Relatório Parcial

PIBIC/CNPq

(Edital Nº 04/2020)

Construção de grafos de Brujin sucintos

Orientador: Felipe Alves da Louza

Aluna: Larissa Lima Moraes Aguiar

FEELT – UFU

01/09/2021 – 28/02/2022

Resumo

Este projeto de iniciação científica tem como objetivo investigar algoritmos e estruturas de dados eficientes para a construção de grafos *de Bruijn* sucintos. Em particular, pretendemos estudar estruturas de dados compactas para a representação de grafos *de Bruijn* utilizando pouco espaço em memória, e implementar algoritmos eficientes para a construção dessas estruturas. Nesse relatório apresentamos todas as atividades desenvolvidas na primeira metade do projeto.

Palavras-chave: Grafos *de Bruijn*, algoritmos, estruturas de dados compactas, processamento de cadeias de caracteres.

Sumário

Capítulo 1

Introdução

O grafo *de Bruijn* é uma ferramenta muito importante para o problema de montagem de genomas (*e.g.* [20, 4, 17, 8, 16]), no qual subcadeias de tamanho fixo k (chamados de *k-mers*) de um conjunto de pequenos fragmentos de sequências de DNA (chamados de *reads*), são representadas no grafo *de Bruijn* e sequências contínuas do DNA original (*contigs*) são obtidas a partir de caminhos Eulerianos no grafo [13].

Construir um grafo *de Bruijn* [13] para um grande conjunto de *reads*, geradas durante o sequenciamento de um genoma, e representá-lo de forma sucinta (utilizando pouco espaço em memória) é um desafio cada vez mais importante devido ao crescente volume de dados coletados e disponíveis para análise em Bioinformática [7, 18].

Este projeto de iniciação científica tem como objetivo investigar algoritmos eficientes para a construção de grafos *de Bruijn* sucintos. Em particular, pretendemos estudar a representação BOSS, proposta por Bowe et al. [2], para grafos *de Bruijn*, e investigar a solução apresentada por Egidi et al. [6], na qual a transformada de Burrows-Wheeler (BWT) [3] é utilizada em conjunto com outras estruturas de dados compactas para construir o grafo. Nesse relatório apresentamos todas as atividades desenvolvidas na primeira metade do projeto.

Capítulo 2

Fundamentação teórica

Seja $T = T[1]T[2]\ldots T[n]$ uma cadeia com n caracteres de um alfabeto ordenado Σ de tamanho constante $|\Sigma| = O(1)$. A concatenação de duas cadeias será denotada com o operador ponto ($\cdot$). O símbolo $<$ será utilizado para a relação de ordem lexicográfica entre cadeias e sufixos.

Definição 1 *A subcadeia incluindo o caractere $T[i]$ até o caractere de $T[j]$, para $1 \leq i \leq j \leq n$, será denotada por $T[i, j]$. Uma subcadeia é chamada de própria se $i \neq 1$ ou $j \neq n$, caso contrário a subcadeia corresponde exatamente à cadeia $T[1,n]$.*

Por conveniência, assumimos que a cadeia T sempre termina com um caractere especial $T[n] = \$$, chamado de sentinela (ou terminador), esse caractere não ocorre em outra posição de T e precede todos os caracteres em $T[1,n-1]$.

Definição 2 *Qualquer subcadeia da forma $T[1,i]$ é chamada de prefixo, e qualquer subcadeia da forma $T[i,n]$ é chamada de sufixo de T, abreviado por T_i, para $1 \leq i \leq j \leq n$.*

2.1 Transformada de Burrows-Wheeler

A transformada de Burrows-Wheeler (BWT) [3] é uma transformação reversível de uma cadeia $T[1,n]$ em uma outra cadeia $T^{\mathsf{BWT}}[1,n]$, de mesmo tamanho e com os mesmos caracteres permutados, de forma que caracteres iguais tendem a ficar agrupados em posições consecutivas na cadeia transformada $T^{\mathsf{BWT}}[1,n]$.

A BWT pode ser obtida listando todos as n rotações de $T[1,n]$ em uma matriz $\mathbb{M}$, ordenando-as lexicograficamente em uma nova matriz $\mathbb{M}'$, e obtendo os caracteres da última coluna de $\mathbb{M}'$ como a cadeia $T^{\mathsf{BWT}}[1,n]$, ou seja, os caracteres que precedem as rotações de $T[1,n]$ em ordem. Esse procedimento é ilustrado na Figura 2.1 para $T = \texttt{banana\$}$.

A ordenação de todas as rotações da matriz $\mathbb{M}$ equivale à ordenação de todos os sufixos de $T[1,n]$, uma vez que o terminador $T[n] = \$$ é diferente de qualquer outro símbolo em $T = [1,n-1]$, e nenhuma comparação de rotações irá ultrapassar $T[n] = \$$. Portanto, a BWT pode

