
Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

# INDEXAÇÃO E BUSCA DE SEQUÊNCIAS BIOLÓGICAS ATRAVÉS DE COMBINAÇÕES DE MINIMISERS DE COMPRIMENTO VARIÁVEL

Diogo Oliveira Rodrigues

TRABALHO DE GRADUAÇÃO

Recife
2019

Universidade Federal de Pernambuco
Centro de Informática

Diogo Oliveira Rodrigues

# INDEXAÇÃO E BUSCA DE SEQUÊNCIAS BIOLÓGICAS ATRAVÉS DE COMBINAÇÕES DE MINIMISERS DE COMPRIMENTO VARIÁVEL

*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
2019

# RESUMO

Um problema algorítmico básico consiste em procurar ocorrências aproximadas de um *padrão* num *texto* de muito maior tamanho. Quando o texto é fixo, um *índice* é construído de forma a permitir que ocorrências de padrões sejam nele encontradas sem a necessidade de percorrê-lo por completo. Em aplicações com grandes volumes de dados, como em Biologia Computacional, índices completos tradicionais como árvores de sufixos ou vetores de sufixos, que representam todas as subcadeias do texto, podem requerer um espaço proibitivo. Alternativamente, alguns índices parciais recentemente propostos representam apenas um subconjunto de subcadeias de tamanho fixo $k$, chamados *minimisers*. Esses índices são combinados a heurísticas de busca que permitem localizar ocorrências aproximadas de um padrão a partir de extensões de ocorrências exatas de minimisers de um tamanho fixo, chamadas 'sementes'. O trabalho recente de Costa [1] demonstrou a vantagem de utilizar minimisers de tamanho variável como sementes para a localização mais precisa das ocorrências do padrão. No presente projeto, estendemos e refinamos esses métodos. Mais especificamente, propomos um método eficiente para a construção *online* do índice, formalizamos o problema da escolha das combinações ótimas de sementes que levem em conta a soma total de seus comprimentos, respeitando a ordem e distâncias relativas entre elas com respeito ao padrão e ao texto, e propomos um algoritmo polinomial exato para o problema descrito. Os métodos foram implementados e analisados experimentalmente, e os resultados demonstram a eficiência e eficácia da abordagem proposta.

**Palavras-chave:** Casamento aproximado de texto, seed&extend, índices parciais, minimisers, Biologia computacional

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ALGORITMOS

# SUMÁRIO

CAPÍTULO 1

# INTRODUÇÃO

## 1.1 MOTIVAÇÃO

Um problema básico de processamento de texto, com aplicação em diversas áreas, consiste em procurar ocorrências aproximadas de uma cadeia, ou *padrão*, $P$ num texto $T$ (ou coleção de textos $\mathcal{T}$) de muito maior tamanho. Por exemplo, em Biologia Computacional, costuma-se alinhar um determinado fragmento de sequência de DNA a um genoma ou conjunto de genomas de referência para identificar regiões homólogas em outras espécies.

Quando o texto é fixo, um *índice* $I(T)$ é construído de forma a permitir que ocorrências de padrões sejam nele encontradas sem a necessidade de percorrer o texto por completo. Um dos índices mais simples consiste numa tabela contendo ponteiros para as ocorrências no texto de subsequências de comprimento fixo $k$, ou $k$*-mers*. Para procurar $P$ em $T$, primeiro consultamos a tabela para descobrir as localizações no texto de um $k$-mer presente no padrão, chamado *semente* (*seed*), e logo confirmamos a ocorrência de $P$ em cada uma dessas posições, estendendo o alinhamento original da semente pelos flancos. A essa estratégia dá-se o nome de *seed&extend*.

Representar todos os $k$-mers do texto pode requerer um espaço proibitivo, como é o caso de genomas. Schleimer [2] e, independentemente, Roberts [3] propuseram a utilização de apenas um subconjunto de $k$-mers, chamados *minimisers* para construção de índices de tamanho reduzido. Li [4] se baseou nesses índices para propor uma ferramenta de mapeamento de sequência de DNA, chamada *Minimap*.

O trabalho recente de Costa [1] demonstrou a vantagem de utilizar minimisers de tamanho variável como sementes para a localização mais precisa das ocorrências do padrão, em contraste à utilização de minimisers de tamanho único como é feito pelo Minimap. Esse trabalho tinha como principal objetivo validar a vantagem na utilização combinada de minimisers de comprimento variável. Entretanto, a construção do índice proposta, apesar do custo assintótico razoável, implicava, na prática, na construção prévia de um índice completo do texto (vetor de sufixos), e mais outras estruturas auxiliares, que demandam memória excessiva. Essa estratégia limita a aplicação do método a textos de tamanho reduzido e vai de encontro à própria motivação de usar um índice parcial, qual seja, evitar a indexação completa do texto.

A etapa de *seed&extend* descrita por Costa [1] tentava selecionar um subconjunto ótimo de sementes levando em conta a maior quantidade de sementes através de um algoritmo guloso. Esse subconjunto ótimo é a região mais provável de haver uma ocorrência. Após encontrar o melhor subconjunto, ainda é necessário completar o alinhamento dos trechos entre as sementes. Com base nisso, interessa escolher o subconjunto não apenas pelo seu *tamanho*, ou seja, a quantidade de sementes, como também pelo seu *peso*, isto é, a soma dos comprimentos das sementes. Um subconjunto de maior peso implica um menor custo da etapa de 'extensão' da estratégia *seed&extend*.

## 1.2 OBJETIVOS

Este trabalho tem como objetivo geral propor um novo método de indexação de minimisers de tamanhos variados e um novo método de mapeamento baseado em *seed&extend* para casamento aproximado de padrão.

São objetivos específicos deste trabalho:

1. Propor um novo método de construção do índice de minimisers de tamanho variável para dispensar a construção de um índice completo, tornando-a mais eficiente em tempo e memória.

2. Definir o problema de selecionar um subconjunto ótimo de sementes de tamanho variado. Deve-se levar em conta o número e a soma total de seus comprimentos, respeitando a ordem e distâncias relativas entre elas.

3. Propor um algoritmo exato para esse problema, portanto melhorando em relação a procedimentos existentes que são gulosos e baseados em heurísticas.

## 1.3 ESTRUTURA DA MONOGRAFIA

O restante desta monografia está organizado conforme descrito a seguir.

O Capítulo 2 contém o referencial teórico do trabalho. Apresentamos os conceitos fundamentais e problemas tratados neste projeto e contextualizamos bibliograficamente nossa contribuição.

O Capítulo 3 apresenta detalhadamente os métodos propostos, com a formulação teórica dos problemas, os algoritmos desenvolvidos, incluindo a análise teórica de complexidade, além de comentários sobre detalhes de implementação e otimizações heurísticas.

O Capítulo 4 contém uma discussão sobre resultados obtidos em experimentos feitos usando nossa implementação dos algoritmos expostos no Capítulo 3.

O Capítulo 5 contém uma discussão final do trabalho, com uma análise crítica global, enfatizando as principais contribuições, limitações, e apontando direções para desenvolvimentos futuros.

CAPÍTULO 2

# FUNDAMENTOS TEÓRICOS E ESTADO DA ARTE

Neste capítulo, introduzimos conceitos elementares sobre casamento de texto, depois falamos sobre índices completos e parciais. Por fim, tratamos do casamento aproximado indexado via *seed&extend* com índice de minimisers. Durante a descrição desses assuntos, falamos sobre ferramentas e trabalhos já existentes na literatura.

## 2.1 CASAMENTO DE TEXTO

Consideramos o problema de encontrar ocorrências de uma cadeia $P = p_0 \cdots p_{m-1}$ (o *padrão*) num texto $T = t_0 \cdots t_{n-1}$ sobre um alfabeto finito de caracteres $\mathcal{A} = \{a_0, \ldots, a_{\ell-1}\}$. Definimos o *comprimento* de uma cadeia $T = t_0 \cdots t_{n-1}$ como a quantidade de caracteres que a compõem, isto é $|T| = n$. Uma subcadeia contígua de $T$ é denotada por $T[i : j] = t_i \cdots t_{j-1}$.

### 2.1.1 Casamento exato

Quando falamos sobre casamento exato, queremos encontrar as subcadeias contíguas de $T$ que são exatamente iguais ao padrão $P$. Dizemos que há uma ocorrência exata de $P$ em $T$ na posição $i$ quando $T[i : i + m] = P$. Existem diversos algoritmos propostos para resolver esse problema como o algoritmo de *Knuth-Morris-Pratt (KMP)* [5] e o algoritmo de *Boyer-Moore* [6], que possuem custo linear ao tamanho de $T$.

### 2.1.2 Casamento aproximado e distância de edição

Nem sempre queremos encontrar exatamente $P$ em $T$. Dizemos que há uma ocorrência aproximada de $P$ em $T$ com erro $r$ quando existe uma subcadeia $T[i : j]$ tal que $d(P, T[i : j]) \leq r$, em que $d(\cdot, \cdot)$ denota a *distância de edição* [7]. A distância de edição é uma medida de erro entre duas cadeias. Ela representa a menor quantidade de operações (edições) que são necessárias para transformar uma cadeia em outra. Essas operações são (i) adição de um caractere, (ii) remoção de um caractere e (iii) substituição de um caractere por outro.

