanto, um dos principais limitadores quanto ao emprego dessas técnicas é o espaço de memória exigido pelos GDB que, se representado explicitamente, pode requerer centenas de GigaBytes [14]. Diante disto, diversos esforços vêm sendo empreendidos para desenvolver estruturas de dados eficientes do ponto de vista de espaço, permitindo todavia operações sobre o GDB em tempo comparável a uma representação tradicional. Dentre as principais linhas de ação encontradas na literatura, têm ganhado importância o desenvolvimento de estruturas de dados ditas *sucintas*, ou seja, cujo espaço ocupado é muito próximo do mínimo teórico necessário.

1.2 Objetivos deste trabalho

O objetivo geral deste projeto é explorar a conexão teórica entre os GdB e as estruturas de índices de texto, e sua realização prática. Em particular, o recente trabalho de Cazaux et al [9] aponta para as conexões entre as árvores de sufixos e os GdB. Especificamente, neste trabalho iremos desenvolver uma implementação sucinta baseada em Árvores de Sufixos Comprimidas [48]. Esta implementação será avaliada do ponto de vista teórico e prático quanto ao (i) tempo e memória de construção da estrutura, (ii) espaço de memória final da estrutura, (iii) tempo das operações de navegação através da simulação de percursos no grafo. Repare que essa primeira implementação será feita desde o início, sem reutilização de códigos de terceiros (apenas aluno e orientador), e servirá como princípio para otimizações e refinamentos futuros.

1.3 Organização desta monografia

O restante desta monografia está organizada da seguinte forma.

- No Capítulo 2 nós introduzimos os conceitos teóricos necessários à compreensão do trabalho realizado. A abordagem é *bottom-up*, ou seja, iniciamos com os conceitos elementares de sequências de caracteres, depois tratamos dos índices, para finalmente chegarmos aos Grafos de de Bruijn. O capítulo termina com uma revisão da literatura sobre as representações desses grafos.

- No Capítulo 3 descrevemos em detalhes técnicos a implementação realizada, incluindo a descrição da estrutura e operações suportadas. São incluídos pseudocódigos e análises teóricas de complexidade assintótica em tempo/espaço dos aspectos mais relevantes.

- No Capítulo 4 exibimos os resultados experimentais de testes feitos com nossa implementação.

- No Capítulo 5 apresentamos uma breve discussão geral sobre o trabalho e apontamos desenvolvimentos futuros.

CAPÍTULO 2

Índices Comprimidos e Grafos de de Bruijn

Tratamos do problema de construir um tipo abstrato de dados para um determinado grafo formado a partir de subsequências de um texto, cuja implementação baseia-se em estruturas de índices completos de representação sucinta. Neste capítulo introduzimos os conceitos e terminologia necessários para a compreensão do problema, bem como apresentamos uma revisão da literatura no tema.

2.1 Cadeias de caracteres

Neste trabalho, serão definidas estruturas para manipular dados de texto sobre um *alfabeto* finito de caracteres $\mathscr{A} = \{a_0, \ldots, a_{m-1}\}$. Uma *cadeia de caracteres* (*string*) sobre $\mathscr{A}$, ou simplesmente *cadeia*, é uma sequência de caracteres $X = x_0 \cdots x_{n-1}$, onde cada $x_i \in \mathscr{A}$. O caractere da posição i, denominado i-ésimo caractere, será identificado por $X[i] = x_i$. O *comprimento* da cadeia $X = x_0 \cdots x_{n-1}$ é denotado por $|X| = n$. A cadeia de comprimento 0, chamada *cadeia vazia*, é denotada por ε. O conjunto de todas as cadeias de comprimento finito sobre $\mathscr{A}$ é denotado por $\mathscr{A}^*$.

A operação de *concatenação* é uma operação binária $\cdot : \mathscr{A}^* \times \mathscr{A}^* \to \mathscr{A}^*$, associativa e não-comutativa, que recebe um par de cadeias $X = x_0 \cdots x_{n-1}$ e $Y = y_0 \cdots y_{s-1}$, e retorna uma cadeia correspondente à justaposição das entradas $X \cdot Y = XY = x_0 \cdots x_{n-1} y_0 \cdots y_{s-1}$, e logo $|XY| = |X| + |Y|$. A cadeia vazia é o elemento neutro da concatenação, isto é, $\varepsilon X = X\varepsilon = X$.

Uma *subcadeia* de $X = x_0 \cdots x_{n-1}$ é uma subsequência contígua denotada equivalentemente por $X[i\colon j] = X_{i..j} = x_i \cdots x_{j-1}$, com $0 \leq i \leq j \leq n$. O *prefixo* de comprimento l de X é definido

como a subcadeia $X[:l] = X[0:l]$. O *sufixo* de comprimento l de X é definido como a subcadeia $X[n-l:] = X[n-l:n]$.

Ao longo do texto, também serão utilizados *vetores* (*arrays*) numéricos $A = (a_0, \cdots, a_{n-1})$. Será empregada uma notação similar àquela usada com as cadeias para denotar subvetores contíguos, nomeadamente $A[i:j] = (a_i, \cdots, a_{j-1})$, $A[:j] = A[0:j]$, e $A[i:] = A[i:n]$. Um caso particular ocorre com vetores de valores binários (*booleanos*). Esses vetores são denominados *bitarrays* e serão denotados na forma de sequência $B = b_0 \cdots b_{n-1}$.

As seguintes convenções notacionais serão utilizadas. Vetores e cadeias serão nomeados por letras maiúsculas (A, X, B, ...) e suas componentes individuais pelas letras minúsculas correspondentes (a_i, x_j, ...). Conjuntos serão denotados por letras maiúsculas caligráficas ($\mathscr{A}$, $\mathscr{V}$, ...), e seus elementos individuais por letras minúsculas correspondentes. Variáveis numéricas serão denotadas por letras latinas minúsculas (m, n, i, j,...). Será usado $\lg n$ para denotar o logaritmo base 2, isto é $\lg n = \log_2 n$.

2.2 Índices de texto completos

Dentre as operações que necessitamos efetuar em dados de texto, destaca-se a localização das ocorrências de uma sequência menor $P = p_0 \cdots p_{m-1}$ numa sequência maior $T = t_0 \cdots t_{n-1}$. Uma ocorrência de P em T é uma subcadeia $T[i:i+m] = P$, a qual identificamos com a sua posição inicial i. Dizemos, nesse caso, que P ocorre em T na posição i. Nesse contexto, a cadeia P é chamada *padrão*, enquanto a cadeia T é chamada de *texto*. Os algoritmos clássicos para esse problema de *casamento exato de padrões* (*exact string matching*) [15] requerem tempo linear no texto. Entretanto, se nos for permitido preprocessar o texto, podemos construir uma estrutura auxiliar, chamada *índice*, que representa uma cadeia T e possibilita operações sobre suas subcadeias sem que seja necessário acessar a cadeia original, em particular permitindo o casamento de padrões em tempo linear no tamanho do padrão. Dentre as principais estruturas

de índicem completos, isto é, aqueles que representam todas as subcadeias do texto, destacam-se as árvores e vetores de sufixos.

2.2.1 Árvores de Sufixos

A *árvore de sufixos* de uma cadeia $S = s_0 \cdots s_{n-1}$ de tamanho n, $\mathrm{ST}(S)$, codifica todos os sufixos de S na forma de uma árvore enraizada na qual as arestas são rotuladas por subcadeias de S e cada nó u da árvore representa uma subcadeia obtida pela concatenação das arestas no caminho da raiz até u. É praxe supor que S é terminado por um caractere especial `$`, diferente dos demais, chamado *sentinela*. Nesse caso, $\mathrm{ST}(S)$ possui exatamente n folhas, cada uma representando um sufixo distinto de S. A árvore é construída de modo que não exista nó interno com menos que dois filhos e que não existam arestas advindas do mesmo nó (arestas irmãs) cujos rótulos possuam o mesmo caractere inicial. Segue, portanto, que $\mathrm{ST}(S)$ possui, no máximo, $2n$ nós. Além disso, a árvore possui ponteiros especiais chamados *suffix links* que ligam nós que representam cadeias na forma $s_i s_{i+1} \cdots s_j$ aos nós que representam os respectivos $s_{i+1} \cdots s_j$. A Figura 2.1 ilustra a árvore de sufixos de $S =$ `banana$`.

Embora apenas $2n$ subcadeias de S sejam explicitamente representadas pelos nós da árvore de sufixos de S, pode-se introduzir o conceito de *nó implícito* da árvore de sufixos. Um nó implícito u representa uma cadeia que é prefixo da cadeia representada por um nó (explícito) v, mas que, ao mesmo tempo, a cadeia representada pelo pai de v é prefixo da cadeia representada por u. Dessa forma, após cada caractere, exceto o último, no rótulo da aresta de u a v, existe um nó implícito que representa a cadeia composta pela concatenação do rótulo de u e de um prefixo da aresta. Note que, incluindo o último caractere das arestas nessa definição, tem-se todo nó, implícito ou explícito, excetuando-se a raiz.

A definição da árvore de sufixos pode ser estendida de modo que contenha todos os sufixos de um conjunto de cadeias. Essa extensão é chamada e *árvore de sufixos generalizada*.

Ukkonen [53] propôs um algoritmo de construção de árvores de sufixos *online* em tempo

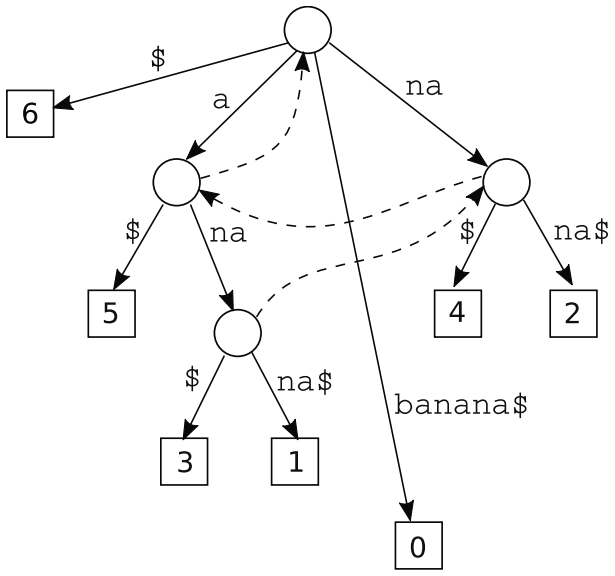

Figura 2.1 Árvore de sufixos de $S = \texttt{banana\$}$. As folhas estão rotuladas pelas posições iniciais dos sufixos que elas representam. Esses sufixos estão em ordem lexicográfica da esquerda para a direita. Os *suffix links* estão representados por linhas tracejadas.

linear para alfabetos de tamanho fixo e tempo $O(n \lg n)$ no caso geral. Nesse trabalho, no entanto, foi desenvolvido um algoritmo de construção que se aproveita da representação específica implementada.

2.2.2 Vetores de Sufixos

Os *vetores de sufixos*[1] foram introduzidos por Manber e Myers [32] na tentativa de compensar a principal limitação das árvores de sufixos, que é o espaço em memória exigido. O vetor de sufixos de $S = s_0 \cdots s_{n-1}$, $\mathrm{SA}(S)$, consiste simplesmente das posições iniciais dos sufixos não-vazios de S lexicograficamente ordenados. Por exemplo, se $S = \texttt{banana\$}$, temos $\mathrm{ST}(S) = (6,5,3,1,0,4,2)$ pois os sufixos de S em ordem lexicográfica são $S[6:] = \$ < S[5:] = \texttt{a\$} < S[3:] = \texttt{ana\$} < \cdots < S[2:] = \texttt{nana\$}$.

Um vetor de sufixos é uma permutação aleatória do intervalo $[0:m]$ e, portanto, requer $n \lg n$ bits de espaço, à medida em que permite a busca de padrões de tamanho m no texto em tempo

[1] Também chamados *arrays* de sufixos.

$O(m \lg n)$ no pior caso através de uma busca binária. Se acrescidos de estruturas auxiliares para armazenar a informação de maiores prefixos comuns (*lcp*, do Inglês *longest common prefixes*) entre sufixos ordenados consecutivos, podemos alcançar o mesmo custo assintótico linear no comprimento do padrão [32].

Diversos algoritmos para a construção do array de sufixos existem, um deles sendo o algoritmo SA-IS, proposto por Nong et al [39], que executa em tempo linear.

2.3 Índices comprimidos

No contexto de processamento de cadeias de tamanho comparável ao DNA humano, a eficiência em memória é uma preocupação presente, sobretudo face ao enorme aumento no volume de dados decorrente dos avanços nas teconologias de sequenciamento. Sendo assim, esforços têm sido empreendidos no sentido de se obter representações para os índices de texto cada vez mais econômicas em memória, sem todavia maiores sacrifícios quanto ao custo assintótico das operações de consulta.

2.3.1 Estruturas de dados comprimidas

Diversas estruturas serão apresentadas ao longo deste capítulo que ilustram a evolução das estruturas em relação à eficiência em memória das representações de leituras de DNA. O problema pode ser abordado utilizando ou não técnicas de compressão sem perda de dados.

Quanto à classificação de estruturas de dados não comprimidas, Jacobson [24] refinou a definição de estrutura de dados implícita e introduziu o conceito de estrutura de dados *sucinta*. Seja n o mínimo teórico de bits de informação necessários para representar um conjunto de dados. Uma estrutura que armazena esses dados é classificada como

- *Implícita*, se utiliza $n + O(1)$ bits de espaço;

- *Sucinta*, se utiliza $n + o(n)$ bits de espaço; e
- *Compacta*, se utiliza $O(n)$ bits de espaço.

Uma estrutura de dados sucinta oferece uma representação eficiente em memória, armazenando uma quantidade de memória sublinear além da ótima.

Para analisar estruturas de dados que utilizam compressão, é necessária a introdução do conceito de *entropia da informação*, e em particular no contexto deste trabalho, a chamada *entropia empírica* de um texto [19].