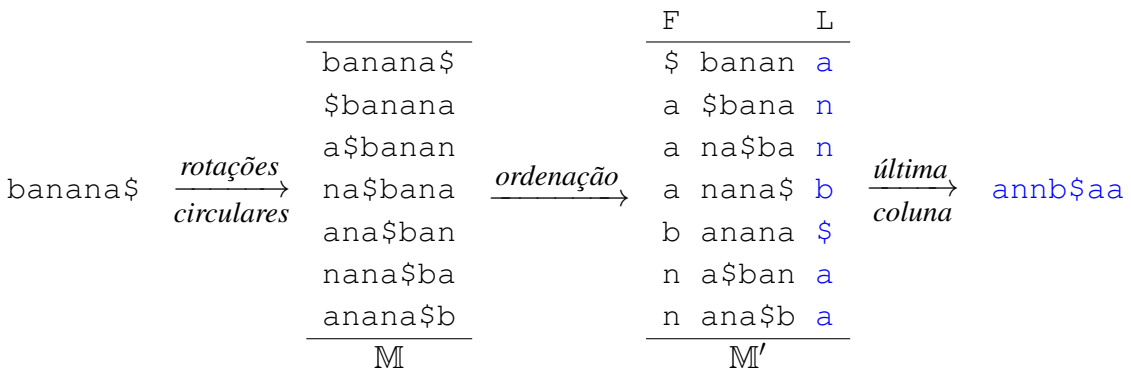


Figura 2.1: A BWT da cadeia $T = \texttt{banana\$}$ é $T^{\mathsf{BWT}} = \texttt{annb\$aa}$.

ser obtida a partir da lista ordenada de todos os sufixos de T. Essa lista corresponde ao *vetor de sufixos* [11], definido a seguir.

Definição 3 *O vetor de sufixos é um vetor de inteiros* $\mathsf{SA}[1,n]$ *com valores no intervalo* $[1,n]$ *que fornecem a ordem lexicográfica dos sufixos de* $T[1,n]$*, tal que:*

$$T[\mathsf{SA}[1],n] < T[\mathsf{SA}[2],n] < \ldots < T[\mathsf{SA}[n],n]$$

Dessa forma, a BWT pode ser obtida em tempo linear a partir de T e de seu vetor de sufixos, tal que:

$$T^{\mathsf{BWT}}[i] = \begin{cases} T[\mathsf{SA}[i]-1] & \text{se } \mathsf{SA}[i] \neq 1 \\ \$ & \text{caso contrário} \end{cases} \tag{2.1}$$

Para a decodificação da BWT são necessárias algumas estruturas, que estão definidas abaixo.

Definição 4 *O vetor de* F *é um vetor de caracteres* $\mathsf{F}[1,n]$ *que armazena em* $F[i]$ *o primeiro caractere do sufixo* $SA[i]$

Definição 5 *O vetor de* L *é um vetor de caracteres* $\mathsf{L}[1,n]$ *que armazena em* $L[i]$ *o último caractere da rotação* $SA[i]$

Definição 6 *O vetor* LF *é um vetor de inteiros* $\mathsf{LF}[1,n]$*, tal que*

$$\mathsf{LF}[i] = j,$$

em que i indica a k-ésima posição de um caractere no vetor $\mathsf{L}[1,n]$ *e* j *é a k-ésima ocorrência do mesmo caractere no vetor* $\mathsf{F}[1,n]$

O vetor de prefixo comum mais longo (LCP) [11] é uma estrutura frequentemente utilizada em conjunto com a BWT para resolver diretamente problemas que envolvem comparações entre cadeias de caracteres.

i	SA	LCP	LF	T^{BWT}	$T[\mathsf{SA}[i],n]$
1	7	0	2	a	$
2	6	0	6	n	a$
3	4	1	7	n	ana$
4	2	3	5	b	anana$
5	1	0	1	$	banana$
6	5	0	3	a	na$
7	3	2	4	a	nana$

Figura 2.2: Vetores SA e LCP e a BWT para $T = banana\$$

Definição 7 *O vetor de LCP é um vetor de inteiros* $\mathsf{LCP}[1,n]$ *que armazena o tamanho do prefixo comum mais longo (*lcp*) entre sufixos consecutivos em* SA*, tal que:*

$$\mathsf{LCP}[i] = \begin{cases} \mathsf{lcp}(T[\mathsf{SA}[i],n], T[\mathsf{SA}[i-1],n]) & \textit{se } i > 1 \\ 0 & \textit{caso contrário} \end{cases} \tag{2.2}$$

O vetor de sufixos e o vetor de LCP podem ser construído em tempo linear utilizando pouco espaço adicional em memória (*e.g.* [14, 12, 9]).

A Figura 2.2 ilustra os vetores SA e LCP e a BWT para a cadeia $T = \texttt{banana\$}$.

2.2 Vetor de bits

Um vetor de bits $\mathsf{B}[1,n]$ é uma estrutura de sequência de bits estática que responde a consultas no vetor com complexidade O(1) e pode também ser utilizado para compressão de dados. Com essa estrutura é possível realizar três operações básicas, sendo elas:

- `acess`(B,i): retorna qual o bit na posição i.
- `rank`(B, i): retorna o número de bits iguais à 1 até a posição i.
- `select`(B, i): retorna o índice do i-ésimo bit igual à 1.

	1	2	3	4	5	6	7	8	9
$B =$	0	1	1	0	0	1	1	0	1

Figura 2.3: Exemplo de vetor de bits

$$acess(5) = 0$$

$$rank(5) = 2$$

$$select(5) = 9$$

2.3 Grafos de Bruijn sucintos

Definição 8 *Um grafo de Bruijn, $G = (V,E)$, para um conjunto de cadeias $\mathbb{T}$ é um grafo direcionado em que cada vértice $v_i \in V$ representa uma subcadeia de tamanho fixo k (k-mer) diferente em $\mathbb{T}$. Para cada $(k+1)$-mer α em $\mathbb{T}$, temos uma aresta $(u,v) \in E$, tal que $\alpha[1,k]$ é representado pelo vértice $u \in U$ e $\alpha[2,k+1]$ por $v \in U$.*

A Figura 2.4 ilustra o grafo *de Bruijn* obtido para o conjunto de cadeias $\mathbb{T} = \{$`TACACT#`, `TACTCG#`, `GACTCA#`$\}$, com o valor de $k = 3$.