**Exemplo 2.1.** A distância de edição entre `avalia` e `valham`, $d($`avalia`, `valha`$)$, é igual a 3. As operações para transformar a cadeia `avalia` na cadeia `valham` são:

1. Remover o primeiro caractere, `a`;
2. Substituir o caractere `i` por `h`;
3. Adicionar o caractere `m` ao final.

### 2.1.3 Aplicações de casamento de texto

Casamento de texto é utilizado em diversas aplicações. Algumas delas são corretores ortográficos, filtros de *spam*, sistemas de detecção de intrusão, motores de busca, detecção de plágio, sistemas de recuperação de informação, e biologia computacional [8].

Sequenciamento de DNA é um dos problemas fundamentais da Biologia Computacional. Um sequenciador de DNA é uma máquina capaz de ler uma sequência de bases genéticas. O resultado de uma leitura de um sequenciador de DNA é o que chamamos de *read*. Um molécula de DNA pode conter bilhões de bases (caracteres), sendo que os sequenciadores são capazes de produzir leituras de comprimento variando entre algumas dezenas e poucos milhares de caracteres. Devido a essa limitação tecnológica, o processo de sequenciamento requer a produção de um grande conjunto de leituras parcialmente sobrepostas. Esse conjunto de *reads* precisa ser manipulado para reconstruir o *genoma*. Entretanto, não se sabe a origem de cada *read*, nem suas ordens relativas. Além disso, essas *reads* podem ter erros introduzidos durante o processo de leitura.

Uma das estratégias de montagem do genoma é com o auxílio de uma *referência*, conhecida como *resequenciamento* [9, Cap 4]. Essa referência deve ser uma sequência genética bastante semelhante à que se precisa reconstruir. Ela geralmente é o genoma de uma espécie próxima. No início do processo, cada *read* é mapeada a uma região da referência. Entretanto, como pode haver erros de leitura ou mesmo diferenças naturais entre o genoma sequenciado e a referência, é exatamente aqui que intervém o problema do casamento aproximado de texto. Esse mapeamento gera uma organização do nosso conjunto de *reads*, a partir da qual podemos extrair uma sequência de consenso, a partir da base mais frequente em cada posição. Essa ideia está representada na Figura 2.1.

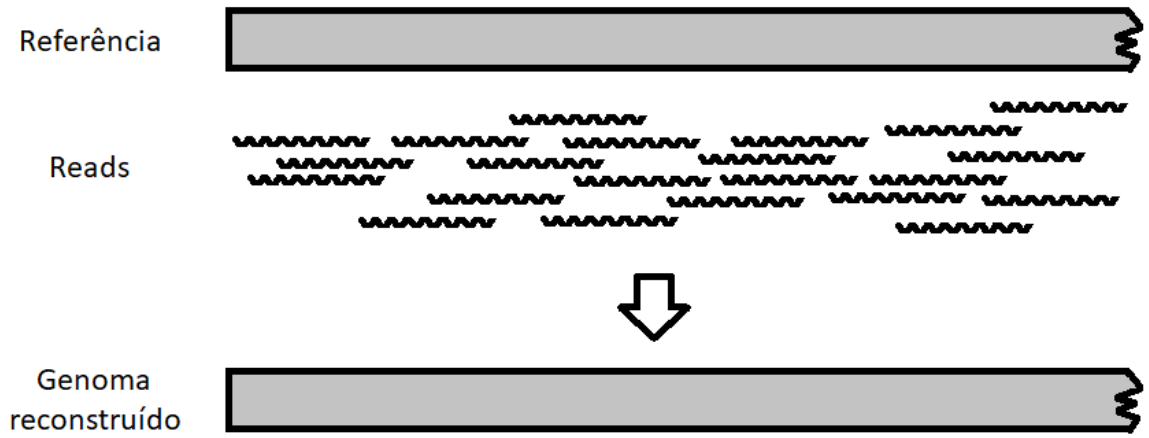


**Figura 2.1** Estratégia de reconstrução de genoma a partir do alinhamento de fragmentos de sequência de DNA a um genoma de referência.

## 2.2 ÍNDICES DE TEXTO

Quando um mesmo texto vai ser usado para encontrar diferentes padrões, existem outros tipos de algoritmos para se fazer o casamento. Eles diferem de algoritmos como o *KMP*, pois o texto é pré-processado de forma a construir um índice $I(T)$. Esse índice pode ser mais custoso de ser construído do que simplesmente fazer o casamento de um padrão. Entretanto, após construído, o casamento indexado é mais eficiente, já que consulta-se apenas $I(T)$, normalmente em tempo proporcional ao tamanho do padrão, não sendo mais necessário percorrer $T$.

### 2.2.1 Índices completos versus índices parciais

Um índice pode ser *completo* ou *parcial*. Um índice completo representa todas as subcadeias de $T$. Exemplos clássicos desses índices são árvore de sufixos [10], e vetor de sufixos [11]. Entretanto, para aplicações com um grande volume de dados, como em Biologia Computacional, a construção desses índices pode levar muito tempo e memória, mesmo com representações comprimidas. Por exemplo, Roberts et al [3] menciona uma aplicação em que o índice de um conjunto de sequências referentes ao genoma da espécie *Rattus norvegicus* requer cerca de 200GB, mesmo utilizando espaço comprimido.

Um outro exemplo de índice mais simples do que os referidos acima, é uma tabela contendo as posição das ocorrências em $T$ de cadeias de comprimento fixo $k$ ($k$-mers). Podemos, inclusive, indexar apenas um subconjunto de $k$-mers de $T$, obtendo dessa forma um índice parcial. Nesse caso, a escolha de quantos e quais $k$-mers indexar impactará diretamente o custo de construção, espaço de armazenagem, e eficácia do índice para aplicações de casamento aproximado.

## 2.3 CASAMENTO APROXIMADO INDEXADO VIA SEED&EXTEND

Se $P = p_0 \cdots p_{m-1}$ ocorre em $T$ com erro $r$, no pior caso em que os erros estão igualmente espaçados em $P$, precisamos ter pelo menos um casamento exato com $T$ de uma subcadeia de $P$ de tamanho pelo menos $(m - r)/(r + 1)$, que é aproximadamente $m/r$ na maioria dos cenários práticos. Portanto, podemos tentar encontrar um tal casamento e usá-lo como *semente (seed)* para alinhar $P$ a alguma região de $T$, e *estender (extend)* pelos flancos para completar o casamento dos caracteres restantes de $P$. Essa estratégia de *seed&extend* está no coração de muitas ferramentas de alinhamento e sequenciamento genético. Sementes podem ser encontradas com a ajuda de um índice construído a partir do texto.

Alguns trabalhos usam essa estratégia para resolver o problema de sequenciamento genético. A ferramenta TAPyR [12] utiliza um índice completo construído a partir do Método de Burrows-Wheeler [13]. Para obter sementes, a ferramenta divide o padrão em várias subcadeias de forma gulosa. A primeira subcadeia é o maior prefixo do padrão que ocorre no texto. Com o restante do padrão, segue-se a mesma estratégia, até que ele esteja totalmente dividido. A ferramenta BWA [14] também usa o mesmo tipo de índice que a ferramenta TAPyR e, para encontrar sementes, aplica algumas heurísticas, que diferem dependendo do tamanho das *reads* e da taxa de erros. A ferramenta MUMmer [15] de

alinhamento de genomas é baseada em árvores de sufixo. Ela foi a primeira ferramenta criada que possibilita comparação de dois genomas inteiros e é muito usada para comparar resultados de ferramentas de montagem de genoma. Os genomas são decompostos nas maiores subcadeias presentes nos dois genomas, usando o índice para cumprir esta tarefa. O survey [16] repertoria várias ferramentas de mapeamento baseadas nessas técnicas e apresenta uma análise experimental comparativa entre elas.

## 2.4 CASAMENTO APROXIMADO INDEXADO COM MINIMISERS

O conceito de *minimisers* foi proposto como uma forma sistemática de selecionar um subconjunto reduzido de $k$-mers a serem representados por um índice parcial [2, 3]. Chamamos o conjunto dos $w$ $k$-mers consecutivos de $T$ a partir de $i$ de $(w,k)$-janela de $T$ na posição $i$, denotada por $T^{(i)}$. Repare que isso corresponde à subcadeia $T[i : i+w+k-1]$. Portanto, usamos o termo 'janela' para nos referirmos indistintamente ao conjunto de $k$-mers ou à correspondente subcadeia do texto. No primeiro caso, dizemos que $w$ é o *tamanho* da janela, denotado por $\#T^{(i)} = w$, enquanto que, no segundo, dizemos que a janela tem 'comprimento' $w+k-1$, denotado por $|T^{(i)}| = w+k-1$.

O *minimiser* de uma $(w,k)$-janela é o mínimo dentre seus $w$ $k$-mers com respeito a uma ordem fixada. Neste trabalho, usaremos a *ordem lexicográfica* sobre $\mathcal{A}^k$ (conjunto dos $k$-mers sobre o alfabeto $\mathcal{A}$ ), o que corresponde a considerar cada $k$-mer como um inteiro não negativo de $k$ dígitos em base-$\ell$.