Considere uma cadeia $X = x_0 \cdots x_{n-1}$ sobre um alfabeto $\mathscr{A} = \{a_0, \ldots, a_{m-1}\}$, e seja c_j a quantidade de ocorrências do caractere a_j em X, para $i = 0, \ldots, m$. A *entropia de ordem* 0 de X é definida por

$$H_0(X) = \frac{1}{n} \sum_{i=0}^{m-1} c_i \cdot \lg \frac{n}{c_i}. \tag{2.1}$$

A entropia assim definida, medida em bits de informação, quantifica a incerteza média sobre os caracteres do texto sem levar em consideração o contexto de cada posição. Esse número também representa a quantidade média de bits necessários para representar cada posição do texto, sendo portanto $n \cdot H_0(m)$ a quantidade esperada de bits para representar a cadeia no total, se não levarmos em conta informação de contexto. Se, em cada posição do texto, qualquer um dos caracteres pode ocorrer com igual probabilidade, independente de outras posições, temos a entropia máxima de $H_0(X) = \lg m$ bits de informação. Esse corresponde ao texto menos previsível (de maior incerteza). Se, no outro extremo, todas as posições do texto são iguais, temos a entropia mínima $H_0(X) = 0$, o que está de acordo com a ideia de que esse é um texto trivial que não apresenta nenhuma imprevisibilidade.

De acordo com a definição 2.1, se considerarmos uma cadeia binária $X = \texttt{ababab} \cdots \texttt{ab}$, a entropia de ordem 0 ainda seria máxima (1 bit por caractere) sendo que, intuitivamente, a cadeia é completamente previsível. Isto ocorre porque sabemos que, após cada ‘a’, temos necessariamente um ‘b’ e vice versa, ou seja, cada posição está completamente determinada se

conhecermos a posição precedente, ou *contexto*. No caso geral, cada posição pode estar mais ou menos influenciada pelas k posições precedentes, pelo que se define a entropia de alta ordem

$$H_k(X) = \frac{1}{n} \sum_{C \in \mathscr{A}^k} |X_C| \cdot H_0(X_C), \tag{2.2}$$

onde, para cada possível contexto C de comprimento k, X_C denota a cadeia obtida através da concatenação de todos os caracteres imediatamente seguintes às ocorrências de C em X. No exemplo corrente, temos apenas dois contextos de comprimento 1, $C = \texttt{a}$ ou $C = \texttt{b}$ e, nesses casos, temos respectivamente $X_{\texttt{a}} = \texttt{bbb}\cdots\texttt{b}$ e $X_{\texttt{b}} = \texttt{aaa}\cdots\texttt{a}$. Daí teríamos $H_1(X) = 0$, o que está de acordo com a intuição que a cadeia é completamente previsível se considerarmos contextos de tamanho 1. Em geral, temos $\lg m \geq H_k(X) \geq H_{k+1}(X)$ para todo $k \geq 0$.

2.3.2 Dicionários indexáveis e Wavelet trees

Na base de praticamente todas as estruturas de dados sucintas, temos os chamados dicionários indexáveis. Um *dicionário indexável* é uma estrutura construída sobre um bitarray[2] $B = b_0 \cdots b_{n-1}$ de tamanho n que oferece as seguintes operações em tempo constante, utilizando $n + o(n)$ bits de memória.

- $rank_x(B,i)$: Retorna o número de posições $j \in [0,i)$ tal que $b_j = x$.
- $select_x(B,i)$: Retorna a posição da i-ésima ocorrência de x em B.
- $pred_x(B,i)$: Retorna a maior posição j menor que i tal que $b_j = x$.
- $succ_x(B,i)$: Retorna a menor posição j maior que i tal que $b_j = x$.

A definição das operações pode ser estendida para x sendo uma cadeia de tamanho maior que um utilizando $B[j : j + |x|] = x$ como comparação.

[2]Usamos indistintamente o termo em Inglês para vetores binários, introduzidos na Seção 2.1.

Grossi et al [21] propuseram a estrutura *Wavelet Tree*, que generaliza a definição das operações de um dicionário indexável, de modo a responder perguntas similares sobre sequências T não necessariamente binárias, definidas a seguir.

- $rank_x(T,i)$: Retorna o número de posições $j \in [0,i)$ tal que $t_j = x$.
- $select_x(T,i)$: Retorna a posição da i-ésima ocorrência de x em T.

A *Wavelet Tree* $W(T,\mathscr{A})$ de uma cadeia T de tamanho n, sobre um alfabeto $\mathscr{A}$ de tamanho m, oferece essas operações em tempo $O(\lg m)$ utilizando $n\lg m + o(n)$ bits de memória.

Para definir a wavelet tree, é necessário introduzir o conceito de *projeção* da cadeia T sobre um alfabeto $\mathscr{A}'$, denotado por $\text{proj}(T,\mathscr{A}')$, definido como, para qualquer subconjunto $\mathscr{A}' \subseteq \mathscr{A}$, a subsequência de T consistindo de todas as suas posições em $\mathscr{A}'$.

A wavelet tree, por fim, é a árvore recursivamente definida por:

$$W(T,\mathscr{A}) = \begin{cases} \perp, & \text{if } |\mathscr{A}| = 1, \\ \begin{array}{ccc} & B(T,\mathscr{A}_0,\mathscr{A}_1) & \\ \swarrow & & \searrow \\ W(\text{proj}(T,\mathscr{A}_0),\mathscr{A}_0) & & W(\text{proj}(T,\mathscr{A}_1),\mathscr{A}_1) \end{array}, & \text{caso contrário,} \end{cases} \tag{2.3}$$

onde

- $\mathscr{A} = \mathscr{A}_0 \cup \mathscr{A}_1$ é uma partição não trivial do alfabeto,
- A raiz consiste em um bitarray $B(T,\mathscr{A}_0,\mathscr{A}_1) = b_0 \cdots b_{n-1}$ de tamanho $n = |T|$, tal que $b_i = 0$, se $t_i \in \mathscr{A}_0$, ou $b_i = 1$, se $t_i \in \mathscr{A}_1$,
- As subárvores esquerda e direita, $W(\text{proj}(T,\mathscr{A}_0),\mathscr{A}_0)$ e $W(\text{proj}(T,\mathscr{A}_1),\mathscr{A}_1)$ correspondem às wavelet trees das projeções de T sobre os subalfabetos $\mathscr{A}_0$ e $\mathscr{A}_1$, respecitivamente, e

- $\perp$ representa uma árvore nula (vazia), sendo ela o caso base para alfabetos unitários.

A Figura 2.2 ilustra uma wavelet tree na qual, o alfabeto é recursivamente particionado em metades. Em cada nó, apenas o bitarray é armazenado, sendo o texto projetado exibido apenas para ilustração. Cada aresta está rotulada pelo subalfabeto sobre o qual a cadeia do nó pai é projetada. Usando essa partição, obtemos uma representação em $n \lg m + o(n \lg m)$ bits sobre a qual as operações de *rank* e *select* podem ser realizadas em tempo $O(\lg m)$ percorrendo um caminho da raiz até uma folha e realizando *rank* e *select* nos bitarrays dos nós nesse caminho. Alternativamente, a estrutura da wavelet tree pode seguir a partição induzida pela Codificação de Huffman [23], caso no qual o espaço da estrutura pode chegar a $nH_0(T) + o(n)$ [37].

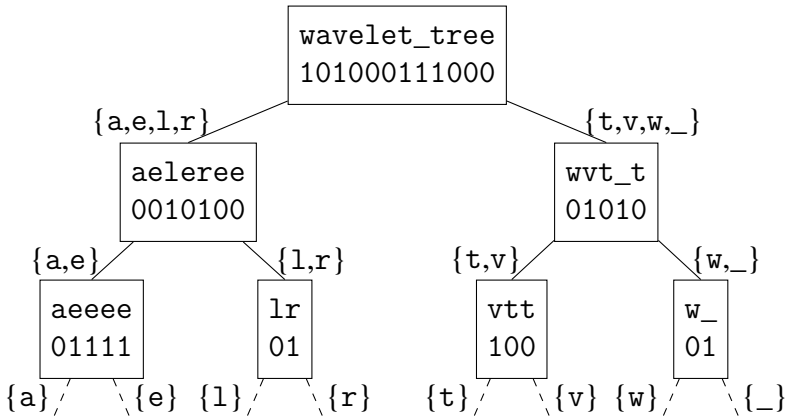

Figura 2.2 Wavelet tree

2.3.3 Vetores de sufixos comprimidos

Um vetor de sufixos de uma cadeia $T = t_0 \cdots t_{n-1}$ é uma permutação $S = \mathrm{SA}(T) = (s_0, \cdots, s_{n-1})$ dos valores $0, \ldots, n-1$. Ocorre que essa informação pode ser representada indiretamente através de um outro vetor $\Phi = (\phi(0), \ldots, \phi(n-1))$, definido por

$$\phi(i) = \begin{cases} S^{-1}[S[i]+1], & \text{se } S[i] < n-1, \\ S^{-1}[0], & \text{se } S[i] = n-1\ , \end{cases} \tag{2.4}$$

onde $S^{-1}[j]$ denota vetor de sufixos inverso, i.e. $S^{-1}[j] = i \iff S[i] = j$. A transformada ϕ associa cada posição i do vetor de sufixos à ordem lexicográfica do sufixo iniciado na posição $s_i + 1$. Por exemplo, na Figura 2.3 temos os valores de S, S^{-1} e Φ para a cadeia $T =$ `banana$`. O valor $S[0] = 6$ indica que o menor sufixo em ordem lexicográfica é o sufixo $T[6:] =$ \$. Como $S[0] = n - 1 = 6$, a definição (2.4) indica que $\phi(0)$ deve ser a posição em S do sufixo $T[0:]$, ou seja $S^{-1}[0] = 4$. De modo similar, para $i = 1$ temos $S[1] = 5$, ou seja o segundo sufixo em ordem lexicográfica é o $T[5:]$. Logo $\phi(1)$ deve ser a ordem do sufixo $T[6:]$, no caso $S^{-1}[6] = 0$.

i	0	1	2	3	4	5	6
T	b	a	n	a	n	a	$
S	6	5	3	1	0	4	2
S^{-1}	4	3	6	2	5	1	0
Φ	4	0	5	6	3	1	2
	$	a			b	n	

Figura 2.3 Transformada ϕ

Repare que, partindo da posição do sufixo $T[0:]$ em S, $i_0 = S^{-1}[0]$, podemos consultar $i_1 = \Phi[i_0]$, obtendo a ordem de $T[1:]$, ou seja $S[i_1] = 1$. Consultando novamente $i_2 = \Phi[i_1]$, obtemos a ordem de $T[2:]$, ou seja $S[i_2] = 2$, e assim sucessivamente, de modo que obtemos uma sequencia $i_0, i_1, i_2, \ldots, i_{n-1}$ com $S[i_k] = k$. Desse modo, podemos computar o valor de $S[i]$ a partir de Φ e i_0, consultando iteradamente $i_k = \Phi[i_{k-1}]$ até que $i_k = i$, quando então saberemos que $S[i = i_k] = k$. No exemplo corrente, suponha que quiséssemos computar $S[i = 5]$. Teríamos então a sequência

$$\begin{aligned} i_0 = 4 &\implies S[4] = 0 \\ i_1 = \Phi[i_0] = \Phi[4] = 3 &\implies S[3] = 1 \\ i_2 = \Phi[i_1] = \Phi[3] = 6 &\implies S[6] = 2 \\ i_3 = \Phi[i_2] = \Phi[6] = 2 &\implies S[2] = 3 \\ i_4 = \Phi[i_3] = \Phi[2] = 5 &\implies S[5] = 4. \end{aligned} \tag{2.5}$$

O processo descrito acima requer tempo $O(n)$ no pior caso, e portanto não há vantagem aparente em representar S através de Φ, uma vez que temos igualmente uma permutação dos índices, porém com tempo de consulta pior. Ocorre que, diferentemente de S, o vetor Φ possui uma propriedade que o torna mais propício à compressão. Repare, na Figura 2.3, que S pode ser particionado em intervalos correspondentes às letras iniciais dos sufixos correspondentes a cada posição. Primeiro, temos o sufixo iniciado em \$, depois os iniciados em `a`, depois os iniciados em `b`, e depois os iniciados em `n`. Esses intervalos estão indicados na última linha da figura. Pode ser facilmente demonstrado que, em cada um desses intervalos, Φ é uma sequencia estritamente crescente, e portanto, mais compressível.

Para demonstrar como comprimir Φ, considere a cadeia $X = x_0 \cdots x_{n-1}$ definida por

$$X[\phi(i)] = T[S[i]]. \tag{2.6}$$

No exemplo corrente, teríamos $X =$ `annb$aa`. Essa cadeia é uma permutação do texto que codifica Φ. Com efeito, se olharmos para X, saberemos, por exemplo, que o caractere `a` ocorre nas posições $0, 5, 6$. Essa é justamente a subsequencia contígua crescente de Φ correspondente ao intervalo dos sufixos iniciados em `a`, como discutido acima. De maneira geral, o valor de $\Phi[i]$ pode ser computado por

$$\Phi[i] = select_{c_i}(X, i - r_i), \tag{2.7}$$

onde $c_i = T[S[i]]$ é o caractere do intervalo que contem a posição i, r_i é o número ocorrências dos caracteres lexicograficamente menores do que c_i em X, e *select* é a operação descrita na Seção 2.3.2. Os valores de c_i e r_i podem ser inferidos a partir de contagens sobre X, mas é comum representar essas contagens numa tabela de tamanho $m \lg m$ bits, onde m é o tamanho do alfabeto. Usando uma wavelet tree para representar X, podemos portanto representar Φ em espaço $(n+m)\lg m + o(n \lg m)$ bits, com consultas em tempo $O(\lg m)$. No caso típico de alfabetos constantes, isso representa um ganho substancial sobre os $n \lg n$ bits do vetor Φ (e por

consequente S) descomprimido.

Apesar dessa representação de S através Φ codificado como uma wavelet tree ser eficiente em espaço, o tempo de consulta a S torna-se $O(n \lg m)$, o que ainda é muito elevado. Para abordar esse problema, Grossi e Vitter [22] propuseram a representação recursiva descrita a seguir.