Em 2012, Bowe et al. [2] introduziram uma representação compacta baseada na BWT para grafos *de Bruijn*, conhecida como BOSS. Nessa representação, apenas as arestas de saída de cada vértice de $G = (V,E)$ são codificadas em uma cadeia de caracteres $W = c_1, c_2, \ldots, c_{|E|}$, com $c_i \in \Sigma \cup \{\#\}$, e com o auxilio de dois vetores de bits last e W^-, ambos de tamanho $|E|$, operações de navegação no grafo podem ser realizadas em tempo $O(\log \sigma)$ (veja [10, Sec. 9.7]).

Cada símbolo c_i corresponde ao símbolo que precede o i-ésimo menor k-*mer* em $\mathbb{T}$, quando comparados do último para o primeiro símbolo, isto é, $\alpha_i[k], \alpha_i[k-1], \ldots, \alpha_i[1]$. Definimos essa relação de ordem como ordem co-lexicográfica ($<_{rev}$). Portanto, para obter W, todos os k-*mers* distintos em $\mathbb{T}$ são listados em ordem co-lexicográfica, de forma que $\alpha_1 <_{rev} \cdots <_{rev} \alpha_{|E|}$, então $W[i] = c_i$ se e somente se o $(k+1)$-*mer* $c_i\alpha_i$ existir em $\mathbb{T}$, caso contrário, $W[i] = \#$. O espaço total ocupado pela estrutura BOSS é de $4|E| + o(|E|)$ bits.

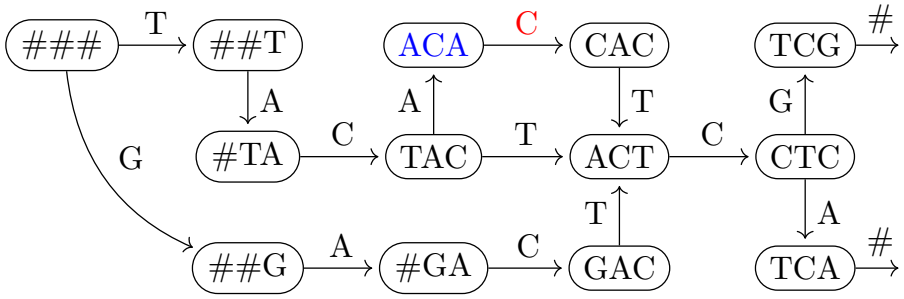


Figura 2.4: Grafo *de Bruijn* para $\mathbb{T} = \{$`TACACT#`, `TACTCG#`, `GACTCA#`$\}$.

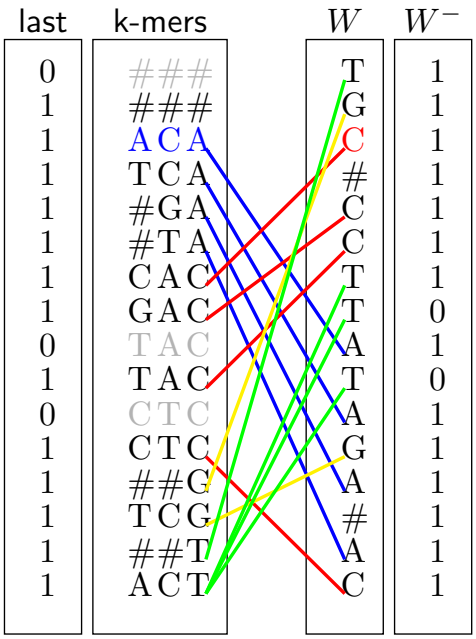


Figura 2.5: Representação BOSS para o grafo *de Bruijn* apresentado na Figura 2.4.

A Figura 2.5 ilustra a representação BOSS para o grafo *de Bruijn* da Figura 2.4. A tabela *k-mers* é apresentada apenas para ilustrar cada *k-mer* correspondente ao vértice v_i que está associado à aresta de saída rotulada com $W[i]$. As linhas coloridas representam as conexões entre os *k-mers* no grafo. Por exemplo, o *k-mer* $v_3 = \texttt{ACA}$ possui uma aresta de saída, rotulada com o símbolo $W[3] = \texttt{C}$, que conecta v_3 ao *k-mer* $v_7 = \texttt{CAC}$, como indicada pela primeira linha em vermelho.

Podemos obter a representação BOSS para o grafo *de Bruijn* de $\mathbb{T}$ a partir da BWT e do vetor de LCP construídos para a concatenação inversa das cadeias em $\mathbb{T}$, como descrito em [5, 6]. Além disso, podemos acrescentar novos subgrafos ao grafo do conjunto $\mathbb{T}$, realizando a união diretamente de suas representações BOSS [6]. Esses dois procedimentos podem ser realizados em tempo $O(|E|)$.

Capítulo 3

Atividades desenvolvidas

Durante a primeira etapa deste projeto, foram estudados conceitos básicos, incluindo as estruturas de dados apresentadas no Capítulo 2. Nesse capítulo apresentamos os algoritmos investigados e implementados. Todas as implementações estão disponível em `https://github.com/larissalmaguiar/Grafos-de-Bruijn.git`.

3.1 Transformada de Burrows-Wheeler

Nessa seção apresentamos um algoritmo para construir a BWT a partir do Vetor de Sufixo, além disso também apresentamos a decodificação da BWT.