**Definição 2.1.** O *rank* do $k$-mer $X = x_0 \cdots x_{k-1}$ sobre o alfabeto $\mathcal{A} = \{a_0, \ldots, a_{\ell-1}\}$ é definido por

$$rank(X) = \sum_{i=0}^{k-1} x_i \ell^{k-1-i}, \tag{2.1}$$

com cada caractere identificado à sua posição no alfabeto ($a_j \equiv j$).

**Exemplo 2.2.** Considerando o alfabeto $\mathcal{A}$ ={a,c,g,t } com $\mathtt{a} \equiv 0, \mathtt{c} \equiv 1, \mathtt{g} \equiv 2, \mathtt{t} \equiv 3$, temos que $rank(\mathtt{tagca}) = 30210_{(4)} = 420_{(10)}$.

A ideia básica é indexar os $k$-mers que são minimiser de alguma $(w,k)$-janela de $T$. Esses minimisers são chamados de $(w,k)$-minimisers de $T$. Note que duas janelas consecutivas, i.e., $T^{(i)}$ e $T^{(i+1)}$, possuem todos os $k$-mers em comum, exceto pelo primeiro de $T^{(i)}$ e pelo último de $T^{(i+1)}$. Consequentemente, eles podem ter o mesmo minimiser. Na verdade, é esperado que o minimiser mude apenas uma vez a cada $w/2$ janelas consecutivas [3]. Portanto, $w$ controla a taxa de amostragem e, como resultado, a quantidade de entradas no índice.

Já que uma janela $T^{(i)}$ tem comprimento $|T^{(i)}| = w+k-1$, e os minimisers são sistematicamente definidos, segue que $P$ e $T$ possuem pelo menos um minimiser em comum se alguma subcadeia de $P$ de comprimento $|T^{(i)}|$ ocorre exatamente em $T$. Em outras palavras, se $P$ e $T$ possuem uma sequência em comum de $w$ $k$-mers consecutivos, $P$ e $T$ possuem pelo menos um minimiser em comum.

Escolher quais $k$-mers representar é crucial para a eficiência e eficácia do índice. Queremos que $k$ seja pequeno o suficiente para permitir casamentos exatos de sementes.

Contudo, um grande número de pequenos $k$-mers pode ser necessário para cobrir o texto. Além disso, $k$-mers pequenos podem ter mais ocorrências redundantes, resultando em índices com uma alta densidade de ocorrências por minimiser, em contraste com $k$-mers maiores.

Existe também um compromisso entre sensibilidade e especificidade do índice, dependendo de $k$. Um $k$ menor resulta em uma maior quantidade de casamento de sementes, muitas das quais podem ser falsos positivos, i.e., não faz parte de uma ocorrência de $P$ como um todo. Um $k$ maior, por outro lado, pode tornar o índice extremamente específico, requerendo um casamento exato de um $k$-mer muito longo.

**Exemplo 2.3.** No texto a seguir, observamos que o 2-mer `ag` ocorre 7 vezes, enquanto o 5-mer `acttg` ocorre apenas 1 vez. Para mapear um padrão, se usarmos o 2-mer como semente, teremos várias posições do texto para processar, enquanto que só teremos uma se usarmos o 5-mer.

Ocorrências de `ag`: `gatgatctcattagaaggacttggaagttccagccctagaagagcc`

Ocorrências de `actt`: `gatgatctcattagaaggacttggaagttccagccctagaagagcc`

Li [4] propôs a ferramenta *Minimap* que utiliza índice de minimisers de tamanho único e a estratégia de *seed&extend* para mapeamento de fragmentos de sequências genéticas. Costa [1] demonstrou que, usando minimisers de múltiplos tamanhos, conseguimos melhor aproveitar os benefícios da especificidade e da sensibilidade de cada tamanho diferente de minimiser.

CAPÍTULO 3

# DESENVOLVIMENTO

Neste capítulo, discutimos com detalhe o algoritmo proposto para construção do índice de minimisers, e definimos o problema de selecionar um subconjunto ótimo de sementes e propomos uma solução exata.

## 3.1 CONSTRUÇÃO DO ÍNDICE

Queremos construir um índice $\mathbf{H}$ dos $(w,k)$-minimisers de $T$ para $k \in K = (k_0, \ldots, k_{s-1})$. Vamos supor, por conveniência, que $k_0 < k_1 < \cdots < k_{s-1}$. Esse índice mapeia um $k$-mer $X$ para o conjunto $\mathbf{H}[X]$ de suas ocorrências, que são representadas por suas posições de início em $T$. Supomos, igualmente sem perda de generalidade, que essas posições estão ordenadas.

Um algoritmo força-bruta para computar os minimisers de $T$ desliza uma janela $T^{(i)}$ sobre o texto, i.e. para $i = 0, \ldots, n-(w+k-1)$ e, para cada janela, computa o *rank* de seus $w$ $k$-mers para selecionar o menor entre eles. Esse algoritmo leva tempo $O(nkw)$ se o *rank* de um $k$-mer for computado em tempo $O(k)$.

Obtemos uma melhoria direta ao notarmos que cada $k$-mer só precisa ter seu rank calculado uma única vez, em vez de até $w$ vezes como descrito. Então, mantemos uma fila com os *ranks* dos $k$-mers que estão na janela e, a cada vez que movemos a janela uma posição para a direita, (i) removemos o primeiro elemento, que corresponde ao *rank* do $k$-mer mais à esquerda da janela anterior e (ii) inserimos o *rank* do $k$-mer mais à direita da janela atual.

Esse tempo $O(nk)$ pode ser melhorado ainda mais com a observação de que dois $k$-mers consecutivos de $T$, $A = t_i \cdots t_{i+k-1}$ e $B = t_{i+1} \cdots t_{i+k}$, são idênticos a menos do primeiro e último caracteres. De (2.1), segue que $r_B = rank(B)$ pode ser derivado de $r_A = rank(A)$ em tempo constante como

$$r_B = \rho(r_A, t_{i+k}, k) = \ell \cdot (r_A - t_i \cdot \ell^{k-1}) + t_{i+k}. \tag{3.1}$$

Portanto, o *rank* do último $k$-mer da janela atual pode ser computado a partir do *rank* do último $k$-mer da janela anterior. Essa ideia é ilustrada na Figura 3.1. Na prática, se o tamanho do alfabeto é uma potência de dois, digamos $\ell = 2^b$, a função de *rank* $\rho(r, a, k)$ pode ser eficientemente calculada com operadores bit-a-bit de deslocamento-para-esquerda ( $\ll$ ) e 'e' binário ( `&` ). Segue que o algoritmo leva tempo $O(n)$ e é *online* no texto de entrada, isto é, cada caractere de $T$ precisa apenas ser lido e processado uma única vez.

Esse procedimento pode ser naturalmente estendido e paralelizado para $s > 1$ comprimentos de minimiser $(k_0, \ldots, k_{s-1})$, como mostrado no Algoritmo 1 e explicado em seguida.

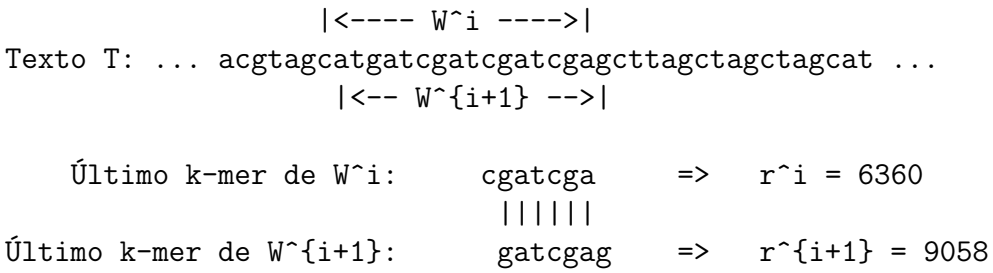


**Figura 3.1** Computação de $r^{i+1} = rank(\texttt{gatcgag})$ a partir de $r^i = rank(\texttt{cgatcga})$ com alfabeto $\mathcal{A} = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$ de acordo com a Equação 3.1. Temos que $r^{i+1} = 4 \cdot (6330 - \texttt{c} \cdot 4^{7-1}) + \texttt{g} = 4 \cdot (6360 - 1 \cdot 4^6) + 2 = 4 \cdot 2264 + 2 = 9058$. Isso corresponde a (i) subtrair a contribuição do 'dígito' mais significativo `c` de `cgatcga`, (ii) deslocar para esquerda uma posição e (iii) adicionar o último 'dígito' menos significativo `g` de `gatcgag`.

O algoritmo *build_index* constrói um índice $\mathbf{H}$ de ocorrências de $(w, k)$-minimisers, para $k \in K = (k_0, \ldots, k_{s-1})$, em que cada $k_j$ corresponde a uma componente $\mathbf{H}_j$ separada. O processo usa uma estrutura de dados chamada *min-queue* para cada $k_j$. Essas estruturas nada mais são do que filas que permitem a consulta pelo elemento mínimo em tempo amortizado constante, conforme explicado na Seção 3.1.2 abaixo. Nesse caso, elas contêm pares $\langle r, i \rangle$, em que $T[i : i + k]$ é um $k$-mer da janela atual e $r = rank(T[i : i + k])$ é o seu *rank*. O 'mínimo' da fila é determinado pelo valor de $r$.