Inicialmente, seja $S_0 = S$ o vetor de sufixos original e defina S_1 o subvetor (não-contíguo) de S_0 correspondente aos sufixos pares, isto é, iniciados nas posições pares do texto. Considere que as posições de S_0 selecionadas para S_1 estão indicadas num bitarray E_0. Suponha então que tenhamos os vetores $\Phi_0 = \Phi$, E_0, e S_1. A Figura 2.4 reproduz essa situação a partir do exemplo anterior. Se assim for, procedemos da seguinte maneira para computar $S[i] = S_0[i]$. Se $S_0[i]$ for par, o que sabemos através de $E_0[i]$, então obtemos o seu valor diretamente a partir de $S_1[rank_1(E_0)]$. Caso contrário, consultando $j = \Phi[i]$, descobrimos a posição j do sufixo iniciado em $S_0[i]+1$. Como, nesse caso, $S_0[i]$ é ímpar, $S[0]+1$ é par, ou seja $E[j] = 1$, e portanto, podemos obter $S[j] = S[i]+1$ diretamente de S_1, como acima, e daí $S[i] = S[j]-1$. Como a consulta a E e a operação de *rank* podem ser realizadas em tempo constante, segue que conseguimos consultar $S[i]$ em tempo constante.

i	0	1	2	3	4	5	6
T	b	a	n	a	n	a	$
Φ_0	4	0	5	6	3	1	2
E_0	1	0	0	0	1	1	1
S_1	6				0	4	2

Figura 2.4 Estrutura recursiva do vetor de sufixos.

Obviamente, no caso acima, teríamos ainda que representar S_1 além do Φ_0 e do E_0. Logo o espaço total seria $B(S) = n \lg m + o(n \lg m) + n + o(n) + B(S_1) = n(\lg m + 1) + o(n(\lg m + 1)) + B(S_1)$ bits. Representando S_1 diretamente, temos $B(S_1) = n(\lg n - 1)/2$ bits. Porém, considere a cadeia T_1, indêntica a $T = T_0$, mas agrupando os caracteres aos pares de forma que o comprimento da cadeia T_1 seja $n_1 = \lceil n/2 \rceil$, e o tamanho do alfabeto seja $m_1 = \min\{m^2, d_1\}$,

onde d_1 denota o número de caracteres pareados diferentes que ocorrem em T_1. No exemplo $T_1 = \texttt{ba na na \$\_}$ ou, reescrevendo equivalentemente, $T_1 = 1220$, sobre o alfabeto com os $m_1 = d_1 = 3$ caracteres $0 \equiv \$\_ < 1 \equiv \texttt{ba} < 2 \equiv \texttt{na}$. Se considerarmos T_1 definido dessa forma, temos que S_1 é exatamente o vetor de sufixos de T_1, apenas com valores dobrados. Normalizando para o comprimento de T_1, ou seja, dividindo os valores por 2, teríamos $S_1 = (3,0,2,1) = \mathrm{SA}(T_1)$. A partir daí, a mesma representação pode ser aplicada a S_1 recursivamente, obtendo-se Φ_1, E_1 e S_2 e, deste último, Φ_2, E_2,.... No limite, teremos $\approx \lg m$ níveis, sendo que, na prática, essa representação é aplicada por k níveis, até que o vetor S_k seja suficientemente pequeno para ser representado diretamente. O tempo de consulta a S passa a ser então $O(k) = O(\lg m)$.

2.3.4 Árvores de sufixos comprimidas

Sadakane [48] propôs uma representação sucinta da árvore de sufixos, utilizando internamente um array de sufixos comprimido. Utilizando sua representação, a árvore de sufixos de uma cadeia de tamanho n utiliza $6n + o(n) + S_{SA}$ bits de memória, sendo S_{SA} o tamanho do array de sufixos utilizado. Sadakane chama a represenção de árvore de sufixos com funcionalidade total, mantendo que as operações definidas por uma árvore de sufixos são todas respondidas eficientemente enquanto sua representação é sucinta.

Neste trabalho foi desenvolvida uma implementação simplificada, embora ainda sucinta, da reprentação proposta por Sadakane, resultando na utilização de $18n + o(n) + S_{SA}$ bits de memória. A representação, bem como sua implementação, será detalhada no Capítulo 3.

2.4 Grafos de de Bruijn

Seja S um conjunto de cadeias. O Grafo de de Bruijn (GdB) de ordem k, denotado $G_k(S)$, é uma representação construída sobre S. Cada nó em $G_k(S)$ corresponde a uma subsequência

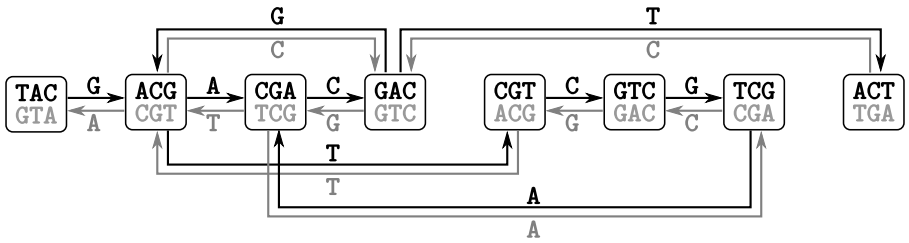

Figura 2.5 GdB de ordem $k = 3$ da cadeia $S =$ `TACGACGTCGACT`. Os nós são rotulados pelos 3-mers (em preto) e temos uma aresta dirigida $x_0x_1x_2 \xrightarrow{y_2} y_0y_1y_2$ sempre que $x_1x_2 = y_0y_1$. Ocorre que, no sequenciamento de DNA, normalmente não sabemos se foi lida a sequência de uma fita ou da sua complementar, no sentido oposto. Portanto, o 4-mer $x_0x_1x_2x_3$ tanto pode representar a aresta $x_0x_1x_2 \xrightarrow{x_3} x_1x_2x_3$ como a aresta $x'_3x'_2x'_1 \xrightarrow{x'_0} x'_2x'_1x'_0$ ($x'j$ denota a base complementar de x_j). Por essa razão é comum identificar o nó (k-mer) com seu complemento reverso (em cinza) e considerar o grafo como bidirecional.

de tamanho k (k-mers) nas cadeias em S. Dois nós são unidos por uma aresta se existe uma sobreposição de tamanho $k-1$ entre seus k-mers correspondentes. Note que isso implica na correspondência de toda aresta com uma $k+1$-mer de S, onde os nós unidos pela aresta correspondem ao prefixo e ao sufixo de tamanho k do $k+1$-mer. A figura 2.5 ilustra a relação entre um conjunto S de tamanho 1 e $G_3(S)$.

2.4.1 GdB para sequenciamento de DNA

Conway e Bromage [14] propuseram uma representação de GdB a partir do seu conjunto de arestas E como um conjunto de $k+1$-mers ordenados lexicograficamente. Cada uma das possíveis arestas é identificada por uma posição de um bitarray BE, de forma que a consulta da presença da aresta no grafo é feita verificando-se se o bit correspondente é igual a 1. Os vértices, em vez de representados explicitamente, são inferidos a partir das arestas. Operações de navegação são definidas sobre as operações *rank* e *select* de um dicionário indexável sobre BE. Aproveitando-se a esparsidade do conjunto das arestas, foram utilizados bitarrays comprimidos que oferecem as operações de *rank* e *select* [16]. Com essa representação, os autores obtiveram cerca de 28 bits por aresta para o GdB do genoma de um humano.

Pell et al [42] propuseram uma representação bastante econômica, que armazena o conjunto de vértices de $G_k(S)$, i.e. as k-mers em S, num *Filtro de Bloom* (FB) [5], sendo as arestas são deduzidas a partir dos vértices. Um FB armazena n elementos utilizando um bitarray B de tamanho m e um conjunto de funções de hash $f_0, \ldots, f_h - 1$. A inserção de um elemento x no FB é dada pela atribuição $B_{f_j(x)} \leftarrow 1$ para toda função de hash f_j. Para verificar a presença de um elemento x no FB, é feita a verificação $B_{f_j(x)} = 1$ para toda função de hash f_j. Se existe algum $B_{f_j(x)} = 0$, x não está contido no FB. No entanto, caso contrário, ainda é possível que x não esteja contido no FB. Existe, portanto, uma taxa de falsos positivos ε, inversamente proporcional à razão $\frac{m}{n}$. Apesar disso, Pell et al sustentam que a representação mantém propriedades adequadas para tarefas como o sequenciamento de DNA.

Chikhi and Rizk [12] estenderam a representação de Pell et al a partir da observação de que apenas uma pequena fração dos 4^k possíveis k-mers são pesquisados no FB durante a navegação e, desses, apenas uma pequena parte acarretam em falsos positivos. Os autores propuseram armazenar explicitamente esses *falsos positivos críticos* (FPc) numa *hash table*, bem como uma tabela de marcadores de *nós complexos*, i.e. nós com grau de entrada ou saída $\neq 1$. Com isso, obteve-se uma representação exata do GdB de tamanho $\approx 1.44nlg(1/\varepsilon) + 16\varepsilon nk$ bits, que, para $k = 27$, resulta em cerca de 13.2 bits pos nó, ajustando-se o tamanho do FB para a taxa de erro mínima, permitindo que o GdB do genoma humano (HapMap: NA18507) seja armazenado em cerca de 6 GigaBytes.

Posteriormente, essa estrutura foi refinada por Salikhov et al [49], que propôs a utilização em cascata de FB para representar o conjunto de FPc. A representação obtida apresentou ser cerca de 30-40% mais eficiente em memória, e 18-30% em tempo de percurso do GdB em testes realizados com 4 FB em cascata.

A representação sucinta apresentada por Bowe et al [7] é bem menos direta e foi inspirada pela *Transformada de Burrows-Wheeler* (BWT). Bowe et al propuseram representar o GdB de uma cadeia S de comprimento n como a cadeia W definida pela concatenação das últimas letras

de todos os seus $k+1$-mers (arestas) ordenados lexicograficamente pelo reverso dos seus k-prefixos. Essa cadeia, juntamente com um bit array de tamanho n e mais as contagens de cada caractere em S, permitem a navegação sobre o GdB com o auxílio das operações *rank* e *select* oferecidas pela Wavelet Tree. No total, essa representação do GdB requer $\approx n\lg m + n + m\lg n$ bits, ou seja, cerca de 2.5 GigaBytes para o mesmo data set do genoma humano (HapMap: NA18507) mencionado acima.

Uma propriedade interessante da representação de Bowe et al é que o fato das arestas ($k+1$-mers) estarem ordenadas lexicograficamente pelos reversos dos seus k-prefixos, (sendo esses justamente os rótulos dos nós dos quais emanam as arestas), então elas também estão automaticamente ordenadas pelos reversos dos prefixos de qualquer tamanho $\leq k$. Ou seja, de certo modo, a representação já inclui os GdB para valores de k menores. Explorando essa característica, Boucher et al [6] estenderam a representação para permitir operações de navegação com o k variável e escolhido *on-the-fly*. Finalmente, essa representação, embora eficiente do ponto de vista de espaço, é assimétrica pois permite que o os rótulos de uma aresta eferente y, seja usada para alcançar o sucessor de αy de um nó $x\alpha$, mas não o sentido contrário. É possível alcançar todos os predecessores de um nó αy, mas para descobrir-se o x, seria preciso retroceder mais k arestas até que esse x passasse a ser o último caractere no nó corrente. Para remediar isso, Belazzougui et al [4] refinaram ainda mais a representação de [6], introduzindo, no lugar da wavelet tree, uma variante bidirecional dessa estrutura [50]. A modificação proposta incorre num custo acrescido de $O(n\lg k)$ bits de espaço (n =número de vértices) sobre a representação base de [7], permitindo todavia navegar sobre o GdB nos dois sentidos e com k variável em tempo $O(\lg k)$ por aresta.

Embora os métodos anteriores já evidenciem a relação entre os GdB e as estruturas de índice, Cazaux et al [9] propuseram-se a investigar mais aprofundadamente essa conexão, em particular com as árvores de sufixos, que é um índice de referência muito estudado [3]. Cazaux et al observaram que o conjunto de k-mers de S pode ser representado por uma *árvore*

de sufixos truncada de altura k [36], e forneceram algoritmos para converter árvores de sufixos generalizadas, truncadas ou não, em estruturas navegáveis para GdB completos ou compactados. Aproveitando a equivalência entre árvores de sufixos e arrays de sufixos enriquecidos com informações sobre maiores prefixos comuns entre todos os pares de sufixos [31], os autores também apresentaram algoritmos para a construção de GdB a partir de arrays de sufixos. Todas essas conversões são lineares em tempo no tamanho do GdB, e a conversão da árvore de sufixos truncada é também linear em espaço. Essa estratégia pode tirar proveito da construção prévia de um desses índices para uma outra tarefa, como a correção de erros de sequenciamento, por exemplo.

CAPÍTULO 3

Uma estrutura de dados para GdB baseada numa árvore de sufixos comprimida

3.1 Árvore de sufixos comprimida

A estrutura suporta as seguintes operações:

- $root(T)$: Retorna a raiz da árvore T.
- $isleaf(T,u)$: Retorna um valor *booleano* representando se o nó u é uma folha em T.
- $firstchild(T,u)$: Retorna o primeiro filho de u.
- $sibling(T,u)$: Retorna o primeiro irmão de u à direita.
- $child(T,u,c)$: Retorna o filho de u percorrendo a aresta cujo primeiro caractere é c.
- $parent(T,u)$: Retorna o pai de u.
- $edge(T,u,i)$: Retorna o i-ésimo caractere na aresta que aponta para u.
- $depth(T,u)$: Retorna o tamanho da cadeia representada por u.
- $edgelength(T,u)$: Retorna o tamanho da aresta que aponta para u.
- $lca(T,u,v)$: Retorna o menor ancestral comum entre u e v.
- $suffixlink(T,u)$: Retorna o nó para o qual a aresta de sufixo de u aponta.