3.1.1 Construção da BWT

Para a construção da BWT o algoritmo 1 recebe como parâmetro o Vetor de Sufixo e a cadeia de entrada $T[1,n]$. Primeiramente, ao mesmo tempo em que há a construção do vetor $\mathsf{F}[1,n]$ que armazena em $\mathsf{F}[i]$ o primeiro caractere do sufixo i (Linha 2), geramos também a BWT seguindo a propriedade descrita na Equação 2.1 (Linhas 3-6).

Algoritmo 1: Construção do vetor BWT

```
   input  : SA[1,n] e T[1,n]
   output: BWT[1,n] e F[i]
1  for i ← 1 to n do
2  |  F[i] ← T[SA[i]]
3  |  if SA[i] = 1 then
4  |  |  BWT[i] ←' $'
5  |  else
6  |  |  BWT[i] ← T[SA[i] − 1]
7  |  end
8  end
```

3.1.2 Decodificação da BWT

Para realizar a decodificação da BWT é necessária a construção do vetor $\mathsf{LF}[1,n]$, descrito no Capítulo 2.

Inicialmente, encontramos o número de ocorrências de cada caractere presente na BWT, armazenando esse número em um vetor de inteiros C na posição correspondente ao valor inteiro do caractere (Linha 2). Em seguida, calculamos $COUNT$(Linhas 4-7). Por fim, é possível construir o vetor $\mathsf{LF}[1,n]$, de maneira que a posição i de LF recebe a posição equivalente em que se incia a ocorrência de um caractere na coluna F (Linhas 9-10).

Algoritmo 2: Criação do vetor LF

input : $\mathsf{SA}[1,n]$
output: $\mathsf{BWT}[1,n]$
1 **for** $i \leftarrow 1$ ***to*** n **do**
2 $\quad C[BWT[i]] \leftarrow C[BWT[i]]+1$
3 **end**
4 $COUNT[1] \leftarrow 1$
5 **for** $i \leftarrow 2$ ***to*** σ **do**
6 $\quad COUNT[i] \leftarrow COUNT[i-1]+C[i-1]$
7 **end**
8 **for** $i \leftarrow 1$ ***to*** n **do**
9 $\quad LF[i] \leftarrow COUNT[\mathsf{BWT}[i]]$
10 $\quad COUNT[\mathsf{BWT}[i]] \leftarrow COUNT[\mathsf{BWT}[i]+1$
11 **end**

Tendo o vetor $\mathsf{LF}[1,n]$ calculado é possível realizar a decodificação da BWT, de forma que o Algoritmo 3 tem como entrada os vetores $\mathsf{LF}[1,n]$ e $\mathsf{BWT}[1,n]$ produzindo como saída a cadeia de caracteres de entrada $T[1,n]$.

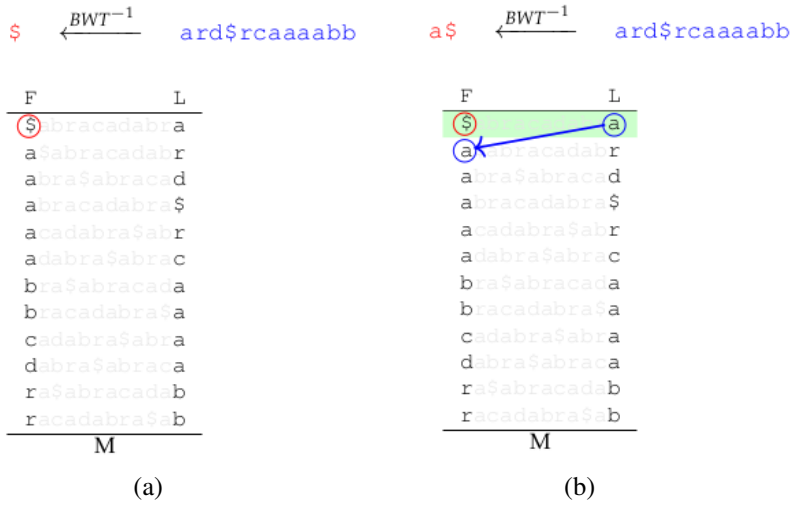
Primeiramente, sabemos que o último caractere da cadeia é o terminador \$, então esse valor é atribuído a última posição em T (Linha 1). Em seguida, a decodificação acontece de trás para frente, de forma que há um looping decrescente (Linha 3), em que $T[i] \leftarrow \mathsf{BWT}[j]$ (Linha 4), isso significa que o caractere presente na BWT será atribuído a posição correta em T, já que $LF[j]$ é igual a posição em que o mesmo carácter da $\mathsf{BWT}[j]$ está em $T[i]$.