Primeiramente, o algoritmo adiciona o primeiro $k$-mer de cada comprimento à sua fila correspondente. Depois, ele lê um caractere de $T$ por vez, e atualiza cada $\mathbf{H}_j$ para levar em consideração o novo caractere. Ou seja, a função *update* preenche a janela atual até que ela tenha tamanho $w$ e, depois disso, a move uma posição para a direita. Para cada nova janela de cada comprimento de $k$-mer, o *rank* do último $k$-mer é computado em tempo constante com base em equação (3.1), e o minimiser da janela é obtido também em tempo constante com auxílio da *min-queue*. Se este minimiser for diferente do da posição anterior da janela, todas as suas ocorrências são recuperadas e adicionadas ao índice. Caso contrário, nada precisa ser adicionado, exceto se o novo último $k$-mer é mais uma ocorrência desse minimiser. Nesse caso, somente essa ocorrência é adicionada.

Repare que os laços das linhas 4–5, e 7–9 podem ser executados em paralelo (conforme indicado pelo prefixo **par-**), ou seja, as atualizações das janelas para os diferentes valores de $k_j$ podem ser feitas simultaneamente, para cada novo caractere lido do texto de entrada. O trabalho total é $O(sn + q) = O(sn)$, em que $q$ é a quantidade total de ocorrências de minimisers.

### 3.1.1 Um detalhe sobre o critério de rank

Ao usarmos a ordem lexicográfica, todo minimiser de comprimento $a < b$ será um prefixo do minimiser de comprimento $b$. Portanto, caso haja um erro num minimiser de comprimento $b$ do padrão, ele não será encontrado no índice, assim como possivelmente não seria encontrado o minimiser de comprimento $a$, caso o erro tenha ocorrido nas $a$ primeiras posições. É preferível que os minimisers de comprimentos diferentes sejam independentes

**Algoritmo** *build_index*
**Entrada** $T = t_0 \cdots t_{n-1}$: texto indexado
$w$: tamanho da janela
$K = (k_0, \ldots, k_{s-1})$: comprimento dos minimisers
**Saída** $\mathbf{H}$: $(w, K)$-índice de minimisers

```
1  H = (H_0, ..., H_{t-1}) ← índices de (w, k_j)-minimiser vazios
2  Q = (Q_0, ..., Q_{s-1}) ← min-queues vazias
3  ▷ Adiciona o primeiro k-mer de cada tamanho
4  par-para j ∈ 0, ..., s − 1 faça
5      Q_j.push(⟨rank(T[0 : k_j]), 0, T[0 : k_j]⟩)
6  para i = 0, ..., n − 1 faça
7      par-para j ∈ 0, ..., s − 1 faça
8          se i ≥ k_j então
9              update(H_j, Q_j, t_i)
10 devolva H
```

**Função** *update*
**Entrada** $\mathbf{H}_j$: $(w, k_j)$-índice de minimisers
$Q$: fila-mínima
$c \in \mathcal{A}$: último caractere da janela
**Saída** $\mathbf{H}_j$ atualizado

```
1  ⟨r_l, i_l⟩ ← Q.back()                          ▷ último k-mer da janela anterior
2  se Q.length < w então
3      Q.push(⟨ ρ(r_l, c, k_j),  i_l + 1 ⟩)
4  senão
5      ⟨r'_min, i'_min⟩ ← Q.last_min()             ▷ minimiser anterior
6      Q.pop()
7      Q.push(⟨ ρ(r_l, c, k_j),  i_l + 1 ⟩)
8      ⟨r_min, i_min⟩ ← Q.last_min()               ▷ minimiser atual
9      se r_min ≠ r'_min então                     ▷ novo minimiser
10         para cada ⟨r, i⟩ ∈ Q.min() faça
11             H_j[r].add(i)
12     senão se i_min = i_l + 1 então              ▷ mesmo minimiser, nova ocorrência
13         H_j[r_min].add(i_min)
```

**Algoritmo 1** Construção do índice. A função $Q.last\_min()$ retorna o 'último' menor elemento, i.e., o elemento adicionado mais recentemente à fila. A função $Q.min()$ retorna todos os menores elementos da fila.

uns dos outros. Isso pode ser obtido usando-se uma ordem diferente para cada valor de $k$.

### 3.1.2 Min-queue

Damos agora uma breve descrição do tipo abstrato de dado (TAD) *min-queue*, que permite inserções e remoções com política de primeiro a entrar, primeiro a sair (*first-in-first-out*, FIFO), além de consultas de mínimo em tempo constante amortizado. Ou seja, a *min-queue* $Q$ possui operações $Q.push(v)$, que insere o elemento $v$ ao final da fila, $Q.pop()$, que remove o elemento da frente da fila, e $Q.min()$ que retorna o(s) elemento(s) mínimo(s) presentes na fila, sem removê-lo(s).

Começamos observando que, para dois elementos $u$ e $v$, em que $u$ foi inserido antes de $v$, se $u > v$, então $u$ nunca será o mínimo nessa fila, já que $v$ só é removido após $u$.

**Exemplo 3.1.** Considere a inserção dos elementos $7, 3, 6, 9, 5, 8$ nessa ordem. Após a inserção do 3, o 7 já não mais poderá ser o mínimo. Da mesma forma, após a inserção do 5, o 6 e o 9 já não poderão mais ser mínimos. Assim, essa fila poderia ser representada por

$$\not{7}\ \ 3\ \ \not{6}\ \ \not{9}\ \ 5\ \ 8,$$

sendo os elementos barrados aqueles que nunca serão dados como resposta a $min()$. Repare que os elementos não-barrados formam uma subsequência crescente que sempre inclui o último elemento.

Uma vez que não nos interessa conhecer o valor dos elementos desenfileirados, nem qualquer outro que não possa ser mínimo, podemos modificar a fila para manter apenas os elementos necessários para responder a essas consultas de $min()$, omitindo os demais. Para tanto, a fila deve obedecer a invariante de estar em ordem crescente, de acordo com o ilustrado no Exemplo 3.1. Com isso, o elemento mínimo será o primeiro elemento efetivamente representado na fila. Se houver $q > 1$ elementos mínimos, podemos recuperar todos em tempo $O(q)$, já que eles formarão um 'prefixo' de tamanho $q$ da fila.

Ao enfileirarmos um novo elemento $v$, devemos barrar/omitir todos os elementos maiores que ele para mantermos a invariante. Esses elementos maiores são um 'sufixo' (possivelmente vazio) da fila, já que ela está em ordem crescente. Portanto, omitimos os elementos do fim para o início, até encontrarmos um elemento menor ou igual a $v$, ou até que a fila esteja vazia. Por exemplo, se inseríssemos o valor 4, os elementos barrados seriam o 5 e o 8, pelo que teríamos

$$\not{7}\ \ 3\ \ \not{6}\ \ \not{9}\ \ \not{5}\ \ \not{8}\ \ 4.$$

Como cada elemento é barrado no máximo uma vez, o custo amortizado é $O(1)$ para inserção. Mais ainda, como não nos interessa saber os valores dos elementos barrados, podemos armazenar apenas as quantidades dos elementos omissos antes de cada elemento efetivamente armazenado. Assim, no exemplo corrente, teríamos

$$[1]\ \ 3\ \ [4]\ \ 4,$$

sendo os números entre colchetes as quantidades de elementos barrados nessas posições.

Desenfileirar o primeiro elemento da fila através de um *pop*() não quebra a invariante da ordem crescente. Entretanto, já que alguns elementos foram omitidos da representação, não necessariamente o primeiro elemento efetivamente armazenado está no início da fila. No exemplo corrente, o elemento 3 armazenado não é primeiro elemento da fila, e sim o 7, que entretanto foi omitido. Assim, o *pop*() deve levar em conta a quantidade de elementos omitidos no início da fila. Caso essa quantidade seja maior que zero, o *pop*() apenas decrementa esse número. No exemplo corrente, uma chamada a *pop*() resultaria

$$[0]\ \ 3\ \ [4]\ \ 4.$$

Se, entretanto, não houver elementos barrados no início da fila, então o primeiro elemento é removido. No exemplo atual, uma segunda chamada a *pop*() resultaria

$$[4]\ \ 4.$$

Em qualquer dos casos, a operação de desenfileirar requer tempo constante.

## 3.2 MAPEAMENTO SEED & EXTEND

Uma vez construído o índice, passamos a descrever nossa proposta de como utilizá-lo para mapear sequências de maneira aproximada através da heurística *seed&extend*. A ideia básica dessa estratégia é a seguinte. Vamos procurar ocorrências dos minimisers do padrão $P$ (sementes) no texto $T$ com auxílio do índice $\mathbf{H}$. Um subconjunto de ocorrências dessas sementes em $T$ que respeite a ordem e as distâncias relativas das ocorrências dos minimisers em $P$ é chamada de uma *âncora*. Essa âncora pode ser usada como base e estendida num alinhamento local de $P$ em $T$. Propomos um algoritmo para identificar a 'melhor' âncora, ou seja, aquela que cobre a maior porção do padrão. Essa ideia é ilustrada esquematicamente na Figura 3.2 e descrita em detalhes a seguir.