Também são suportadas as seguintes operações em nós implícitos da árvore:

- $isimplicit(T,u)$: Retorna um valor *booleano* representando se o nó u é implícito.
- $implicitfirstchild(T,u)$: Retorna o primeiro filho de u.
- $implicitsibling(T,u)$: Retorna o´primeiro irmão de u à direita.
- $implicitchild(T,u,c)$: Retorna o filho de u percorrendo o caractere c.
- $implicitparent(T,u)$: Retorna o pai de u.
- $char(T,u)$: Retorna o último caractere da cadeia representada por u.

As operações lca e $suffixlink$ funcionam para nós implícitos e explícitos sem que seja necessária a implementação de versões especificas para cada tipo de nó.

3.1.1 Visão geral da estrutura e suas componentes

A árvore de sufixos comprimida é composta pelas seguintes estruturas:

- Array de sufixos comprimido.
- Array de alturas.
  - Array de inteiros ordenados.
- Estrutura de parênteses balanceados.
- Estrutura de RMQ.

3.1.2 Dicionário indexável

Essa estrutura permite realizar as seguintes operações em uma sequência binária B de tamanho n em tempo constante utilizando $n+o(n)$ bits de memória.

- $rank_x(B,i)$: Retorna o número de posições $j \in [0,i)$ tal que $B_j = x$.
- $select_x(B,i)$: Retorna a posição da i-ésima ocorrência de x em B.
- $pred_x(B,i)$: Retorna a maior posição j menor que i tal que $B_j = x$.
- $succ_x(B,i)$: Retorna a menor posição j maior que i tal que $B_j = x$.

A definição das operações pode ser estendida para x sendo uma cadeia de tamanho maior que um utilizando $B_{j\ldots j+|x|} = x$ como comparação.

Embora esse trabalho não inclua a implementação dessa estrutura, ela foi utilizada em algumas das estruturas implementadas.

3.1.3 Array de sufixos comprimido

Essa estrutura permite realizar as seguintes operações sobre uma cadeia T de tamanho n.

- Retornar $SA[i]$ em tempo $O(t_{SA})$.
- Retornar $\Psi[i]$ em tempo $O(t_\Psi)$.
- Retornar T_i em tempo $O(t_{SA})$.

Embora esse trabalho não inclua a implementação dessa estrutura, ela foi utilizada em algumas das operações implementadas. Para isto, foi utilizada a representação proposta por Grossi e Vitter [22].

3.1.4 Array de inteiros ordenados

Essa estrutura permite armazenar uma sequência estática A de n inteiros não negativos ordenados cujo máximo valor é k, oferecendo acesso randômico em tempo constante, utilizando $n+k+o(n+k)$ bits de memória.

A estrutura consiste numa sequência de bits B de tamanho $n+k$ na qual exatamente n bits têm valor 1. As posições dos bits com valor 1 são definidas por $a_i + i$, para todo $i \in [0,n)$. O valor do i-ésimo inteiro em A é dado pela quantidade de bits com valor 0 precedendo o i-ésimo bit com valor 1.

Demonstração. Como A é uma sequência ordenada, $i < j \implies i < j \wedge a_i \leq a_j \implies a_i + i < a_j + j$. Logo, $\left|\{j : b_j = 1, 0 \leq j < a_i + i\}\right| = i$. Portanto, o valor do i-ésimo inteiro é dado por $\left|\{j : b_j = 0, 0 \leq j < a_i + i\}\right| = a_i + i - \left|\{j : b_j = 1, 0 \leq j < a_i + i\}\right| = a_i + i - i = a_i$. □

O acesso é feito em tempo constante utilizando um dicionário indexável com a operação $a_i = rank_0(B, select_1(B, i))$.

3.1.5 Array de alturas

Essa estrutura permite armazenar o LCP (Maior Prefixo Comum) entre sufixos adjascentes no array de sufixos da cadeia T de tamanho n, com acesso em tempo $O(t_{SA})$, utilizando $2n + o(n)$ bits de memória. Através dela também é possível obter a representação da árvore como parênteses balanceados. A definição do array de alturas segue:

$$Hgt[i] = \begin{cases} lcp\left(T_{SA_i}, T_{SA_{i+1}}\right) & 0 \leq i \leq n-2 \\ 0 & i = n-1 \end{cases}$$

Para armazenar a estrutura eficientemente, o array de inteiros ordenadas é usado. No entanto, como Hgt não é garantidamente ordenado, é necessário codifica-lo a fim de obter uma sequência ordenada. A codificação se baseia no seguinte lema:

Lema 3.1. $Hgt[\Psi[i]] \geq Hgt[i] - 1$.

Demonstração. Se $T_{SA[i]} \neq T_{SA[i+1]}$, $Hgt[i] = 0$ e a inequação se sustenta. Caso contrário, pela definição de ordem lexicográfica, $\Psi[i] < \Psi[i+1]$. Logo, existe $i' = \Psi[i] + 1 \leq \Psi[i+$

1]. Pela definição de ordem lexicográfica, o sufixo $T_{SA[\Psi[i]+1]}$ tem um prefixo de tamanho $lcp(T_{SA[i]}, T_{SA[i+1]}) - 1$ em comum com os sufixos $T_{SA[i]+1} = T_{SA[\Psi[i]]}$ e $T_{SA[i+1]+1} = T_{SA[\Psi[i+1]]}$, isto é, $lcp(T_{SA[\Psi[i]]}, T_{SA[\Psi[i]+1]}) \geq lcp(T_{SA[i]}, T_{SA[i+1]}) - 1$. Portanto, $Hgt[\Psi[i]] \geq Hgt[i] - 1$. □

Seja $p = SA^{-1}[0]$.

$$\begin{aligned} Hgt[p] &\geq Hgt[\Psi[p]] + 1 \\ &\geq Hgt[\Psi^k[p]] + k \\ &\geq Hgt[\Psi^{n-1}[p]] + n - 1 \\ &= n - 1 \end{aligned}$$

A igualdade, que limita o valor máximo da sequência, é verdade pois, como $\Psi^{n-1}[p] = n$ e $T_{n-1} =$ '\$', tem-se $Hgt[\Psi^{n-1}[p]] = 0$.

Para recuperar $Hgt[i]$, é necessário computar k tal que $i = \Psi^k[p]$. Pela definição de Ψ,

$$SA[\Psi^k[i]] = SA[i] + k$$

$$\therefore SA[i] = SA[\Psi^k[p]] = SA[p] + k = k$$

Essa codificação nos disponibiza uma sequência de n inteiros ordenados $Hgt[\Psi^k[p]] + k$, para $k = 0, 1 \ldots n-1$. Note que a sequência resultante é uma permutação de $SA[i] + Hgt[i]$. Utilizando o array de inteiros ordenados, ela pode ser armazenada em $n + n + o(n+n) = 2n + o(n)$ bits.

3.1.5.1 Construção da sequência de parênteses balanceados

O array de alturas também é utilizado para construir o bitarray contendo a sequência de parênteses balanceados utilizada na árvore de sufixos.

O algoritmo se baseia na traversia *bottom-up* proposta por Kasai [26]. O algoritmo simula, com o uso do array de sufixos e do array de alturas, uma traversia na árvore de sufixos referente ao texto.

Uma traversia bottom-up é uma iteração numa árvore onde cada nó da árvore é processado exatamente uma vez e todos os filhos de um nó são processados antes dele. No caso da traversia usada, sendo l_u o índice da folha mais à esquerda na subárvore do nó u, r_u o da mais à direita e i_u a ordem em que u aparece na traversia, tem-se $\forall u \forall v : i_u < i_v \implies r_u < r_v \vee (r_u = r_v \wedge l_u > l_v)$.

Algorithm *Construir Parênteses Balanceados*
Input L: Lista dos índices da folha mais à esquerda na subárvore de u na ordem da traversia
R: Lista dos índices da folha mais à direita na subárvore de u na ordem da traversia
H: Lista das distâncias entre u e a folha mais à direita na subárvore de u na ordem da traversia
f: Tamanho de T
Output B: Bitarray de parênteses balanceados
n: Tamanho de B

1 $n \leftarrow 0$
2 $L' \leftarrow$ array de tamanho f preenchido com zeros.
3 $R' \leftarrow$ array de tamanho f preenchido com zeros.
4 **for each** nó u na traversia **do**
5 $\quad L'_u \leftarrow L'_u + 1$
6 $\quad R'_u \leftarrow R'_u + 1$
7 $\quad n \leftarrow n + 1$
8 **for** $i \leftarrow f - 1 \ldots 1$ **do**
9 $\quad L'_u \leftarrow L'_{u+1}$
10 $\quad R'_u \leftarrow R'_{u+1}$
11 $B \leftarrow$ bitarray de tamanho n preenchido com '('
12 **for each** nó u na traversia **do**
13 $\quad k \leftarrow L'_{R_u+1} + R'_{R_u} - H_u - 1$
14 $\quad B_{n-k-1} \leftarrow$ ')'
15 **return** B, n

Algorithm 1 Construção do bitarray de parênteses balanceados.

O Algoritmo 1 baseia-se em encontrar as posições dos parênteses fechados de cada nó na árvore. Ele é divido em duas etapas, cada uma realizando uma traversia.

Inicialmente, são construídos os arrays L' e R' de modo que: L'_i armazene a quantidade de nós cujas folhas mais à esquerda são maiores ou iguais a i; e, analogamente, R'_i armazene a quantidade de nós cujas folhas mais à direita são maiores ou iguais a i. Nesse momento também é feita a contagem de nós da árvore de sufixos.

Em seguida, B é construido. Para encontrar a posição p do parêntese fechado do nó u, é contado: O número de parênteses abertos após p: L'_{R_u+1}; e o numero de parênteses fechados após p: $R'_{R_u} - H_u - 1$. Note que é necessário remover o número H_u de parênteses fechados que são descendentes de u, mas possuem a mesma folha mais à direita na subárvore.

3.1.6 Estrutura de parênteses balanceados

Essa estrutura permite armazenar uma sequência de parênteses balanceados B de tamanho $2n$ com n pares de parênteses correspondentes, oferecendo as seguintes operações em tempo constante:

- $findclose(B,i)$: Retorna o parêntese fechado que corresponda ao parêntese aberto em i.
- $findopen(B,i)$: Retorna o parêntese aberto que corresponda ao parêntese fechado em i.
- $enclose(B,i)$: Retorna o parêntese aberto do par mais interno que contenha o parêntese em i.

A sequência B é dividida (implicitamente) em $\lceil \frac{2n}{m} \rceil$ blocos de tamanho $m = \left\lceil \frac{\lg n}{2} \right\rceil$. Uma série de definições necessárias para a estrutura segue:

- É definido $b(p)$ como o bloco do qual o parêntese p faz parte.
- É definido $\mu(p)$ como o bloco do qual o parêntese correspondente a p faz parte.

- Um parêntese p é dito *afastado* se $b(p) \neq b(\mu(p))$.
- A definição de parêntese *pioneiro* se divide em três partes:
  - Um parêntese aberto afastado p é dito *pioneiro* se $b(\mu(p)) \neq b(\mu(q))$, sendo q o parêntese aberto afastado mais à direita que precede p.
  - Analogamente, um parêntese fechado afastado p é dito *pioneiro* se $b(\mu(p)) \neq b(\mu(q))$, sendo q o parêntese fechado afastado mais à esquerda que sucede p.
  - O primeiro e o último parênteses em B são *pioneiros*.
- Um parêntese p pertence à *família pioneira* de B se p é pioneiro ou $\mu(p)$ é pioneiro.
- É definido o excesso $e(p)$ do parêntese p como a quantidade de parênteses abertos até p menos a quantidade de parênteses fechados até p. Note que, pela definição de sequência de parênteses balanceados, esse valor sempre é não negativo.

A estrutura é composta das seguintes subestruturas:

- Tabelas de preprocessamento
- Família pioneira
- Estrutura recursiva

3.1.6.1 Tabelas de preprocessamento

Um série de tabelas é preprocessada para armazenar valores relacionados à divisão de B em blocos. Todas as tabelas são indexadas por uma sequência binária S de tamanho máximo $m = \left\lceil \frac{lgn}{2} \right\rceil$ representando uma sequência de parênteses e, opcionalmente, um inteiro.

As seguintes tabelas são utilizadas:

- $matches[S,i]$: Armazena a posição do parêntese correspondente ao parêntese na posição i. Se o parêntese não existir em S, armazena uma valor indicativo.

- $enclose[S, i]$: Armazena a posição do parêntese mais interno que contenha o parêntese na posição i. Se o parêntese não existir em S, armazena uma valor indicativo.
- $leftmost\_close\_with\_excess[S, e]$: Armazena a posição do parêntese em S mais à esquerda que possua excesso e. Se o parêntese não existir em S, armazena uma valor indicativo.
- $rightmost\_close\_with\_excess[S, e]$: Armazena a posição do parêntese em S mais à direita que possua excesso e. Se o parêntese não existir em S, armazena uma valor indicativo.
- $rightmost\_far\_open\_until[S, i]$: Armazena a posição do parêntese aberto em S, que não possua correspondente em S, mais à direita cuja posição seja menor ou igual a i. Se o parêntese não existir em S, armazena uma valor indicativo.
- $minimum\_left\_excess[S]$: Armazena o menor excesso em S. Essa tabela é opcional, e pode ser utilizada para armazenar apenas valores não negativos nas tabelas $rightmost\_close\_with\_excess$ e $leftmost\_close\_with\_excess$.

Todas as tabelas podem ser construídas em $O(\sqrt{n}\lg n)$ e ocupam $O\left(\sqrt{n}\lg^2 n\right) = o(n)$ bits de memória.

3.1.6.2 Família pioneira

A família pioneira é utilizada como base da lógica recursiva da estrutura. Para que isso seja possível, é necessária a prova dos seguintes teoremas, bem como a disponibilização das operações descritas em seguida.