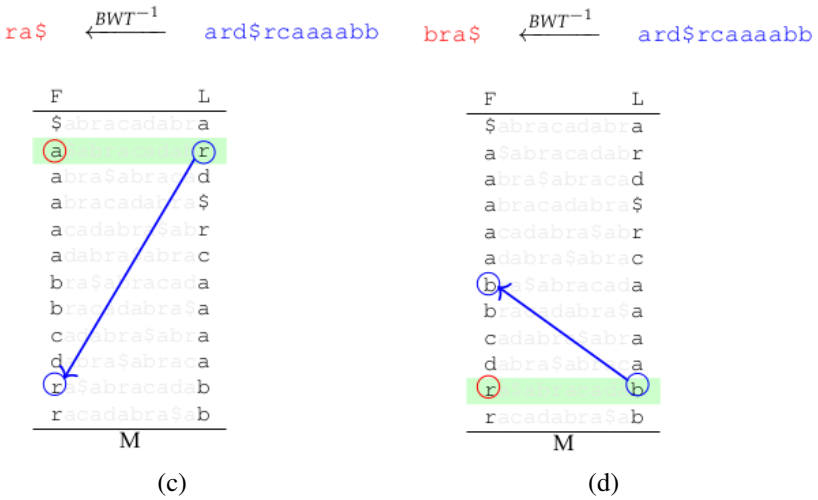
Algoritmo 3: Decodificação da BWT

input : $\mathsf{BWT}[1,n]$ e $\mathsf{LF}[1,n]$
output: $T[1,n]$

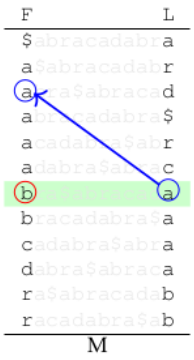
1 $T[n] \leftarrow' \$'$
2 $j \leftarrow 1$
3 **for** $i \leftarrow n-1$ ***to*** 1 **do**
4 $\quad T[i] \leftarrow \mathsf{BWT}[j]$
5 $\quad j \leftarrow \mathsf{LF}[j]$
6 **end**

Abaixo apresentamos um exemplo passo a passo da decodificação da BWT = *ard*$*rcaaaabb*.



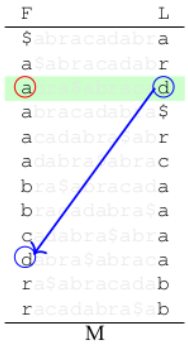


abra$ $\xleftarrow{BWT^{-1}}$ ard$rcaaaabb



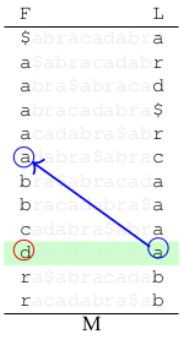
(e)

dabra$ $\xleftarrow{BWT^{-1}}$ ard$rcaaaabb



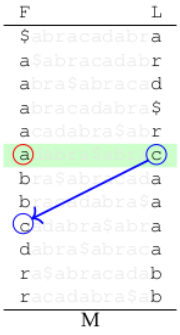
(f)

adabra$ $\xleftarrow{BWT^{-1}}$ ard$rcaaaabb



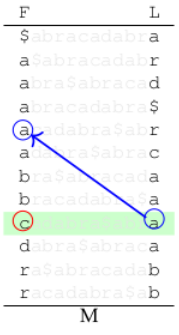
(g)

cadabra$ $\xleftarrow{BWT^{-1}}$ ard$rcaaaabb



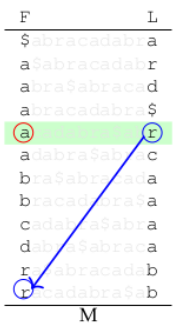
(h)

acadabra$ $\xleftarrow{BWT^{-1}}$ ard$rcaaaabb



(i)

racadabra$ $\xleftarrow{BWT^{-1}}$ ard$rcaaaabb



(j)

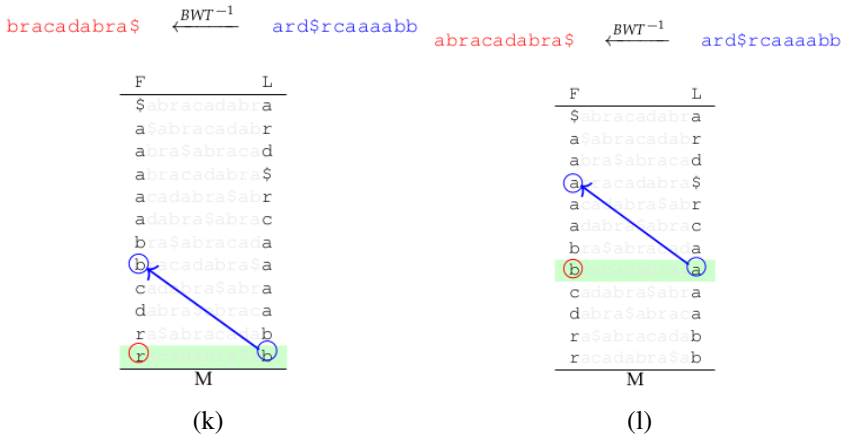


Figura 3.1: Exemplo de decodificação da BWT = *ard*$*rcaaabb*

3.2 Vetor de Bits

3.2.1 Operação Rank

A operação rank tem como objetivo encontrar a quantidade de bits ligados até uma determinada posição do bitvector, para realizar essa função em tempo constante, foi proposto por Raman et al. [15], dividir o bitvector em grandes blocos, de tamanho g, de forma que cada bloco armazene a soma de todos os bits iguais à 1 até a posição em que o bloco termina. Além disso, grandes blocos são divididos em blocos menores, de tamanho p, para ser possível calcular o rank de posições intermediárias. Os blocos menores armazenam o número de bits iguais à 1 de maneira semelhante ao que acontece com os grandes blocos, como está ilustrado na Figura 3.2.

No exemplo da Figura 3.2, os blocos grandes B_g possuem quatro blocos pequenos B_p, e os blocos pequenos possuem quatro bits cada um. Dessa forma, o vetor B_p é referente a quantidade de bits iguais à 1 nos pequenos blocos, quando um bloco grande acaba, os blocos pequenos começam sua contagem novamente. Em seguida, o vetor B_g é referente aos blocos grandes, que guardam a soma de todos os outros blocos anteriores.