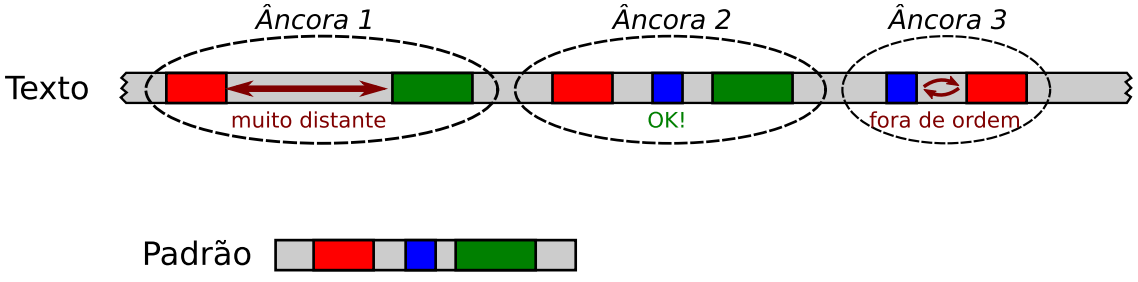


**Figura 3.2** Estrátegia de escolha da âncora ótima. Nesse caso, encontramos ocorrências de três sementes no texto. Uma combinação de ocorrências forma uma âncora. Nessa ilustração destacamos três dessas combinações sendo que a segunda é a melhor delas pois as três sementes estão na ordem correta e em distâncias compatíveis com as do padrão.

No que se segue, sejam $\mathcal{M}_{T,K}$ e $\mathcal{M}_{P,K}$ os conjuntos de $(w, K)$-minimisers de $T$ e $P$, respectivamente, para janelas de tamanho fixo $w$ e minimisers de comprimento em $K = (k_0, \ldots, k_{s-1})$.

**Definição 3.1.** O $(w, K)$-*layout* de $P$ é definido como a sequência de todas as ocorrências dos seus $(w, K)$-minimisers, ordenadas pelas suas posições iniciais, i.e.

$$\hat{P}_K = (\langle j_0, Y_0\rangle, \ldots, \langle j_{u-1}, Y_{u-1}\rangle), \tag{3.2}$$

em que $j_0 < \cdots < j_{u-1}$, $0 \leq j_q \leq m - |Y_q|$, e $P[j_q : j_q + |Y_q|] = Y_q \in \mathcal{M}_{P,K}$ para $q = 0, \ldots, u-1$.

O *layout* sintetiza a composição do padrão em termos dos seus minimisers. Analogamente, precisamos resumir a composição do texto em termos desses mesmos minimisers.

**Definição 3.2.** A $(w, K)$-*projeção* de $P$ em $T$ é definida como a sequência de *todas* as ocorrências em $T$ dos minimisers de $P$, ordenados por suas posições iniciais, i.e.

$$\hat{T}_K = (\langle i_0, X_0\rangle, \ldots, \langle i_{v-1}, X_{v-1}\rangle), \tag{3.3}$$

em que $i_0 < \cdots < i_{v-1}$, $0 \leq i_q \leq n - |X_q|$, e $T[i_q : i_q + |X_q|] = X_q \in \mathcal{M}_{P,K} \cap \mathcal{M}_{T,K}$, para $q = 0, \ldots, v-1$.

Na Figura 3.2, o layout e a projeção correspondem às sequências de blocos coloridos, do padrão e do texto, respectivamente.

Uma $(w, K)$-*âncora* (de $P$ em $T$) é um subconjunto da $(w, K)$-projeção correspondente. Nosso objetivo é localizar a 'melhor' âncora, com base na qual esperamos obter o alinhamento local de $P$ em $T$ com menor erro. Para propósitos de apresentação, definiremos esse problema e apresentaremos a solução correspondente da forma mais simples até a forma mais completa.

### 3.2.1 Ordem apenas

Na primeira versão, definimos como ótima a âncora composta pela maior quantidade possível de ocorrências de minimisers na mesma ordem relativa tanto em $P$ quanto em $T$, i.e., a maior subsequência não-contígua

$$\hat{F} = (\langle i_{q_0}, X_{q_0}\rangle, \ldots, \langle i_{q_{v'-1}}, X_{q_{v'-1}}\rangle), \tag{3.4}$$

em que $0 \leq q_0 < \cdots < q_{v'-1} < v$ e os minimisers $X_{q_0}, \ldots, X_{q_{v'-1}}$ ocorrem em $P$ e $T$ nesta ordem.

Suponhamos inicialmente que cada minimiser aparece apenas uma vez no layout $\hat{P}_K$. Como os elementos de qualquer subsequência da projeção $\hat{T}_K$ estão ordenados pela posição em $T$, resta-nos garantir que os minimisers dessa subsequência também estejam ordenados pela posição em $P$.

Seja a transformação $\varphi$ que associa cada elemento da projeção $\langle i_q, X_q\rangle$ à posição no padrão $P$ do minimiser $X_q$, isto é, $\varphi_q \stackrel{\text{def}}{=} \varphi(\langle i_q, X_q\rangle)$ é tal que $P[\varphi_q : \varphi_q + |X_q|] = X_q$. Considere então a representação da projeção pela sua transformada $\varphi = (\varphi_0, \ldots, \varphi_{v-1})$. Uma subsequência qualquer de $\varphi$ continua a corresponder a um subconjunto de ocorrências de minimisers ordenados pela posição em $T$. Se, além disso, os valores dessa subsequência

(que correspondem a posições em $P$) estão em ordem crescente, então os respectivos minimisers ocorrem na mesma ordem relativa em $T$ e em $P$. Segue daí que o problema de encontrar a âncora ótima reduz-se ao problema de encontrar a maior subsequência crescente (*longest increasing subsequence*—*l.i.s.*) de $\varphi$.

Se considerarmos múltiplas ocorrências de um minimiser em $P$, então mudamos levemente a formulação atual. Nesse caso, $\varphi(\langle i_q, X_q\rangle)$ associa cada minimiser da projeção a uma lista ordenada de posições em $P$. A tranformada $\varphi$ será então a concatenação dessas listas.

**3.2.1.1 Maior subsequência crescente** Passamos então a descrever uma solução para esse problema de l.i.s.. Dada a sequência de entrada $\varphi = (\varphi_0, \ldots, \varphi_{v-1})$, defina um vetor $L$ de mesmo comprimento tal que $L[q]$ é o tamanho da l.i.s. de $\varphi$ que termina com o elemento $\varphi_q$, para $0 \leq q < v$. Os elementos de $L$ são recursivamente definidos por

$$L[q] = \begin{cases} 1, & \text{se } q = 0 \\ 1 + \max_i\{L[i] \mid i < q \text{ e } \varphi_i < \varphi_q\}, & \text{se } q > 0, \end{cases} \tag{3·5}$$

e a solução global para o problema é dada pelo elemento máximo desse vetor.

Para calcular (3·5) eficientemente, podemos utilizar uma *árvore de segmentos* (*segment tree*—ST). Uma ST é uma árvore binária balanceada com $n$ folhas, numeradas de $0, \ldots, n-1$, e com suporte a duas operações: (i) $rmq(l, r)$, que retorna a *maior* folha no intervalo $[l, r)$, e (ii) $update(i, val)$, que atualiza o valor da folha $i$ para $val$. Uma árvore de segmentos pode ser construída a partir de um vetor contendo os valores das folhas em tempo e espaço $O(n)$. Além disso, cada operação requer tempo $O(\log n)$.

O vetor $L$ pode ser representado por uma ST, sendo a posição $q$ correspondente à folha $q$. Iniciamos as folhas da ST com valor 0, e processamos os elementos de $\varphi$ por ordem crescente de valor. Ao processarmos o elemento $\varphi_q$, atualizamos a folha respectiva $L[q]$ com o valor definido pela expressão (3·5). O valor do max do segundo caso da expressão pode ser obtido através de uma consulta $L.rmq(0, q)$. Com efeito, o valor de $L[q]$ só depende de valores de $L[i]$ com $i < q$ e $\varphi_i < \varphi_q$, e todos os elementos com valor inferior a $\varphi_q$ já terão sido processados, em particular aqueles que se encontram à esquerda da posição $q$. A construção de $L$ requer portanto ordenar $\varphi$ em tempo $O(v \log v)$ e, em seguida, processar cada elemento com uma consulta *rmq* e um *update* na ST, ou seja, em tempo total $O(v \log v + v \cdot (2 \log v)) = O(v \log v)$.

Para recuperarmos a l.i.s., recuperamos o maior elemento de $L$, digamos $L[q]$. Sabemos então que $\varphi_q$ faz parte da l.i.s.. Depois, encontramos um elemento $L[i]$ tal que $i < q$, $\varphi_i < \varphi_q$ e $L[i] = L[q] - 1$. Este elemento também faz parte da l.i.s.. Repetimos este processo até que a l.i.s. seja totalmente reconstruída. O custo desta parte é $O(v)$, pois fazemos uma varredura em $L$ para encontrar o maior elemento e depois fazemos uma varredura de trás para frente em $L$ para reconstruir a l.i.s..