Teorema 3.1. *A família pioneira é, ela mesma, uma sequência de parênteses balanceados.*

Demonstração. A família pioneira pode ser obtida com a remoção de todos os pares de parênteses correspondentes da sequência original B. A remoção de um par de parênteses correspondentes de uma sequência de parênteses balanceados resulta numa sequência de parênteses

balanceados. Como a sequência original B é balanceada, a remoção sucessiva de pares de parênteses correspondentes resulta numa sequência de parênteses balanceados. □

Teorema 3.2. *O número de pares de parênteses correspondentes na família pioneira é limitado por* $O(n/\lg n)$.

Demonstração. O grafo G que possui $V = \lceil \frac{n}{m} \rceil$ vértices representados pelo blocos de B e arestas representadas por $(b(p), b(\mu(p))$, para todo par de parênteses correspondentes na família pioneira, é simples e periplanar (planaridade exterior). Logo, G possui no máximo $2V - 3 = 2\lceil \frac{n}{m} \rceil - 3$ arestas. Pela definição de G, a família pioneira possui no máximo $2\lceil \frac{n}{m} \rceil - 3$ pares de parênteses correspondentes. Portanto, o número de pares de parênteses correspondentes na família pioneira é limitado por $O\left(2\lceil \frac{n}{m} \rceil - 3\right) = O\left(\frac{n}{\lg n}\right)$. □

A estrutura inclui um dicionário indexável sobre uma sequência binária indicando quais parênteses pertecem à família pioneira. O dicionário é utilizado para:

- Obter as posições dos parênteses pertencentes à família pioneira a partir das suas posições na sequência original B e vice-versa. O mapeamento é realizado utilizando as funções *rank* e *select* do dicionário.
- Obter a posição do primeiro parêntese pertencente à família pioneira precedendo ou sucedendo uma posição. Essa funcionalidade é realizada utilizando as funções *pred* e *succ* do dicionário.

3.1.6.3 Implementação ingênua

A implementação ingênua é utilizada como caso base da estrutura recursiva. Ela consiste em armazenar as repostas das três operações providas pela estrutura em tabelas com entradas para cada posição da sequência.

O espaço em memória utilizado é de $O(n \lg n)$ bits e o tempo de resposta é constante.

3.1.6.4 Estrutura recursiva

A estrutura de parênteses balanceados é utilizada recursivamente sobre a família pioneira. Como o tamanho da família pioneira é limitado por $O(n/\lg n)$, após dois níveis da recursão o tamanho da sequência de parênteses balanceados é de $O(n/\lg n \lg\lg n)$. A implementação ingênua pode ser utilizada nesse nível de recursão, utilizando

$$
\begin{aligned}
& O\left(\frac{n}{\lg n \lg\lg n} \lg\left(\frac{n}{\lg n \lg\lg n}\right)\right) \\
&= O\left(\frac{n}{\lg\lg n} \frac{\lg\left(\frac{n}{\lg n \lg\lg n}\right)}{\lg n}\right) \\
&= o(n)
\end{aligned}
$$

bits de memória.

3.1.6.5 Operações

Aqui serão descritas as operações realizadas pela estrutura. A fim de simplificar a notação, serão utilizadas as funções:

- $blockbegin(i) = m\left\lfloor \frac{i}{m} \right\rfloor$
- $blockend(i) = \min\left(m\left(\left\lfloor \frac{i}{m} \right\rfloor + 1\right), n\right)$
- $block(i) = B_{blockbegin(i)..blockend(i)}$

Será também utilizado B' como a sequência de parênteses pertencentes à família pioneira em B. A posição p', em B', representa o parêntese em p em B. Note que $p' = rank_1(P, p)$ e $p = select_1(P, p')$, onde P é o bitarray com as posições em B dos parênteses pertencentes à família pioneira.

Algorithm *findclose*
Input B: Sequência de parênteses balanceados
B': Sequência de parênteses pertencentes à família pioneira
P: Dicionário indexável com as posições dos parênteses pertencentes à família pioneira
i: Posição de um parêntese aberto em B
Output Posição do parêntese fechado que corresponde ao parêntese em i

```
1 if matches[block(i), i] ≠ afastado then
2     return matches[block(i), i]
3 j ← pred_1(P, i+1)
4 j' ← rank_1(P, j)
5 μ(j)' ← findclose(B', j')
6 μ(j) ← select_1(P, μ(j)')
7 e(μ(i)) ← e(i) − 1
8 μ(i) ← leftmost_close_with_excess[block(μ(j)), e(μ(i))]
9 return μ(i)
```

Algorithm 2 Operação $findclose$ sobre um bitarray de parênteses balanceados.

Prova do algoritmo: Se o parêntese em i não é afastado, a resposta pode ser obtida através tabela $matches$. Caso contrário, sabendo o bloco e o excesso da posição desejada, pode-se encontra-la utilizando a tabela $leftmost\_close\_with\_excess$. Isso se deve aos seguintes dois teoremas:

Teorema 3.3. *O menor índice que sucede i e tem excesso $e(i)-1$ é $\mu(i)$*

Demonstração. Pelas definições de excesso e sequência de parênteses balenceados, $e(\mu(i)) = e(i)-1$ e $\min_{i<k<\mu(i)} e(k) > e(\mu(i))$. □

Teorema 3.4. *O bloco que contém $\mu(i)$ também contém $\mu(j)$*

Demonstração. Se o parêntese em i pertence à família pioneira, $i = j \therefore b(\mu(i)) = b(\mu(j))$. Caso contrário, existe um número positivo de parênteses afastados em $b(i)$ à esquerda de i cujo correspondente está em $b(\mu(i))$. O mais à esquerda é j. Note que não existe parêntese pertencente à família pioneira entre i e j. □

O algoritmo *findopen* é analogo.

Algorithm *enclose*
Input B: Sequência de parênteses balanceados
B': Sequência de parênteses pertencentes à família pioneira
P: Dicionário indexável com as posições dos parênteses pertencentes à família pioneira
i: Posição de um parêntese aberto em B
Output Posição j do parêntese aberto mais à direita cujo par que contém o parêntese em i

```
1  if encloses[block(i), i] armazena uma posição then
2      if b_{encloses[block(i),i]} = ')' then
3          return findopen(B, P, encloses[block(i), i])
4      return encloses[block(i), i]
5  s ← succ_1(P, i − 1)
6  if b_s = ')' then
7      p ← findopen(B, P, s)
8  else
9      s' ← rank_1(P, s)
10     p' ← enclose(B', s')
11     p ← select_1(P, p')
12 q ← succ_1(P, p)
13 if b(p) = b(q) then
14     j ← rightmost_far_open_util[block(p), q − 1]
15 else
16     j ← rightmost_far_open_util[block(p), blockend(p)]
17 return j
```

Algorithm 3 Operação *enclose* sobre um bitarray de parênteses balanceados.

Prova do algoritmo: Se j ou $\mu(j)$ estiverem no mesmo bloco que i, a resposta é obtida pela tabela *encloses* (e possivelmente uma chamada a $findopen$). Caso contrário, $b(j) < b(i) < b(\mu(j))$.

Pela definição de parêntese pioneiro, existe o parêntese em p pertencente à família pioneira tal que $b(p) = b(j)$ e $b(\mu(p)) = b(\mu(j))$. O parêntese em p é o parêntese pertencente à família pioneira mais à direita cujo par contém o parêntese em i. Se houvesse um parêntese pertencente à família pioneira mais à direita que p cujo par contivesse o parêntese em i, este parêntese seria contido pelo par do parêntese em j, contradizendo sua definição.

Seja s a posição do primeiro parêntese pertencente à família pioneira a partir de i. Se $B_s = \text{`)'}$, o par de parênteses do qual o parêntese s faz parte contém i. Como não há parênteses

pertecentes à família pioneira entre i e s, $p = \mu(s)$. Caso contrário, o parêntese mais à direita cujo par contém o parêntese em p é o mesmo que aquele que contém s. Logo, p pode ser obtido com uma chamada de *enclose* na estrutura rescursiva.

Existe um número positivo de parênteses em $b(i)$ com correspondente em $b(\mu(i))$, sendo p o mais à esquerda e j o mais à direita. Seja q o primeiro parêntese pioneiro após p. Pela definição de parênteses pioneiros, o parêntese em j é o parêntese afastado aberto mais à direita em $b(p)$ antes de q. Caso $b(p) \neq b(q)$, o parêntese em j é o parêntese afasto aberto mais à direito em $b(p)$.

3.1.7 Estrutura de RMQ (Consulta mínima de intervalo)

Essa estrutura permite realizar consultas de valor mínimo em intervalos de uma sequência A de n inteiros com diferenças consecutivas unitárias. A consulta no intervalo $[l,r]$, denotada por $RMQ(A,l,r)$, é respondida por $\arg\min_{l \leq i \leq r} a_i$. Caso existam dois elementos com valor mínimo em $[l,r]$, a resposta é o de menor índice.

A sequência é representada como uma sequência de bits B tal que $b_i = 1 \implies a_i = a_{i-1} + 1$ e $b_i = 0 \implies a_i = a_{i-1} - 1$ A fim de tornar a implementação eficiente em memoria, três implementações são usadas:

- Baseada em tabela: Realizar consultas em intervalos de tamanho limitado.
- Baseada em blocos: Realizar consultas em intervalos que contenham apenas blocos predeterminados inteiros.
- Completa: Realizar consultas para quaisquer intervalos, utilizando as outras implementações.

Pela definição de excesso, ao aplicar essa estrutura no bitarray B de parênteses balanceados, é obtida a funcionalidade de RMQ sobre o array que contém os excessos dos parênteses em B.

3.1.7.1 Estrutura de RMQ baseada em tabela

Essa estrutura permite realizar consultas em intervalos $[l,r]$ de tamanho limitado a m. Para toda sequência de bits $S, |S| \leq m$, a resposta da consulta é armazenada numa tabela $T[S]$. A resposta da consulta $[l,r]$ em B é dada por $l + T[B_{l..r}]$.

O espaço em memória utilizado é de $O(2^m \lg n)$ bits além da representação de B. O tempo de consulta é $O(1)$ se a sequência $B_{l..r}$ puder ser lida como um inteiro.

3.1.7.2 Estrutura de RMQ baseada em blocos

Essa estrutura divide B (implicitamente) em blocos de tamanho m e permite realizar consultas em intervalos $[l,r]$ que contenham apenas blocos inteiros.

Para todo bloco i e inteiro k a resposta da consulta no intervalo $\left[mi,\ m(i+2^k-1)\right]$ é armazenada em $T[i,k]$. A resposta da consulta $[l,r]$ em B é dada por $\min\left\{T\left[\left\lfloor \frac{l}{m} \right\rfloor, k\right], T\left[\left\lfloor \frac{r}{m} - k + 1 \right\rfloor, k\right]\right\}$ onde $k = \left\lfloor \lg \frac{r-l+1}{m} \right\rfloor$.

O espaço em memória utilizado é de $O(n \lg^2 n/m)$ bits além da representação de B. O tempo de consulta é $O(1)$ se k puder ser obtido em $O(1)$.

3.1.7.3 Estrutura de RMQ completa

Essa estrutura inicialmente divide B em $\left\lceil \frac{n}{m} \right\rceil$ blocos grandes de tamanho $m = \left\lceil \lg^3 n \right\rceil$. É utilizada uma estrutura de RMQ baseada em blocos para responder consultas relacionadas aos blocos grandes.

Cada bloco grande é dividido em $\left\lceil \frac{m}{m'} \right\rceil$ blocos pequenos de tamanho $m' = \left\lceil \frac{\lg n}{2} \right\rceil$. Para cada bloco grande é utilizada uma estrutura de RMQ baseada em blocos para responder consultas relacionadas aos blocos pequenos nele contidos.

Por fim, é utilizada uma estrutura de RMQ baseada em tabela para responder consultas de tamanho máximo m'.

A resposta da consulta $[l,r]$ é dada pelo mínimo entre as consultas nos intervalos $[l,s_0-1]$, $[s_0,b_0-1]$, $[b_0,b_1]$, $[b_1+1,s_1]$, $[s_1+1,r]$, onde $[b_0,b_1]$ é o maior intervalo que contém apenas blocos grandes inteiros, s_0 é a menor posição maior que l correspondente ao início de um bloco pequeno e s_1 é a maior posição menor que r correspondente ao fim de um bloco pequeno.

Pela definição de s_0, tem-se $s_0 - l \leq m'$. Analogamente, $r - s_1 \leq m'$. Logo, as consultas nos intervalos $[l,s_0-1]$ e $[s_1+1,r]$ podem ser obtidas pela estrutura baseada em tabela.

Pela definição de b_0, tem-se que as posições s_0 e b_0-1 se encontram no mesmo bloco grande. Analogamente, b_1+1 e s_1 se encontram no mesmo bloco grande. Logo, as consultas nos intervalos $[s_0,b_0-1]$ e $[b_1+1,s_1]$ podem ser obtidas pelas estruturas baseadas em blocos relacionadas às subdivisões em blocos pequenos nos respectivos blocos grandes.

Pela definição de $[b_0,b_1]$, $[b_0,b_1]$ contêm apenas blocos grandes. Logo, a consulta no intervalo $[b_0,b_1]$ pode ser obtida pela estrutura baseada em blocos relacionada aos blocos grandes.

O espaço de memória utilizado é de

$$O\left(\frac{n\lg^2 n}{m} + \left\lceil \frac{n}{m} \right\rceil \frac{m\lg^2 m}{m'} + 2^{m'}\lg n\right)$$

$$= O\left(\frac{n\lg^2 n}{\lg^3 n} + \frac{n\lg^2\left(\lg^3 n\right)}{\left\lceil \frac{\lg n}{2} \right\rceil} + \sqrt{n}\lg n\right)$$

$$= O\left(\frac{n}{\lg n} + \frac{n\lg^2 \lg n}{\lg n} + \sqrt{n}\lg n\right)$$

$$= o(n)$$

bits além da representação de B. O tempo de consulta é $O(1)$.

3.1.8 Operações suportadas

Aqui será descrita a implementação das operações suportadas pela estrutura.