	4	8	12	16	20	24	28	32	36	40
$B =$	1000	0010	0000	0110	0000	1010	0000	1011	1000	0001
$B_p =$	1	2	2	4	0	2	2	5	1	2
$B_g =$				4				9		

Figura 3.2: Bitvector e seus respectivos vetores de pequenos e grandes blocos

Para obter o rank de uma posição i, somamos: o valor do início do bloco grande, em que $k = \frac{i}{g} - 1$ é o bloco grande no qual i está incluso; o valor do início do bloco grande até o bloco pequeno que a posição i está inclusa, em que $j = \frac{i}{p} - 1$ é a posição do bloco pequeno que i está; e, por fim, o rank do prefixo i dentro do bloco pequeno, que é calculado pela função *count* que retorna o número de bits iguais a 1 no intervalo do início do bloco pequeno que i está contido até a posição i. Por exemplo, com essa soma temos que

$$rank(B,i) = B_g[i/g - 1] + B_p[i/p - 1] + count_{\mathsf{B}}(i - p + 1, i) \tag{3.1}$$

$$rank(B,24) = 4 + 0 + 2 = 8$$

3.2.2 Operação Select

A operação Select retorna a posição em que o i-ésimo bit igual à 1 está. Para realizar essa operação, uma das soluções é dividir o bitvector em superblocos com o mesmo número de bits igual a 1, o que ocasionará blocos de tamanhos diferentes, alguns serão esparsos e suas respostas serão pré-calculadas, enquanto outros serão densos e é possível calcular a resposta por busca binária ou linear a depender do tamanho do bloco. A Figura 3.3 exemplifica a divisão do bitvector em blocos com o mesmo número de bits iguais a 1.

	14	24	32	41
$S =$	10000010000001	1000001010	00001011	100000011

Figura 3.3: Bitvector dividido em superblocos com o mesmo número de bits iguais a 1

3.3 Montagem de genomas

Atualmente, as tecnologias de leitura de genomas não permitem lê-lo completamente de uma só vez, com isso, a forma de realizar essa leitura é dividir a sequência genômica em *reads*, que são fragmentos de DNA possíveis de serem lidos. As técnicas de montagem genômicas possuem taxas de erros, que as novas tecnologias tem tentado diminuí-las.

Existem diferentes técnicas para realizar a montagem (reconstrução) do genoma original procurando diminuir as taxas de erro. As técnicas existentes até então são divididas em três gerações, em que a primeira é muito lenta, cara e com uma baixa taxa de erro. A segunda é mais rápida, barata e também com uma pequena taxa de erro, no entanto pode ter limitações devido ao pequeno tamanho dos reads produzidos. Já a terceira geração também é barata e rápida, porém com maior taxa de erro do que a primeira geração. A técnica mais usada, atualmente, é uma junção entre a segunda e a terceira geração, que proporcionam uma montagem rápida, barata e com uma baixa taxa de erro.

3.3.1 K-mers

Dentro da bioinformática, *k-mers* são subcadeias de DNA com tamanho k, que são usados para resolução de problemas.

A partir de uma sequência de DNA de tamanho n, podemos extrair $n-k+1$ k-mers dessa sequência. Por exemplo, para a sequência $ACGTACACGTAATT$, podemos extrair os seguintes k-mers:

$$ACG, CGT, GTA, TAC, ACA, CAC, ACG, CGT, GTA, TAA, AAT, ATT$$

3.4 Grafos de Bruijn

Os grafos de Bruijn podem ser utilizados para encontrar uma cadeia que contém subcadeia de tamanho k (problema da supercadeia). Mais tarde, essa ferramenta começou a ser amplamente utilizada na bioinformática para a montagem de genomas.

Os grafos de Bruijn são construídos de maneira que em seus vértices representam *k-mers*. A partir de um vértice, o início do próximo *k-mer* ligado a ele por uma aresta compartilha $k-1$ com o final do anterior, como é possível observar pela Figura 3.4

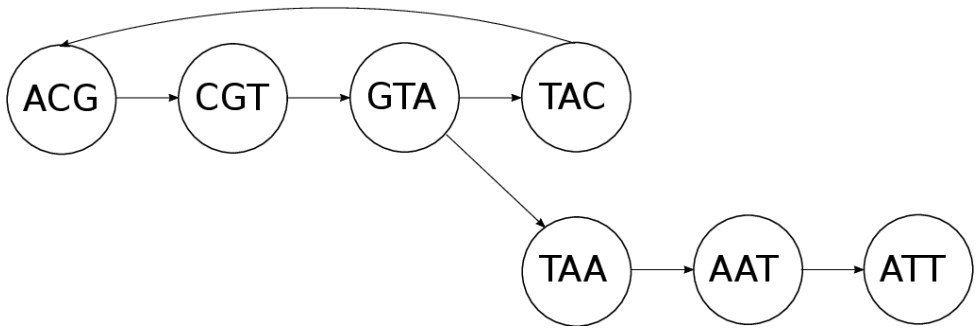


Figura 3.4: Representação de um grafo de Bruijn
Fonte: [19]

Para ser possível realizar a montagem de genoma a partir dos grafos de Bruijn é necessário traçar um caminho entre os vértices, de forma que represente a sequência genética. Podemos observar o caminho do grafo acima (Figura 3.4) na imagem abaixo:

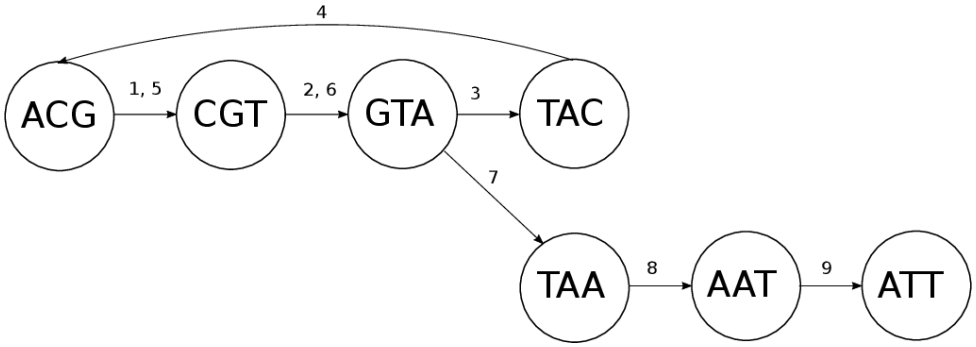


Figura 3.5: Representação do caminho do grafo da Figura 3.4
Fonte: [19]

De um grafo de Bruijn é possível extrair diferentes sequências quando há ciclos no caminho do grafo, para amenizar esse tipo de controvérsia é necessário escolher com cautela o tamanho k dos vértices do grafo.