### 3.2.2 Ordem e comprimento

Devido ao tamanho variável dos minimisers, pode ser que os minimisers da l.i.s. 'cubram' menos caracteres do que uma âncora com menos minimisers, porém mais longos no total.

Nesta segunda versão, em vez de nos importarmos com a quantidade de ocorrências, nos importamos com o comprimento de cada ocorrência. Ou seja, em vez de $\hat{F}$ ser a subsequência crescente mais longa, a âncora ótima é definida como a subsequência crescente mais 'pesada' (*heaviest increasing subsequence*—*h.i.s.*), em que o peso de um minimiser semente é dado pelo seu comprimento. Mais precisamente, queremos resolver o problema combinatório de maximizar a soma dos comprimentos $|X_{q_0}| + \ldots + |X_{q_{v'-1}}|$ sujeito a que $X_{q_0}, \ldots, X_{q_{v'-1}}$ estejam ordenados por posição tanto em $P$ quanto em $T$.

Nosso algoritmo sofre uma pequena modificação para satisfazer a esse novo objetivo. Como agora também nos importamos com o comprimento/peso do minimiser, modificamos a transformação $\varphi$ para associar ao elemento $\langle i_q, X_q\rangle$ o par $\varphi_q \stackrel{\text{def}}{=} \langle \varphi_q.pos,\ \varphi_q.len\rangle$, em que $\varphi_q.pos$ é a posição de $X_q$ em $P$, como no caso anterior, e $\varphi_q.len = |X_q|$.

A solução do problema h.i.s. é similar à solução do problema l.i.s. anterior, sendo que agora definimos um vetor $H$, em que $H[q]$ é o peso da h.i.s. de $\varphi$ que termina com o elemento $\varphi_q$, para $0 \leq q < v$, e que é dado por

$$H[q] = \begin{cases} \varphi_0.len, & \text{se } q = 0 \\ \varphi_q.len + \max_i\{H[i] \mid i < q \text{ e } \varphi_i.pos < \varphi_q.pos\}, & \text{se } q > 0. \end{cases} \tag{3.6}$$

O algoritmo de cálculo de $H$ e reconstrução da h.i.s. são, portanto, essencialmente os mesmos da l.i.s.. Para calcular $H$, cada elemento $\varphi_q$ é processado por ordem crescente de $\varphi_q.pos$ com $H.update(q, \varphi_q.len + H.rmq(0, q))$. Na reconstrução da h.i.s., partimos da posição $q$ do elemento máximo de $H$ e andamos para trás procurando por uma posição $i < q$ tal que $\varphi_i.pos < \varphi_q.pos$ e $H[i] = H[q] - \varphi_q.len$. É interessante notar que a l.i.s. é apenas um caso particular da h.i.s. em que o peso de cada ocorrência é igual a 1.

Para efeito de visualização, considere a Figura 3.3. No eixo horizontal, temos o índice $q$ do elemento no vetor de entrada $\varphi$. No eixo vertical, temos os valores $\varphi_q.pos$ associado a cada ponto. Cada ponto tem também uma massa associada (o comprimento do minimiser), que não é representada na figura. Os elementos são processados de baixo para cima (pela ordem de posição no padrão). Ilustrado na figura está o momento em que todos os pontos pretos já foram processados e o ponto vermelho está sendo processado. Nesse instante, consideramos como estender da melhor forma uma sequência crescente formada por pontos abaixo e à esquerda do ponto vermelho com este ponto. A h.i.s. que termina nesse ponto está indicada pela região sombreada. Repare que os dois pontos abaixo e à direita do ponto vermelho não puderam ser incluídos pois estão fora de ordem em relação a ele, ou seja, correspondem a minimisers que aparecem depois no texto, mas antes no padrão. A solução final poderá eventualmente incluir, além dos pontos na região sombreada, o ponto mais alto da figura, que corresponde ao último minimiser no padrão e que, portanto, será o último a ser processado.

### 3.2.3 Ordem, comprimento e distância

Na terceira e última versão, levaremos em consideração também a distância entre duas sementes consecutivas da âncora $\hat{F}$. A distância entre dois minimisers em $T$ não precisa ser igual, mas não deve ser muito diferente da distância das ocorrências correspondentes em $P$, isso porque pode haver erros de inserção e/ou remoção entre elas.

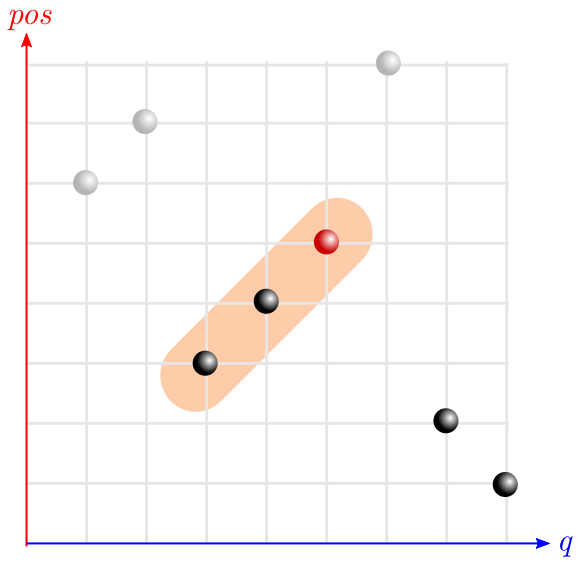


**Figura 3.3** Representação gráfica do problema h.i.s. com restrição de ordem.

Considere dois minimisers $A$ e $B$, que ocorrem no padrão $P$ nas posições $j_A$ e $j_B$, respectivamente. Suponha que esses minimisers ocorrem no texto $T$ nas posições $i_A$ e $i_B$, respectivamente. A distância relativa entre as posições de $A$ e $B$ no padrão será dada por $(j_B - j_A)$, e analogamente, a distância no texto será $(i_B - i_A)$. Queremos impor uma restrição para que as distâncias entre os minimisers no texto e no padrão não sejam muito diferentes. Concretamente, exigimos que

$$|(i_B - i_A) - (j_B - j_A)| = |(i_B - j_B) - (i_A - j_A)| < \epsilon,$$

para uma constante $\epsilon \geq 0$. Definindo $\delta_A = (i_A - j_A)$ para um minimiser $A$, a desigualdade anterior pode ser reescrita como

$$|\delta_B - \delta_A| < \epsilon. \tag{3.7}$$

Nesta última formulação, exigimos adicionalmente que as componentes consecutivas da âncora ótima $\hat{F} = (\langle i_{q_0}, X_{q_0}\rangle, \ldots, \langle i_{q_{v'-1}}, X_{q_{v'-1}}\rangle)$ respeitem a restrição (3.7), ou seja, desejamos resolver o problema combinatório de maximizar a soma dos comprimentos $|X_{q_0}| + \ldots + |X_{q_{v'-1}}|$ sujeito a que $X_{q_0}, \ldots, X_{q_{v'-1}}$ estejam ordenados por posição tanto em $P$ quanto em $T$ e que, $|\delta_{q_{i+1}} - \delta_{q_i}| \leq \epsilon$ para $0 \leq i < v'$, sendo $\delta_{q_i} \equiv \delta_{X_{q_i}}$.

De maneira análoga às soluções anteriores, vamos definir uma transformada da projeção, de forma a obter a entrada para o algoritmo que vamos descrever. Neste caso, a função $\varphi$ associa a cada componente $\langle i_q, X_q\rangle$ uma tripla $\varphi_q = \langle \varphi_q.pos,\ \varphi_q.len,\ \varphi_q.delta\rangle$, onde $\varphi.pos$ e $\varphi.len$ representam, como anteriormente, a posição no padrão e o comprimento de $X_q$, respectivamente, e $\varphi_q.delta = \delta_{X_q} = (i_q - \varphi_q.pos)$.

Para resolver esta versão da h.i.s., também definimos um vetor $H$, análogo ao da Seção 3.2.2, em que que $H[q]$ representa o peso da h.i.s. que termina com o elemento $\varphi_q$,

para $0 \leq q < v$, e que agora é dado por

$$H[q] = \begin{cases} \varphi_0.len, & \text{se } q = 0 \\ \varphi_q.len + \max_i\{H[i] \mid i < q \text{ e} \\ \qquad\qquad \varphi_i.pos < \varphi_q.pos \text{ e} \\ \qquad\qquad |\varphi_q.delta - \varphi_i.delta| < \epsilon\}, & \text{se } q > 0. \end{cases} \tag{3.8}$$

Relembremos que, na representação da solução da Figura 3.3, ao processarmos um ponto $q$ (vermelho), consideramos as subsequências já processadas que terminam em pontos à esquerda ($i < q$) e abaixo ($\varphi_i.pos < \varphi_q.pos$) desse ponto. Na presente formulação, essas restrições não são suficientes, uma vez que impomos também a restrição de distância. Precisamos portanto de mais uma dimensão para representá-la. A Figura 3.4 ilustra graficamente esta situação. No gráfico (a) temos a entrada representada em três dimensões: no eixo-$x$, temos o índice $q$ do elemento $\varphi_q$, no eixo-$y$ temos o valor de $\varphi_q.delta$, e no eixo-$z$ temos o valor de $\varphi_q.pos$. Cada ponto possui uma massa ($\varphi_q.len$, não-representada na figura). Assim como no caso anterior, os pontos são processados de baixo para cima. Ao processarmos o ponto $q$ vermelho, consideramos os pontos abaixo e à esquerda no plano $xz$, como no caso anterior. No gráfico (b), temos a projeção do gráfico (a) nos eixos-$xz$. Ocorre que, diferentemente do caso da Figura 3.3 o ponto vermelho não pode estender a sequência que termina no ponto negro $q-1$ (logo abaixo e à esquerda), pois este ponto, apesar de respeitar a ordem relativa, não respeita a restrição de distância. Isso pode ser visualizado no gráfico (c) que ilustra a projeção no plano $xy$. Nesse plano, só podem ser considerados os pontos à esquerda e dentro de uma banda de largura $2\epsilon$ em torno do ponto $q$, que comporta a restrição $|\delta_q - \delta_i| < \epsilon$.