3.1.8.1 Representação do nó

Um nó u da árvore de sufixos é representado pela posição i de seu parêntese aberto na sequência de parênteses balanceados B. Para representar também nós implícitos, é utilizado um inteiro o representando a distância entre u e o nó explícito obtido ao percorrer toda a aresta a qual u pertence. Assim, todo nó u pode ser descrito na forma de dois inteiros (u_i, u_o), onde $u_o = 0$ para nós explícitos da árvore.

Para realizar algumas operações sobre a árvore, é necessário também representar um nó explicito pelos seus valores *preorder* e *inorder*.

Como B é definido a partir da traversia *preorder* da árvore, o valor *preorder* p do nó u pode ser mapeado em tempo constante:

$$p = rank_{(}(B, u_i)$$

$$u_i = select_{(}(B, p)$$

O valor *inorder* i do nó u pode ser mapeado em tempo contante por:

$$i = rank_{()}(B, findclose(B, u_i + 1))$$

$$u_i = parent(B, select_{)(}(B, i) + 1)$$

Embora as operações $rank_{()}$ e $select_{)(}$ possam ser implementadas em $o(n)$ bits extras de memória sobre B, a fim de simplificar a implementação, foram criados dois novos bitarrays de tamanho n, cada um armazenando as posições das subcadeias '()' e ')(' em B.

3.1.8.2 Navegação

Aqui são descritas as implementações das operações relacionadas à navegação. Como as operações a seguir atuam apenas sobre nós explícitos da árvore, $u_o = 0$ para todo u.

- $root(T)$: A raiz u da árvore corresponde ao par de parênteses mais externo de B, logo, sempre é representada por $u_i = 0$.
- $isleaf(T,u)$: Um nó da árvore é uma folha se e somente se sua representação em B é '()'. Logo, u é folha se $B_{u_i} =$ '(' e $B_{u_i+1} =$ ')'.
- $firstchild(T,u)$: O primeiro filho de u é computado por $u_i + 1$.
- $sibling(T,u)$: O irmão de u é computado em tempo constante por $findclose(B,u_i)+1$.
- $child(T,u,c)$: Primeiro, é obtido o primeiro filho de u. Em seguida, itera-se nos irmãos seguintes. A cada passo, é computado o primeiro caractere da aresta que aponta para o nó sendo processado em $O(t_{SA})$ e este é comparado com c. O tempo gasto pela operação é $O(|\mathscr{A}|t_{SA})$.
- $parent(T,u)$: O pai de u é computado em tempo constante por $enclose(B,u_i)$.

As operações lca e $suffixlink$ são descritas em suas próprias seções.

As operações a seguir atuam sobre nós implícitos e explícitos da árvore:

- $isimplicit(T,u)$: Um nó e implícito se $u_o \neq 0$.
- $implicitfirstchild(T,u)$: Se u é implícito, o primeiro filho de u é computado por $(u_i, u_o - 1)$. Caso contrário, encontra-se $v = firstchild(T,u)$. O nó é computado por $(v, edgelength(T,v) - 1)$.
- $implicitsibling(T,u)$: Se $u_o \neq edgelength(T,(u_i,0)) - 1$, u não possui irmão. Caso contrário, sendo $v = sibling(T,(u_i,0))$, o próximo irmão de u é computado por $(v, edgelength(T,v) - 1)$.

- $implicitchild(T,u,c)$: Se u é implícito, u possui exatamente um filho. Se $char(T, firstchild(T,u))$, a resposta é computada por $firstchild(T,u)$. Caso u não seja implícito, sendo $v = child(T,u,c)$, a resposta é computada por $(v, edgelength(T,v)-1)$

- $implicitparent(T,u)$: Se $u_o \neq edgelength(T,(u_i,0))-1$, o pai de u é computado por (u_i, u_o+1). Caso contrário, é computado por $(parent(T,u),0)$.

3.1.8.3 Operações relacionadas ao texto

Aqui serão descritas as operações relacionadas à recuperação da cadeia representado por um nó da árvore de sufixos.

A partir da operação $depth$, encontra-se a cadeia representada pelo nó u. A partir da cadeia, é trivial encontrar as respostas para as outras operações.

Se u é uma folha, a cadeia representada por u é um sufixo do texto original. Logo, sendo $p = preorder(u)$, $depth(T,u)$ é computado por $n - SA[p] + 1$. Caso contrário, sendo $i = inorder(u)$, $depth(T,u)$ é computado por $Hgt[i]$.

Sejam:

$$i = inorder(u)$$
$$d_1 = depth(T, parent(T,u))$$
$$d_2 = depth(T,u)$$

A cadeia representada por u é $T_{SA[i]+d_1 \ldots SA[i]+d_2}$

3.1.8.4 Menor ancestral em comum

Aqui será descrita a operação $lca(T,u,v)$.

Sem perda de generalidade, assuma-se $u_i \leq v_i$. Se u é ancestral de v ou $u = v$, a resposta

é u. Para verificar se u é ancestral de v, é computado em tempo constante $findclose(B,u) \geq findclose(B,v)$.

Caso contrário, a resposta é computada em tempo constante por $parent(B,(RMQ(E,u_i,v_i)+1,0))$, onde E é o array de excessos em B.

Demonstração. Seja $m = RMQ(E,u,v)$. Como m é minimal, $B_m =$ ')' e $B_{m+1} =$ '('. Logo, B_{m+1} é o parêntese aberto de um filho de $lca(T,u,v)$. □

Note que, se u ou v é implícito e u não é ancestral de v, pela definição de nó implícito, $lca(T,u,v) = lca(T,(u_i,0),(v_i,0))$.

3.1.8.5 Aresta de sufixo

Aqui será descrita a operação $suffixlink(T,u)$.

$$
\begin{aligned}
l &= rank_{()}(B,u-1) \\
r &= rank_{()}(B,findclose(B,u))-1 \\
l' &= \Psi[l] \\
r' &= \Psi[r] \\
suffixlink(T,u,v) &= lca(select_{()}(B,l'),select_{()}(B,r'))
\end{aligned}
$$

Demonstração. O algoritmo inicialmente computa l e r, que são, respectivamente, os índices das folhas mais à esquerda e mais à direita na subárvore de u. A folha mais à esquerda é a primeira folha a aparecer em B a partir de u. A folha mais à direita é a última folha antes de $findclose(B,u)$. Seja $h = lcp(T_{SA[l]},T_{SA[r]})$. Pela definição de l e r, h é p tamanho da cadeia representada por u. Trivialmente, $h-1 = lcp(T_{SA[l']},T_{SA[r']})$. Logo, o tamanho da cadeia representada por $lca(select_{()}(B,l'),select_{()}(B,r'))$ é $h-1$, ou seja, é o nó apontado pela aresta

de sufixo de u. □

Para aplicar o algoritmo a um nó implícito u, não é suficiente copiar u_o, pois o tamanho da aresta que aponta para $v = suffixlink(T,u)$ pode ser menor do que o da que aponta para u. Para ajustar a representação, o pai de v_i é atribuído a v_i e o tamanho da aresta apontando para v_i é subtraído de v_o iterativamente, até que a representação seja válida.

3.1.9 Custo de memória

Aqui será calculado o custo total teórico em bits da estrutura.

Para suportar todas as operações, foram utilizadas a seguintes estruturas:

- Array de sufixos: S_{SA}.
- Array de alturas: $2n + o(n)$.
- Bitarray de parênteses balanceados B: $4n + o(n)$.
- Bitarray das posições da família pioneira: $4n + o(n)$.
- RMQ sobre o excesso em B: $o(n)$. Note que o B não precisa ser construído duas vezes.
- Bitarray das posições das cadeias ‘)(’: $4n + o(n)$.
- Bitarray das posições das cadeias ‘()’: $4n + o(n)$.

Custo total: $18n + S_{SA} + o(n)$

3.2 Grafo de de Bruijn

Aqui será apresentada a implementação do GdB utilizando a árvore de sufixos.

Seja G o GdB de ordem k simulado na árvore de sufixos T. Os nós de G são representados pelos nós de T, implícitos e explícitos, que possuem profundidade k. As arestas de G são representadas pelos nós de T, implícitos e explícitos, que possuem profundidade $k+1$. O nó u em T representa a aresta de $parent(T,u)$ a $suffixlink(T,u)$ em G.

Armazenando a folha que representa a cadeia original na árvore, é possível obter o nó inicial para qualquer GdB de ordem $k < n$.

CAPÍTULO 4

Avaliação experimental

Todos os testes foram executados em prefixos da cadeia de DNA disponível em (http://pizzachili.dcc.uchile.cl/t
A representação utilizada para comparação foi a proposta por Bowe [7].

Quando é relevante ilustrar a influência do valor da ordem k do GdB, é utilizado $k \in \{3, 16, 40\}$, consistindo em uma amostra de valores sobre um intervalo utilizado na prática.

Todos os testes foram realizados numa máquina com o processador 64-bit *AMD A10 PRO-7800B R7, 12 Compute Cores 4C+8G* × *4* e 6.9 GiB de memória RAM.

4.1 Tempo e espaço de construção

As figuras 4.1 e 4.2 mostram a eficiência em tempo e memória, respectivamente, das estruturas comparadas. Note que, como a construção da árvore de sufixos independe da ordem k escolhida para o GdB, não é necessário incluir dados para ordens diferentes nesse momento.

Como pode ser visto na figura 4.1, a representação de [7] utiliza-se de uma quantidade significativamente menor de memória. Isto é esperado, devido à simplicidade da representação e pequena quantidade de estruturas internas necessárias. No entanto, é importante notar que, empregando-se a representação de [48] sem as simplificações utilizadas neste trabalho, a complexidade de memória da representação cai de $18n + o(n)$ para $6n + o(n)$, o que pode tornar a estrutura mais próxima de ser competitiva. A figura 4.2 mostra que a representação implementada também não apresenta uma construção eficiente.

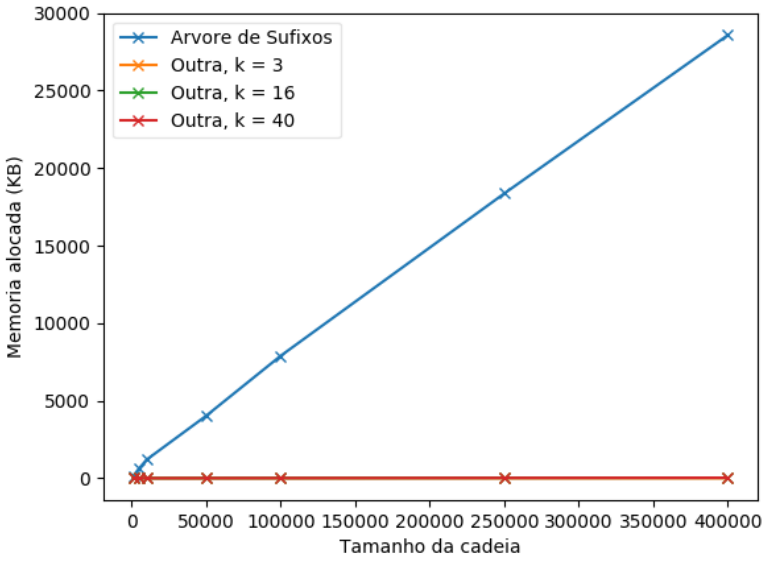

Figura 4.1 Comparação entre a quantidade de memória alocada pelas estruturas implementada e proposta por Bowe.

4.2 Caminho Euleriano

Como discutido no capítulo 1, são utilizados, na montagem de DNA, algoritmos baseados em encontrar um caminho euleriano sobre o GdB. Para demonstrar o uso da estrutura numa situação próxima à prática, utilizou-se um algoritmo de cálculo de caminho euleriano sobre um grafo, ilustrado em Algoritmo 4. O algoritmo percorre o grafo numa traversia em profundidade (DFS) enquanto constroi uma lista baseada nos nós visitados. Desta forma, este algoritmo ilustra a eficiência em tempo da navegação sobre grafo. A fim de minimizar o custo apresentado por operações não relacionadas ao grafo, foi apenas implementada uma DFS para ser utilizada nos testes a seguir. A DFS foi implementada com uma pilha explícita no lugar da recursão.

As figuras 4.3 e 4.5 mostram a eficiência em tempo das estruturas comparadas para GdB de ordem $k \in \{3, 16, 40\}$.

Quando $k = 3$, a representação de Bowe se mostra independente do tamanho da cadeia. Isso se deve ao fato de que, com um k pequeno, a quantidade máxima de nós do GdB é atingida com

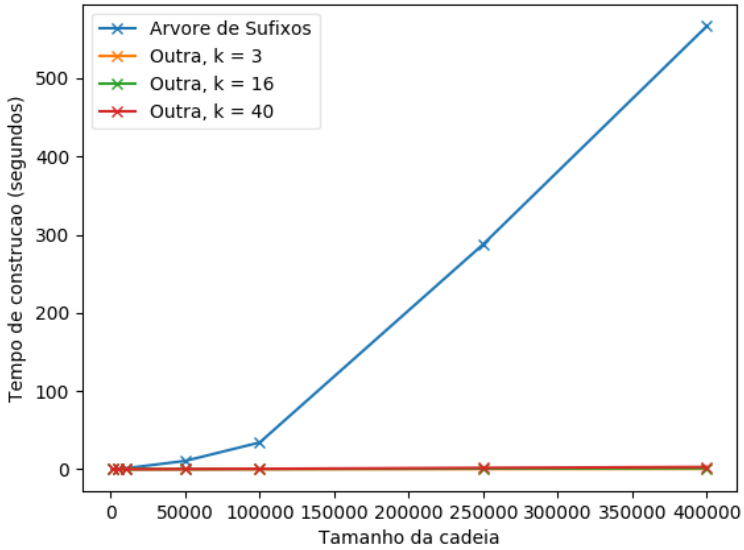

Figura 4.2 Comparação entre o tempo de construção das estruturas implementada e proposta por Bowe.

um tamanho de cadeia menor. Especificamente, o número máximo de nós distintos no GdB de ordem 3 é dado por $|\mathscr{A}|^k = 4^3 = 64$. Como a representação de Bowe apenas constrói o GdB levando em conta esse número máximo de nós, ela não é afetada quando o tamanho da cadeia é aumentado além deste número. A representação implementada, por representar os GdB de qualquer ordem para a cadeia na qual é construída, inclui informação irrelevante quando o k é pequeno demais, e suas operações de navegação são influenciadas por isso.