Para valores pequenos de k, os grafos apresentam um menor número de vértices, porém apresentam mais sobreposições, o que ocasiona mais conexões nos grafos.

Já para valores maiores de k, temos mais vértices, com menores probabilidade de repetição, o que diminui o número de conexões entre os vértices.

Encontrar o melhor valor de k é um trabalho difícil de ser realizado e depende do tamanho dos *reads* de entrada. Dessa forma, atualmente, algumas das melhores soluções realiza a construção de grafos com diferentes tamanhos para k e escolhem o que apresenta o melhor desempenho.

Para realizar a construção dos grafos, todos os *k-mers* são representados como nós e as arestas são adicionadas com base nas sobreposições dos *k-mers*. A forma mais intuitiva de recuperar essas informações de um conjunto de *reads* é percorrer todos os *reads* e os *k-mers* dentro dessas. Essa abordagem é adotada pela maioria dos aplicativos de última geração.

3.5 Formatos de arquivo padrão

3.5.1 Arquivos FASTQ

Arquivos FASTQ armazenam *reads* em registros, cada um composto por quatro linhas consecutivas, conforme descrito a seguir:

Linha 1 armazena o identificador do registro, e inicia com o símbolo @.

Linha 2 armazena uma cadeia de caracteres $T[1,n]$ com os pares de base (símbolos de DNA) da *read* sequenciada.

Linha 3 inicia como o símbolo $+$, e pode opcionalmente ser seguida pelo mesmo identificado de registro da Linha 1.

Linha 4 armazena uma cadeia de caracteres $Q[1,n]$ com indicadores de qualidade do sequenciamento da *read* presente na Linha 2.

A Figura 3.6 apresenta um exemplo de arquivo FASTQ com 4 registros. Cada *read* está destacada em amarelo na figura.

```
@HWI-ST928:79:C0GNWACXX:6:1101:1184
AGTTAGGACTATTCGAACATTATGTCACAAACGTGATGTCACAAAGCCGAATTGTCTGGAGTTAAGA
+
@C@FDEDDHHGHHJIIGGHJJIJGIJIHGIIFGEFIIJJJGHIGGF@DHEHIIIIJIIGGIIIGE@C
@HWI-ST928:79:C0GNWACXX:6:1101:1185
ATTGGGCACAGACGGAGTAGGGCAGCCTTACGTACAGATACAGATACAAACGAGAGACCAAATCATA
+
@@@DDDDDHHHFBGIIIH>>HH@CFHGEHG?FDFDHGIII??BBGGAGHIFGBE@A;AEEDEEE>@C
@HWI-ST928:79:C0GNWACXX:6:1101:1186
CATTCATTTATTCCATTCATCGAGTGAGTTTTATTAAATGAGCACGTAGACCATTCTTCGTTTTTTT
+
CCCFFFFFHGHHHJEHHIGJJJIIIHCHHIJJFIIIGEHHGIFHIIJIHFIHHIJJJIIHHIJJJJH
@HWI-ST928:79:C0GNWACXX:6:1101:1187
TGAATGCAGTGAGTGGTTAAATAAAAATGTTAAAGTTTGAGAGGTCCATTGATAAAACCGCAAANAT
+
@@@FFFFFHHHDHEGIFFDEGHGEIJIIGIIGGGIBGIGCFBGIGHJIFIIJJEGHGBGHGEGI!GJ
```

Figura 3.6: Exemplo de arquivo FASTQ

3.5.2 Arquivos GFA

Os arquivos GFA (Graphical Fragment Assembly) são usados para descrever grafos de Bruijn, a primeira linha do arquivo é o cabeçalho que começa com a letra H e possui campos opcionais no restante da linha. No restante das linhas as arestas do grafo são marcadas pela letra S e são descritas da seguinte maneira:

'S' ID k-mer [tamanho] [contador]

Além disso, há linhas que inciam com o carácter 'L' e representam o link das sequências, em que $+/-$ são usados para indicar se é possível extrair o complemento reverso da sequência.

A Figura 3.7 apresenta um exemplo de arquivo GFA.

H	VN:Z:1.0				
S	185	ATATA	LN:i:5	KC:i:1310	
L	185	+	492	+	4M
L	185	-	185	+	4M
S	316	CGACA	LN:i:5	KC:i:1223	
L	316	+	211	-	4M
L	316	-	94	-	4M
S	268	CATGG	LN:i:5	KC:i:1217	
L	268	-	268	+	4M
L	268	-	77	-	4M

Figura 3.7: Exemplo de arquivo GFA

Capítulo 4

Próximos passos

Inicialmente, vamos estudar a representação sucinta BOSS, proposta por Bowe et al. [2], para grafos *de Bruijn*. Estudaremos as principais operações de navegação no grafo diretamente pela estrutura BOSS.