Esse tipo de consulta sobre pontos em espaços multidimensionais é suportada por estruturas como *range tree* [17] ou *K-D tree* [18, 19]. Entretanto, apresentaremos outra solução baseada em *Dividir para Conquistar* e árvore de segmentos para resolver o problema em uma complexidade de espaço melhor que range trees e complexidade de tempo melhor que K-D trees. Esta solução está exposta no Algoritmo 2 que detalhamos a seguir.

O Algoritmo *solve_his* recebe a entrada $\varphi = [\varphi_d = (\varphi_d.pos, \varphi_d.len, \varphi_d.delta)]_{d=0\ldots v-1}$ e apenas faz a primeira chamada à função recursiva *solve* sobre todo o intervalo de pontos $[0, v)$. A função *solve* recebe, além do vetor de entrada, um intervalo de pontos $[l, r)$ e resolve o problema para os pontos nesse intervalo, ou seja, calcula o valor de $H[q]$ para $q = l, \ldots, r-1$. Isso é feito dividindo-se o intervalo em duas metades $[l, r) = [l, m) \cup [m, r)$ a serem resolvidas separadamente de forma recursiva. O detalhe aqui é que $H[q]$ representa o peso máximo de um subsequência que termina com o elemento da posição $q$, porém essa sequência pode ter elementos em qualquer posição $0 \leq i < q$. Portanto, não podemos resolver o problema para a segunda metade dos pontos $[m, r)$ considerando-os de forma isolada dos pontos da primeira metade. Por essa razão, uma chamada à função *link* é realizada entre as duas chamadas a *solve*, com o propósito de 'conectar' os resultados da primeira metade aos pontos da segunda metade. Mais especificamente, essa função calcula, para cada ponto na segunda metade $q \in [m, r)$, a h.i.s. terminada no elemento $q$ do tipo $L \to q$, onde $L$ é uma das h.i.s. terminadas num ponto da primeira metade, que já terão sido encontradas pela chamada anterior a *solve*. Essa sequência pode superar e

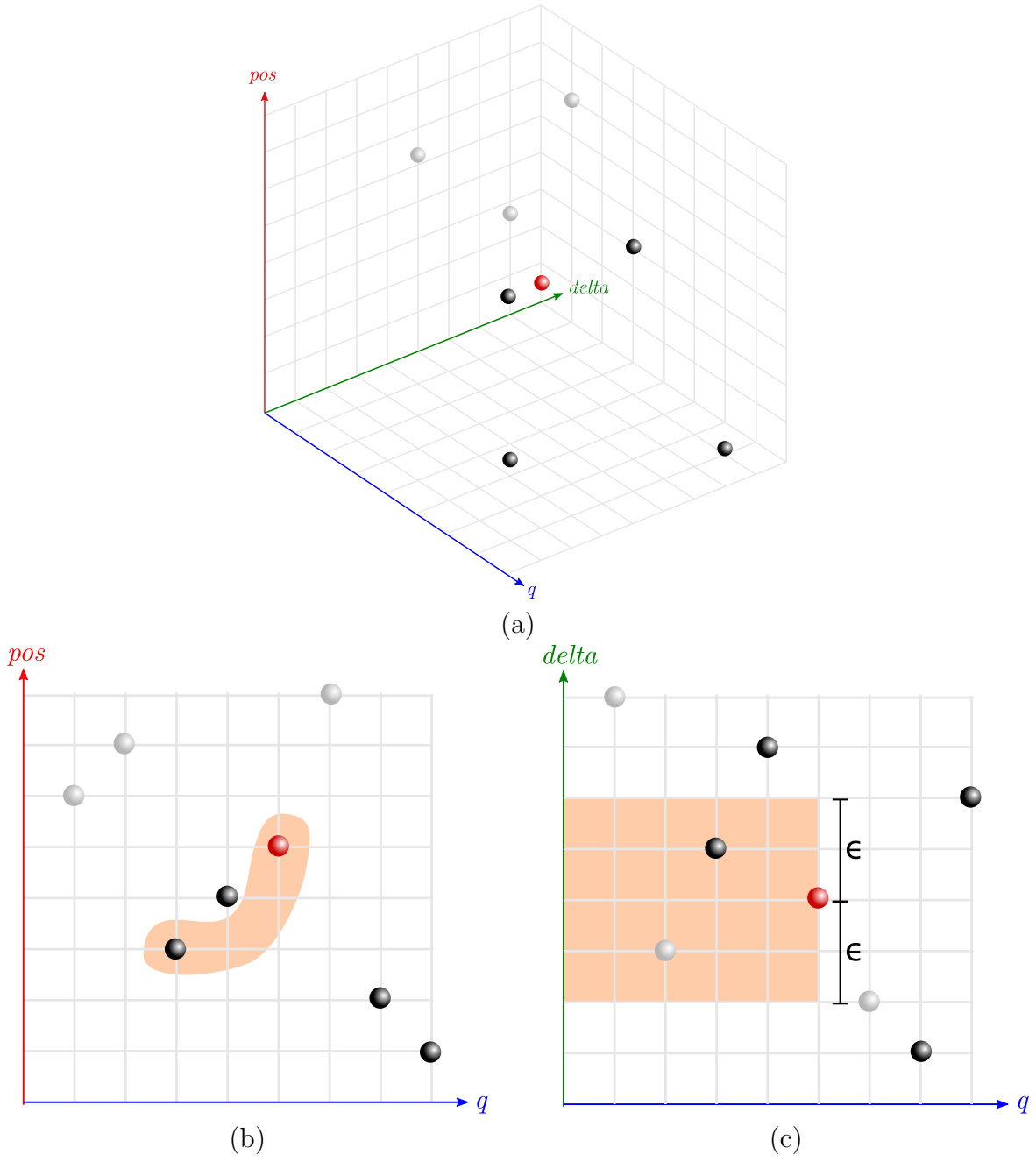


**Figura 3.4** Representação gráfica do problema h.i.s. com restrição de ordem e distância.

**Algoritmo** *solve_his*
**Entrada** $\Phi = (\phi_0, \ldots, \phi_{v-1})$ tal que $\phi_d = (\phi_d.pos,\ \phi_d.len,\ \phi_d.delta)$
$\epsilon \geq 0$
**Saída** $H = (h_0, \ldots, h_{v-1})$ tal que
$H[q]$ =peso da h.i.s. terminada em $q$ com restrições de ordem e distância
$\epsilon \geq 0$

```
1 H ← (φ_0.len, ··· , φ_{v−1}.len)        ▷ caso-base: contém apenas o próprio elemento
2 devolva solve(Φ, 0, v, H)
```

**Função** *solve*
**Entrada** $\Phi = (\phi_0, \ldots, \phi_{v-1})$
$l, r$: intervalo de $\Phi$ considerado
$H = (h_0, \ldots, h_{v-1})$ parcialmente calculado
$\epsilon \geq 0$
**Saída** $H$ atualizado t.q. $H[q]$ =peso da h.i.s. terminada em $q$, para $q = l, \ldots, r-1$

```
1 se r − l ≤ 1 então
2     devolva H
3 m ← ⌊(l + r)/2⌋
4 H ← solve(Φ, l, m, H)                    ▷ computa H no intervalo [l, m)
5 H ← link(Φ, l, r, H)          ▷ conecta resultados em [l, m) com pontos em [m, r)
6 ← solve(Φ, m, r, H)                      ▷ computa H no intervalo [m, r)
7 devolva H
```

**Função** *link*
**Entrada** $\Phi = (\phi_0, \ldots, \phi_{v-1})$
$l, r$: intervalo de $\Phi$ considerado
$H = (h_0, \ldots, h_{v-1})$ parcialmente calculado
**Saída** $H$ atualizado

```
1 m ← ⌊(l + r)/2⌋
2 ST ← nova árvore de segmentos vazia
3 para cada ⟨φ_q.pos, φ_q.len, φ_q.delta⟩ ∈ Φ em ordem crescente de φ.pos faça
4     se q < m então
5         se ST[δ_q] < H[q] então
6             ST.update(φ_q.delta, H[q])
7     senão
8         H[q] ←max(H[q], φ_q.len + ST.rmq(φ_q.delta − ε + 1, φ_q.delta + ε))
9 devolva H
```

**Algoritmo 2** Solução do problema h.i.s. com restrições de ordem e distância.

substituir uma outra subsequência terminada em $q$ calculada anteriormente.