Quando $k = 16$ ou $k = 40$, consistindo numa ordem grande o suficiente para que o tamanho máximo do GdB não seja relevante, pode-se observar que as duas representações possuem comportamentos similares. Embora nos testes a representação implementada demonstre resultados inferiores, estes ainda são comparáveis, e é possível que se obtenha resultados competitivos ou mais eficientes após realizadas otimizações sobre ela.

Algorithm *Encontrar Caminho Euleriano*
Input G: GdB
l: Ponteiro de um nó na lista E
u: Nó em G
Output E: Lista contendo o caminho euleriano

```
1 for v ← filhos de u em G do
2     if v não estiver marcado then
3         Marque v.
4         l' ← Nó da lista E contendo o caractere da aresta (u, v).
5         Insira l' após l.
6         Encontrar Caminho Euleriano(l', v)
```

Algorithm 4 Construção do caminho euleriano do GdB.

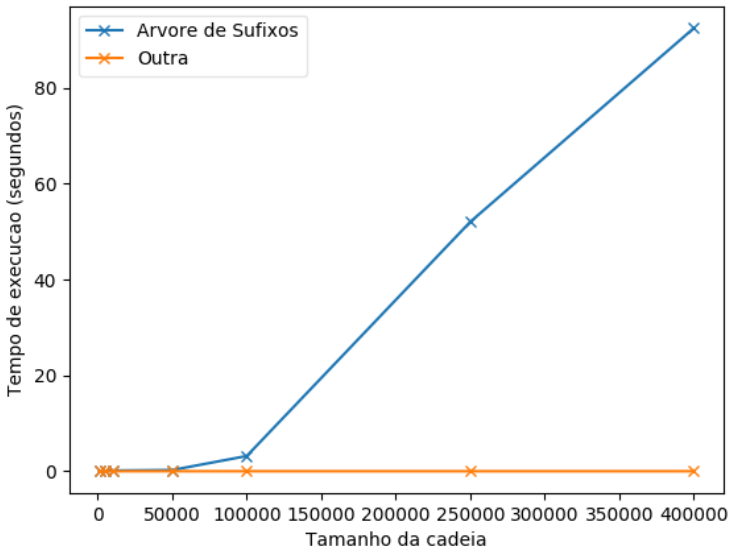

Figura 4.3 Comparação entre o tempo de execução do algoritmo de caminho euleriano das estruturas implementada e proposta por Bowe para $k = 3$.

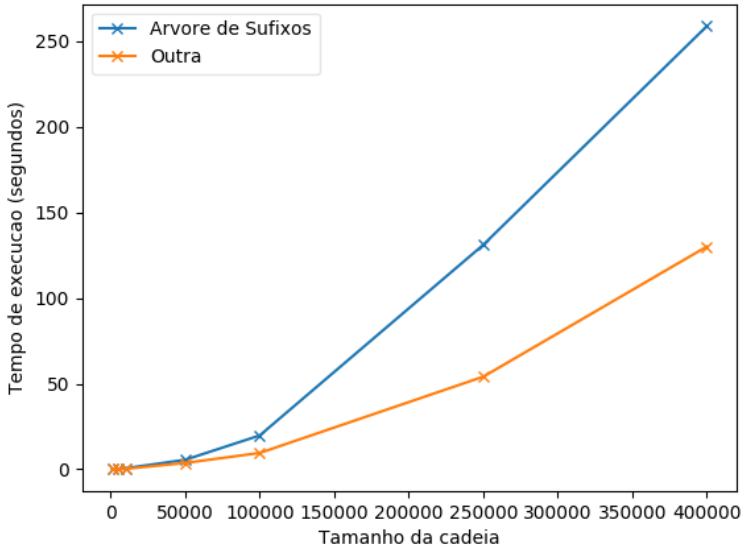

Figura 4.4 Comparação entre o tempo de execução do algoritmo de caminho euleriano das estruturas implementada e proposta por Bowe para $k = 16$.

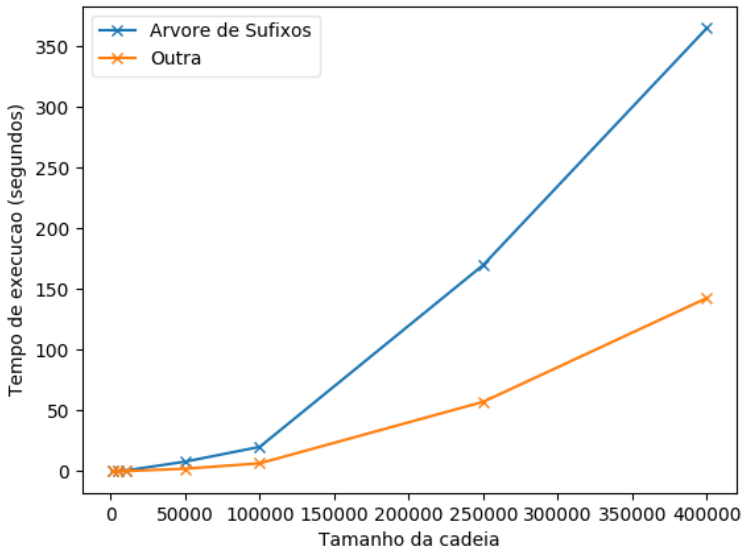

Figura 4.5 Comparação entre o tempo de execução do algoritmo de caminho euleriano das estruturas implementada e proposta por Bowe para $k = 40$.

CAPÍTULO 5

Conclusão e Trabalhos Futuros

5.1 Discussão

Embora a estrutura implementada teoricamente mostre uma boa complexidade em memória enquanto mantem as operações realizadas eficientes, na prática o custo das diversas estruturas internas utilizadas acaba tornando-a inferior à representação de Bowe [7] nos testes realizados. No entanto, é importante notar que não foram realizadas otimizações sobre as estruturas implementadas, e é possível que a representação de Sadakane [48] se mostre competitiva, ou mesmo mais eficiente, após um tratamento mais cuidadoso sobre a implementação. De qualquer forma, a flexibilidade da árvore de sufixos permite a realização de operações não disponíveis por outras representações, como a mudança de ordem durante a execução. Por esse motivo, para aplicações mais complexas, pode ser desejável a implementação desta representação, ainda que seja necessário um custo maior de memória e tempo.

A robustez da árvore de sufixos ainda suporta a busca por representações mais eficientes e simples. Embora a resentação interna da árvore de sufixos seja bastante complexa e acarrete em um overhead considerável, a representação do GdB sobre ela é simples e estimula o estudo de extensões baseadas em suas funcionalidades.

5.2 Desenvolvimentos futuros

Como visto durante o Capítulo 4, pode ser interessante refinar a implementação, obtendo a complexidade de $6n + o(n)$ bits de memória além do array de sufixos utilizado, e otimizando o código em geral. Com isso, é possível tornar a representação baseada em Árvore de Sufixos competitiva com outras representações que possuam um conjunto mais limitado de operações.

Especificamente, a otimização em memória consiste na substituição de três estruturas internas de $4n + o(n)$ bits cada. A primeira delas é o bitarray da família pioneira, utilizado na estrutura de parênteses pioneiros. Utilizando a propriedade de esparsidade dos parênteses pertencentes à família pioneira, [48] propõe uma estrutura que fornece as funcionalidades de um dicionário indexável que nescessita de uma quantidade sublinear de memória.

As outras duas estruturas a serem substituídas são os dicionários indexáveis utilizados sobre o bitarray de parênteses balanceados para as subcadeias '()' e ')('. Estas foram utilizadas na implementação dos mapeamentos dos valores *preorder* e *inorder* para a representação interna dos nós da árvore de sufixos. Na implementação realizada foram criados dois bitarrays de tamanho $4n$, preprocessando as posições das cadeias utilizadas. No entanto, a implementação do dicionário indexável pode ser alterada de modo que os bitarrays extras não sejam necessários.

É necessário, também, estudar o comportamento da estrutura em situações diferentes das vistas no capítulo quatro. O GdB pode ser aplicado a cadeias de proteínas, por exemplo, que são constituídas de alfabetos maiores que as cadeias de DNA. Outro uso diferente da representação é em aplicações que utilizem a funcionalidade de alterar a ordem k do GdB após a construção. Essa funcionalidade, que é praticamente gratuita para a árvore de sufixos, pode ser impossível ou custosa para outras representações.

Além do trabalho diretamente em cima da estrutura, existe a possibilidade de estende-la. Uma opção é estudar a adição da funcionalidade de navegação bidirecional. Isto pode ser obtido com o uso da estrutura chamada *Árvore de Afixos*, uma extensão da árvore de sufixos

que oferece a navegação no sentido contrário do suffix link.

Referências Bibliográficas

[1] Andrés Abeliuk, Rodrigo Cánovas, and Gonzalo Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.

[2] ACM Special Interest Group for Automata and Computability Theory. and SIAM Activity Group on Discrete Mathematics. *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms.* Society for Industrial and Applied Mathematics, 1992.

[3] A Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, volume 12, pages 85–96, 1985. 2.4.1

[4] Djamal Belazzougui, Travis Gagie, Veli Mäkinen, Marco Previtali, and Simon J Puglisi. Bidirectional variable-order de Bruijn Graphs. In *LATIN 2016: Theoretical Informatics, 12th Latin American Symposium*, pages 164–178, 2016. 2.4.1

[5] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. 2.4.1

[6] Christina Boucher, Alex Bowe, Travis Gagie, Simon J. Puglisi, and Kunihiko Sadakane. Variable-Order de Bruijn Graphs. In *Proceedings of the 2015 Data Compression Conference - DCC 2015*, pages 383–392, Snowbird, 2015. 2.4.1

[7] Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *Proceedings of the Workshop on Algorithms in Bioinformatics - WABI 2012*, pages 225–235, Ljubljana, 2012. Springer, Berlin, Heidelberg. 2.4.1, 4, 4.1, 5.1

[8] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. ALLPATHS: De novo assembly of whole-genome shotgun microreads. *Genome Research*, 18(5):810–820, 2008.

[9] Bastien Cazaux, Thierry Lecroq, and Eric Rivals. Linking indexing data structures to de Bruijn graphs: Construction and update. *Journal of Computer and System Sciences*, jul 2016. 1.2, 2.4.1

[10] Mark J Chaisson and Pavel A Pevzner. Short read fragment assembly of bacterial genomes. *Genome Research*, 18(2):324–330, 2008.

[11] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the Representation of De Bruijn Graphs. *Journal of Computational Biology*, 22(5):336–352, may 2015.

[12] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a Bloom Filter. *Algorithms for Molecular Biology*, 8:22, 2013. 2.4.1

[13] Phillip E C Compeau, Pavel A Pevzner, and Glenn Tesler. How to apply de Bruijn graphs to genome assembly. *Nature Biotechnology*, 29(11):987–991, 2011. 1.1.1

[14] Thomas C. Conway and Andrew J. Bromage. Succinct data structures for assembling large genomes. *Bioinformatics*, 27(4):479–486, feb 2011. 1.1.1, 2.4.1

[15] M Crochemore and W Rytter. Text Algorithms. *Text*, page 412, 1994. 2.2

[16] Antonio Fariña, Susana Ladra, Oscar Pedreira, and Ángeles S. Places. Rank and Select for Succinct Data Structures. *Electronic Notes in Theoretical Computer Science*, 236(C):131–145, 2009. 2.4.1

[17] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, Redondo Beach, 2000.

[18] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of Wavelet Trees. *Information and Computation*, 207(8):849–866, 2009.

[19] Travis Gagie. Empirical entropy in context. *Arxiv preprint arXiv:0708.2084*, 2007. 2.3.1

[20] Richard F. Geary, Naila Rahman, Rajeev Raman, and Venkatesh Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, dec 2006.

[21] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms SODA'03*, pages 841–850, Philadelphia, 2003. 2.3.2

[22] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). *Proceedings of the thirty-second annual ACM symposium on Theory of computing - STOC '00*, pages 397–406, 2000. 2.3.3, 3.1.3

[23] D. A. Huffman. A Method for the Construction of Minimum-Redundance Codes. *Proceedings of the I.R.E.*, 40(9):1098–1101, 1952. 2.3.2

[24] Guy Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1989. 2.3.1

[25] Neil C Jones and Pavel Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT Press, 2004.