Em seguida, vamos investigar o problema de construção da representação BOSS. Pretendemos desenvolver uma solução *naive* para construir a representação BOSS por meio da ordenação direta dos *k-mers* utilizando o algoritmo de ordenação *Radix sort* [1]. Depois, vamos investigar a solução apresentada por Egidi et al. [6], na qual a representação BOSS é construída a partir da BWT e o vetor de LCP para o conjunto de *reads*.

Por fim, serão realizados testes comparativos com os algoritmos investigados utilizando dados reais de sequências de DNA. Esse processo de validação será feito por meio de análise teórica e experimentos práticos com dados abertos, como os obtidos em:

1. `https://www.ncbi.nlm.nih.gov/genbank/`
2. `http://www.ensembl.org/`
3. `http://pizzachili.dcc.uchile.cl/`

Todos os algoritmos serão implementados em `C` ou `C++`, suas saídas serão comparadas para garantir que os códigos foram escritos corretamente, e os códigos-fonte serão disponibilizados de forma livre em um repositório aberto, como em `https://github.com/larissalmaguiar/Grafos-de-Bruijn`.

Capítulo 5

Cronograma

As atividades previstas no projeto (**A1**-**A3**) foram cumpridas conforme o planejado. Atualmente o projeto encontra-se no desenvolvimento da atividade **A4**.

A1. Estudar conceitos básicos, algoritmos e estruturas de dados para cadeias de caracteres.

A2. Estudar as operações de rank e select em vetores de bits.

A3. Estudar o problema de montagem de genomas e grafos *de Bruijn*.

A4. Estudar a representação BOSS para grafos *de Bruijn* e as principais operações de navegação na estrutura BOSS.

A5. Implementar uma solução *naive* para construir o grafo *de Bruijn* sucinto através da ordenação direta de todos os *k-mers* com o algoritmo *Radix sort*.

A6. Implementar a solução apresentada por Egidi et al. [6] para construir o grafo *de Bruijn* sucinto utilizando a BWT em conjunto com o vetor de LCP.

A7. Realizar testes comparativos com os algoritmos implementados em **A4** e **A6**.

A8. Escrever relatório de acompanhamento.

Além dessas atividades, também está previsto a escrita e divulgação dos resultados do projeto através de um artigo científico em veículo adequado.

Tabela 5.1: Cronograma de execução das atividades (em meses).

Atividade	1	2	3	4	5	6	7	8	9	10	11	12
A1	●	●										
A2		●	●	●								
A3			●	●	●							
A4					●	◔	○					
A5						○	○	○	○			
A6								○	○	○		
A7									○	○	○	
A8						●						○

Referências Bibliográficas

[1] Andersson, A. and Nilsson, S. (1998). Implementing radixsort. *ACM J. Exp. Algorithmics*, 3:7.

[2] Bowe, A., Onodera, T., Sadakane, K., and Shibuya, T. (2012). Succinct de Bruijn graphs. In *Proc. International Workshop on Algorithms in Bioinformatics (WABI)*, volume 7534, pages 225–235. Springer.

[3] Burrows, M. and Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report.

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

[5] Egidi, L., Louza, F. A., Manzini, G., and Telles, G. P. (2018). External memory BWT and LCP computation for sequence collections with applications. In *Proc. International Workshop on Algorithms in Bioinformatics (WABI)*, pages 10:1–10:14.

[6] Egidi, L., Louza, F. A., Manzini, G., and Telles, G. P. (2019). External memory BWT and LCP computation for sequence collections with applications. *Algorithms for Molecular Biology*, 14(1):6:1–6:15.

[7] El-Metwally, S., Hamza, T., Zakaria, M., and Helmy, M. (2013). Next-generation sequence assembly: four stages of data processing and computational challenges. *PLoS computational biology*, 9(12):e1003345.

[8] Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., and McVean, G. (2012). De novo assembly and genotyping of variants using colored de Bruijn graphs. *Nature Genetics*, 44(2):226–232.

[9] Louza, F. A., Gog, S., and Telles, G. P. (2017). Optimal suffix sorting and LCP array construction for constant alphabets. *Inf. Process. Lett.*, 118:30–34.

[10] Mäkinen, V., Belazzougui, D., Cunial, F., and Tomescu, A. I. (2015). *Genome-Scale Algorithm Design*. Cambridge University Press.

[11] Manber, U. and Myers, E. W. (1993). Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948.

[12] Nong, G. (2013). Practical linear-time O(1)-workspace suffix sorting for constant alphabets. *ACM Trans. Inform. Syst.*, 31(3):1–15.

[13] Pevzner, P. A., Tang, H., and Waterman, M. S. (2001). An eulerian path approach to dna fragment assembly. *Proc. Natl. Acad. Sci.*, 98(17):9748–9753.

[14] Puglisi, S. J., Smyth, W. F., and Turpin, A. H. (2007). A taxonomy of suffix array construction algorithms. *ACM Comp. Surv.*, 39(2):1–31.

[15] Raman, R., Raman, V., and Rao, S. (2007). Succinct indexable dictionaries with applications to encoding *k*-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4).

[16] Rizzi, R., Beretta, S., Patterson, M., Pirola, Y., Previtali, M., Vedova, G. D., and Bonizzoni, P. (2019). Overlap graphs and de bruijn graphs: data structures for de novo genome assembly in the big data era. *Quant. Biol.*, 7(4):278–292.

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

[18] Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., and Robinson, G. E. (2015). Big data: Astronomical or genomical? *PLOS Biology*, 13:1–11.

[19] Wollaert, L. (2018). *Efficient de Bruijn graph construction using suffix trees*. Master dissertation, Ghent University.

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