[26] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. pages 181–192. Springer, Berlin, Heidelberg, 2001. 3.1.5.1

[27] Eric S Lander, Lauren M Linton, Bruce Birren, Chad Nusbaum, Michael C Zody, Jennifer Baldwin, Keri Devon, Ken Dewar, Michael Doyle, William FitzHugh, Roel Funke, Diane Gage, Katrina Harris, Andrew Heaford, John Howland, Lisa Kann, Jessica Lehoczky, Rosie LeVine, Paul McEwan, Kevin McKernan, James Meldrim, Jill P Mesirov, Cher Miranda, William Morris, Jerome Naylor, Christina Raymond, Mark Rosetti, Ralph Santos, Andrew Sheridan, Carrie Sougnez, Nicole Stange-Thomann, Nikola Stojanovic, Aravind Subramanian, Dudley Wyman, Jane Rogers, John Sulston, Rachael Ainscough, Stephan Beck, David Bentley, John Burton, Christopher Clee, Nigel Carter, Alan Coulson, Rebecca Deadman, Panos Deloukas, Andrew Dunham, Ian Dunham, Richard Durbin, Lisa French, Darren Grafham, Simon Gregory, Tim Hubbard, Sean Humphray, Adrienne Hunt, Matthew Jones, Christine Lloyd, Amanda McMurray, Lucy Matthews, Simon Mercer, Sarah Milne, James C Mullikin, Andrew Mungall, Robert Plumb, Mark Ross, Ratna Shownkeen, Sarah Sims, Robert H Waterston, Richard K Wilson, LaDeana W Hillier, John D McPherson, Marco A Marra, Elaine R Mardis, Lucinda A Fulton, Asif T Chinwalla, Kymberlie H Pepin, Warren R Gish, Stephanie L Chissoe, Michael C Wendl, Kim D Delehaunty, Tracie L Miner, Andrew Delehaunty, Jason B Kramer, Lisa L Cook, Robert S Fulton, Douglas L Johnson, Patrick J Minx, Sandra W Clifton, Trevor Hawkins, Elbert Branscomb, Paul Predki, Paul Richardson, Sarah Wenning, Tom Slezak, Norman Doggett, Jan-Fang Cheng, Anne Olsen, Susan Lucas, Christopher Elkin, Edward Uberbacher, Marvin Frazier, Richard A Gibbs, Donna M Muzny, Steven E Scherer, John B Bouck, Erica J Sodergren, Kim C Worley, Catherine M Rives, James H Gorrell, Michael L Metzker, Susan L Naylor, Raju S Kucherlapati, David L Nelson, George M Weinstock, Yoshiyuki Sakaki, Asao Fujiyama, Masahira Hattori, Tetsushi Yada, Atsushi Toyoda, Takehiko Itoh,

Chiharu Kawagoe, Hidemi Watanabe, Yasushi Totoki, Todd Taylor, Jean Weissenbach, Roland Heilig, William Saurin, Francois Artiguenave, Philippe Brottier, Thomas Bruls, Eric Pelletier, Catherine Robert, Patrick Wincker, André Rosenthal, Matthias Platzer, Gerald Nyakatura, Stefan Taudien, Andreas Rump, Douglas R Smith, Lynn Doucette-Stamm, Marc Rubenfield, Keith Weinstock, Hong Mei Lee, JoAnn Dubois, Huanming Yang, Jun Yu, Jian Wang, Guyang Huang, Jun Gu, Leroy Hood, Lee Rowen, Anup Madan, Shizen Qin, Ronald W Davis, Nancy A Federspiel, A Pia Abola, Michael J Proctor, Bruce A Roe, Feng Chen, Huaqin Pan, Juliane Ramser, Hans Lehrach, Richard Reinhardt, W Richard McCombie, Melissa de la Bastide, Neilay Dedhia, Helmut Blöcker, Klaus Hornischer, Gabriele Nordsiek, Richa Agarwala, L Aravind, Jeffrey A Bailey, Alex Bateman, Serafim Batzoglou, Ewan Birney, Peer Bork, Daniel G Brown, Christopher B Burge, Lorenzo Cerutti, Hsiu-Chuan Chen, Deanna Church, Michele Clamp, Richard R Copley, Tobias Doerks, Sean R Eddy, Evan E Eichler, Terrence S Furey, James Galagan, James G R Gilbert, Cyrus Harmon, Yoshihide Hayashizaki, David Haussler, Henning Hermjakob, Karsten Hokamp, Wonhee Jang, L Steven Johnson, Thomas A Jones, Simon Kasif, Arek Kaspryzk, Scot Kennedy, W James Kent, Paul Kitts, Eugene V Koonin, Ian Korf, David Kulp, Doron Lancet, Todd M Lowe, Aoife McLysaght, Tarjei Mikkelsen, John V Moran, Nicola Mulder, Victor J Pollara, Chris P Ponting, Greg Schuler, Jörg Schultz, Guy Slater, Arian F A Smit, Elia Stupka, Joseph Szustakowki, Danielle Thierry-Mieg, Jean Thierry-Mieg, Lukas Wagner, John Wallis, Raymond Wheeler, Alan Williams, Yuri I Wolf, Kenneth H Wolfe, Shiaw-Pyng Yang, Ru-Fang Yeh, Francis Collins, Mark S Guyer, Jane Peterson, Adam Felsenfeld, Kris A Wetterstrand, Richard M Myers, Jeremy Schmutz, Mark Dickson, Jane Grimwood, David R Cox, Maynard V Olson, Rajinder Kaul, Christopher Raymond, Nobuyoshi Shimizu, Kazuhiko Kawasaki, Shinsei Minoshima, Glen A Evans, Maria Athanasiou, Roger Schultz, Aristides Patrinos, and Michael J Morgan. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[28] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, Bicheng Yang, and Wei Fan. Comparison of the two major classes of assembly algorithms: Overlap-layout-consensus and de-bruijn-graph. *Briefings in Functional Genomics*, 11(1):25–37, 2012.

[29] Antoine Limasset, Bastien Cazaux, Eric Rivals, and Pierre Peterlongo. Read mapping on de Bruijn graphs. *BMC bioinformatics*, 17(1):237, 2016.

[30] Moritz Maaß. Linear bidirectional on-line construction of affix trees. *Algorithmica*, 37(1):43–74, 2003.

[31] U Manber and G Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 2013. 2.4.1

[32] Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. 2.2.2

[33] Elaine R Mardis. Next-generation DNA sequencing methods. *Annual Reviews of Genomics and Human Genetics*, 9:387–402, 2008.

[34] J. Ian Munro. Tables. In *Proceedings of the International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42, Hyderabad, 1996.

[35] J. Ian Munro and Venkatesh Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[36] Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theoretical Computer Science*, 304(1-3):87–101, 2003. 2.4.1

[37] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. 2.3.2

[38] Gonzalo Navarro and Eliana Providel. Fast, Small, Simple Rank/Select on Bitmaps. In *Proceedings of the 11th international Symposium on Experimental Algorithms, SEA 2012*, pages 295–306, Bordeaux, 2012.

[39] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings - 2009 Data Compression Conference, DCC 2009*, pages 193–202. IEEE, mar 2009. 2.2.2

[40] Daisuke Okanohara and Kunihiko Sadakane. Practical Entropy-Compressed Rank / Select Dictionary. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments - ALENEX 2007*, pages 60–70, New Orleans, 2007.

[41] Chandra Shekhar Pareek, Rafal Smoczynski, and Andrzej Tretyn. Sequencing technologies and genome sequencing. *Journal of Applied Genetics*, 52(4):413–435, 2011.

[42] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. M. Tiedje, and C. T. Brown. Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 109(33):13272–13277, 2012. 2.4.1

[43] P A Pevzner, H Tang, and M S Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–53, aug 2001.

[44] Mihai Pop and Steven L Salzberg. Bioinformatics challenges of new sequencing technology. *Trends in Genetics*, 24(3):142–149, 2008. 1.1

[45] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms - SODA 2002*, pages 233–242, San Francisco, 2002.

[46] Einar Andreas Rødland. Compact representation of k-mer de Bruijn graphs for genome read assembly. *BMC bioinformatics*, 14(1):313, 2013.

[47] Gustavo A T Sacomoto, Janice Kielbassa, Rayan Chikhi, Raluca Uricaru, Pavlos Antoniou, Marie-France Sagot, Pierre Peterlongo, and Vincent Lacroix. KISSPLICE: de-novo calling alternative splicing events from RNA-seq data. *BMC Bioinformatics*, 13 Suppl 6:S5, 2012.

[48] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. 1.2, 2.3.4, 4.1, 5.1, 5.2

[49] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithms for Molecular Biology*, 9:2, feb 2014. 2.4.1

[50] Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees and bidirectional matching statistics. *Information and Computation*, 213:13–22, 2012. 2.4.1

[51] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven J M Jones, and İnanç Birol. ABySS: A parallel assembler for short read sequence data. *Genome Research*, 19:1117–1123, 2009.

[52] Dirk Strothmann. The affix array data structure and its applications to RNA secondary structure analysis. *Theoretical Computer Science*, 389(1-2):278–294, 2007.

[53] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. 2.2.1

[54] J Craig Venter, Mark D Adams, Eugene W Myers, Peter W Li, Richard J Mural, Granger G Sutton, Hamilton O Smith, Mark Yandell, Cheryl A Evans, Robert A Holt, Jeannine D Gocayne, Peter Amanatides, Richard M Ballew, Daniel H Huson, Jennifer Russo Wortman, Qing Zhang, Chinnappa D Kodira, Xiangqun H Zheng, Lin Chen, Marian Skupski, Gangadharan Subramanian, Paul D Thomas, Jinghui Zhang, George L Gabor

Miklos, Catherine Nelson, Samuel Broder, Andrew G Clark, Joe Nadeau, Victor A McKusick, Norton Zinder, Arnold J Levine, Richard J Roberts, Mel Simon, Carolyn Slayman, Michael Hunkapiller, Randall Bolanos, Arthur Delcher, Ian Dew, Daniel Fasulo, Michael Flanigan, Liliana Florea, Aaron Halpern, Sridhar Hannenhalli, Saul Kravitz, Samuel Levy, Clark Mobarry, Knut Reinert, Karin Remington, Jane Abu-Threideh, Ellen Beasley, Kendra Biddick, Vivien Bonazzi, Rhonda Brandon, Michele Cargill, Ishwar Chandramouliswaran, Rosane Charlab, Kabir Chaturvedi, Zuoming Deng, Valentina Di Francesco, Patrick Dunn, Karen Eilbeck, Carlos Evangelista, Andrei E Gabrielian, Weiniu Gan, Wangmao Ge, Fangcheng Gong, Zhiping Gu, Ping Guan, Thomas J Heiman, Maureen E Higgins, Rui-Ru Ji, Zhaoxi Ke, Karen A Ketchum, Zhongwu Lai, Yiding Lei, Zhenya Li, Jiayin Li, Yong Liang, Xiaoying Lin, Fu Lu, Gennady V Merkulov, Natalia Milshina, Helen M Moore, Ashwinikumar K Naik, Vaibhav A Narayan, Beena Neelam, Deborah Nusskern, Douglas B Rusch, Steven Salzberg, Wei Shao, Bixiong Shue, Jingtao Sun, Zhen Yuan Wang, Aihui Wang, Xin Wang, Jian Wang, Ming-Hui Wei, Ron Wides, Chunlin Xiao, Chunhua Yan, Alison Yao, Jane Ye, Ming Zhan, Weiqing Zhang, Hongyu Zhang, Qi Zhao, Liansheng Zheng, Fei Zhong, Wenyan Zhong, Shiaoping C Zhu, Shaying Zhao, Dennis Gilbert, Suzanna Baumhueter, Gene Spier, Christine Carter, Anibal Cravchik, Trevor Woodage, Feroze Ali, Huijin An, Aderonke Awe, Danita Baldwin, Holly Baden, Mary Barnstead, Ian Barrow, Karen Beeson, Dana Busam, Amy Carver, Angela Center, Ming Lai Cheng, Liz Curry, Steve Danaher, Lionel Davenport, Raymond Desilets, Susanne Dietz, Kristina Dodson, Lisa Doup, Steven Ferriera, Neha Garg, Andres Gluecksmann, Brit Hart, Jason Haynes, Charles Haynes, Cheryl Heiner, Suzanne Hladun, Damon Hostin, Jarrett Houck, Timothy Howland, Chinyere Ibegwam, Jeffery Johnson, Francis Kalush, Lesley Kline, Shashi Koduru, Amy Love, Felecia Mann, David May, Steven McCawley, Tina McIntosh, Ivy McMullen, Mee Moy, Linda Moy, Brian Murphy, Keith Nelson, Cynthia Pfannkoch, Eric Pratts, Vinita Puri, Hina Qureshi, Matthew Re-

ardon, Robert Rodriguez, Yu-Hui Rogers, Deanna Romblad, Bob Ruhfel, Richard Scott, Cynthia Sitter, Michelle Smallwood, Erin Stewart, Renee Strong, Ellen Suh, Reginald Thomas, Ni Ni Tint, Sukyee Tse, Claire Vech, Gary Wang, Jeremy Wetter, Sherita Williams, Monica Williams, Sandra Windsor, Emily Winn-Deen, Keriellen Wolfe, Jayshree Zaveri, Karena Zaveri, Josep F Abril, Roderic Guigó, Michael J Campbell, Kimmen V Sjolander, Brian Karlak, Anish Kejariwal, Huaiyu Mi, Betty Lazareva, Thomas Hatton, Apurva Narechania, Karen Diemer, Anushya Muruganujan, Nan Guo, Shinji Sato, Vineet Bafna, Sorin Istrail, Ross Lippert, Russell Schwartz, Brian Walenz, Shibu Yooseph, David Allen, Anand Basu, James Baxendale, Louis Blick, Marcelo Caminha, John Carnes-Stine, Parris Caulk, Yen-Hui Chiang, My Coyne, Carl Dahlke, Anne Deslattes Mays, Maria Dombroski, Michael Donnelly, Dale Ely, Shiva Esparham, Carl Fosler, Harold Gire, Stephen Glanowski, Kenneth Glasser, Anna Glodek, Mark Gorokhov, Ken Graham, Barry Gropman, Michael Harris, Jeremy Heil, Scott Henderson, Jeffrey Hoover, Donald Jennings, Catherine Jordan, James Jordan, John Kasha, Leonid Kagan, Cheryl Kraft, Alexander Levitsky, Mark Lewis, Xiangjun Liu, John Lopez, Daniel Ma, William Majoros, Joe McDaniel, Sean Murphy, Matthew Newman, Trung Nguyen, Ngoc Nguyen, Marc Nodell, Sue Pan, Jim Peck, Marshall Peterson, William Rowe, Robert Sanders, John Scott, Michael Simpson, Thomas Smith, Arlan Sprague, Timothy Stockwell, Russell Turner, Eli Venter, Mei Wang, Meiyuan Wen, David Wu, Mitchell Wu, Ashley Xia, Ali Zandieh, and Xiaohong Zhu. The Sequence of the Human Genome. *Science*, 291(5507):1304–1351, 2001.

[55] Sebastiano Vigna. Broadword Implementation of Rank / Select Queries. In *Proceedings of the 7th international conference on Experimental algorithms - WEA'08*, pages 1–11, Provincetown, 2008.

[56] James D Watson and Francis H Crick. Molecular structure of nucleic acids; a structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, 1953.

[57] Peter Weiner. Linear pattern matching algorithms. *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, 1973.

[58] Daniel R Zerbino and Ewan Birney. Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821–829, feb 2008.