Mesmo antes de descrever os detalhes da função *link*, podemos argumentar indutivamente que o algoritmo está correto. Com efeito, o algoritmo é trivialmente correto para sequências de tamanho um. Para $v > 1$, na primeira chamada a *solve*, supomos por hipótese de indução que as soluções para a primeira metade dos pontos $q \in [0, v/2)$ estão corretas. A função *link* calculará, para cada $q$ na segunda metade $[v/2, v)$, a h.i.s. terminada em $q$ e com *precursor* (elemento anterior a $q$ na sequência) em $[0, v/2)$. Essa ainda não é necessariamente a solução do problema pois pode haver uma outra subsequência melhor terminada em $q$ e com precursor também na segunda metade. Porém, essa opção será eventualmente considerada durante a chamada subsequente a *solve*. É fundamental perceber que o único lugar em que os pontos são efetivamente conectados para formar subsequências é na função *link*. Serão eventualmente consideradas todas as maneiras de ligar $q$ a um precursor $i = 0, \ldots, q-1$, nessa ordem, sendo que, quando tal opção for considerada, o valor final de $H[i]$ já será conhecido. Segue daí a correção do algoritmo.

Consideremos, por fim, a função *link*, que é responsável por conectar da melhor forma as h.i.s. (já conhecidas) que terminam em pontos na primeira metade $[l, m)$ do intervalo $[l, r)$ com os elementos da segunda metade $[m, r)$. Ao considerarmos um ponto $q$ na segunda metade, só podemos conectá-lo a um precursor $i < q$ que obedeça $\varphi_i.pos < \varphi_q.pos$ e $|\varphi_q.delta - \varphi_i.delta| < \epsilon$. A primeira restrição pode ser imposta processando-se os elementos em ordem crescente de $\varphi_i.pos$. A segunda restrição, que pode ser expressa como

$$\varphi_q.delta - \epsilon < \varphi_i.delta < \varphi_q.delta + \epsilon, \tag{3.9}$$

implica consultar o valor máximo de $H[i]$ para $i$ com valores de $\varphi_i.delta$ num certo intervalo, ou seja, um tipo de rmq. Para tal, usaremos uma árvore de segmentos para os pontos $\varphi_i$, porém agora com as folhas indexadas por $\varphi_i.delta$. Concretamente, processamos cada elemento em $[l, r)$ na mesma ordem que nas versões anteriores do problema, ou seja, na ordem dada por $\varphi_i.pos$ com desempate por maior $i$. Considere que processaremos o elemento $q$. Caso $q \in [l, m)$, a folha correspondente da ST é atualizado para o valor já calculado de $H[q]$, caso este supere o valor corrente dessa folha (linha 6). Se $q \in [m, r)$, apenas atualizamos o valor correspondente $H[q]$ (linha 8), recorrendo a uma consulta rmq na ST para obter o melhor precursor que obedece às restrições de ordem e distância. Note que todas as restrições são, de fato, obedecidas:

i) $i < q$, pois os únicos elementos mantidos na árvore de segmentos são aqueles que pertencem ao intervalo $[l, m)$;
ii) $\varphi_i.pos < \varphi_q.pos$, devido à ordem que os elementos são processados;
iii) A restrição de distância (3.9) corresponde ao intervalo de consulta na árvore de segmentos.

O Algoritmo 2 possui custo de tempo $T(v) = 2 \cdot T(v/2) + O(v \log v) = O(v \log^2 v)$, em que $O(v \log v)$ é o custo de *link*. Além disso, nosso algoritmo usa $O(v)$ de memória, já que a árvore de segmentos possui $O(v)$ folhas. Como as folhas da árvore de segmentos são numeradas de 0 até $n$, precisamos de um passo adicional para comprimir os valores de $\varphi_q.delta$ em coordenadas nesse intervalo. Entretanto isso não muda o custo de *link*, já que isso pode ser feito em $O(v \log v)$, ordenando os valores de $\varphi_q.delta$ e substituindo

**Algoritmo** *optimal_anchor*
**Entrada** $\Phi = (\phi_0, \ldots, \phi_{v-1})$
$H = (h_0, \ldots, h_{v-1})$ calculado
$\epsilon \geq 0$
**Saída** $\hat{F}$: âncora ótima com restrições de ordem e distância

```
1  (weight, i) ← max_element(H)
2  F̂ ←[φ_i]
3  remaining_weight ←weight − φ_i.len
4  enquanto remaining_weight > 0 faça
5      i ← i − 1
6      φ_l ← F̂.front()
7      se H[i] = remaining_weight e φ_i.pos < φ_l.pos e |φ_l.delta − φ_i.delta| < ε então
8          F̂.push_front(φ_i)
9          remaining_weight ←remaining_weight − φ_i.len
10 devolva F̂
```

**Algoritmo 3** Obtenção da âncora ótima a partir de $H$.

seus valores pelos seus índices ordenados.

A h.i.s. pode ser reconstruída de modo similar à versão anterior do problema, mas adicionando a restrição $|\delta_q - \delta_i| < \epsilon$ ao procurar um elemento, como mostrado no Algoritmo 3, que tem custo $O(v)$.

## 3.3 ESCOLHA DAS SEEDS

Na seção anterior, consideramos todas as ocorrências dos $(w, K)$-minimisers em $P$ como *seeds*. Entretanto, propomos uma nova forma de escolher essas sementes com o objetivo de criar um melhor balanço entre a especificidade e a sensibilidade dos diferentes tamanhos de minimiser. Para cada $k$-minimiser da janela $P^{(i)}$, para algum $k \in K$, consideramos apenas as ocorrências do maior minimiser que também ocorre em $T$. Essa estratégia toma proveito da especificidade dos minimisers de tamanhos maiores. Entretanto, quando eles não ocorrem em $T$, recorremos à sensibilidade dos minimisers de tamanhos menores.

## 3.4 IMPLEMENTAÇÃO

A implementação desses algoritmos foram feitas em C++ e podem ser obtidas no repositório

https://github.com/dorDiogo/tg-dor.

A seguir vamos discutir alguns detalhes sobre a implementação dos algoritmos. Apesar dos processos aqui discutidos não estarem restritos a aplicações em Biologia computacional, essa é a principal aplicação em vista, pelo que os algoritmos foram implementados sobre o alfabeto $\mathcal{A} = \{a_0, a_1, a_2, a_3\} = \{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$.

Em relação ao índice, alguns aspectos devem ser destacados. Primeiro, ele foi implementado usando uma *hash table* fornecida pela biblioteca open source Abseil [20], conhecida como *Swiss table* [21], pois é mais eficiente que a hash table da biblioteca padrão de C++, `std::unordered_map`. Segundo, a construção do índice é feita de forma paralelizada, como descrito na Seção 3.1, através da interface OpenMP [22]. Por fim, sobre o critério de rank para diferentes valores de k citado na Seção 3.1.1, na nossa implementação, usamos ainda a ordem lexicográfica (2.1), porém permutando a ordem das letras no alfabeto conforme o valor de $k$.

Outro detalhe de implementação está relacionado ao Algoritmo 2. Em vez de executar o algoritmo em toda a entrada, a entrada foi dividida em vários trechos e cada trecho foi processado individualmente. Cada divisão é feita quando duas ocorrências consecutivas estão a uma distância no texto maior que o tamanho do padrão. Nesse caso, é impossível que haja uma combinação entre *seeds* de trechos diferentes que faça sentido com as restrições. Além disso, trechos a serem processados que sejam menores do que o tamanho da h.i.s. dos trechos já processados são ignorados.

CAPÍTULO 4

# EXPERIMENTOS

## 4.1 DADOS DE TESTE

De forma a avaliar os métodos desenvolvidos, extraímos do corpus Pizza&Chilli[23] nossa sequência genética ($T$) usada como referência para mapeamento de *reads*. Para geração das *reads*, copiamos trechos da referência e introduzimos erros aleatoriamente. Um erro pode ser de três tipos: inserção de um caractere, remoção de um caractere e substituição de um caractere.

A vantagem principal de produzirmos *reads* a partir da referência é a facilidade para o cálculo de métricas do mapeamento. Como sabemos exatamente de que trecho a *read* foi produzida, podemos compará-lo com o trecho para o qual ela foi mapeada pelo algoritmo. A desvantagem é que não estamos usando *reads* verdadeiras, produzidas por máquinas de leitura de DNA, para avaliar nosso método.

## 4.2 MÉTRICAS DO ÍNDICE

Nesta seção, descrevemos as métricas utilizadas para análise de estatísticas do índice.

**Densidade de ocorrência por minimiser**

Esta métrica é a média de ocorrências de cada minimiser em posições diferentes de $T$. É calculada da seguinte forma:

$$\frac{\sum_{X \in \mathcal{M}_{T,k}} |\mathbf{H}[X]|}{|\mathcal{M}_{T,k}|} \tag{4.1}$$

**Cobertura do índice**

Esta métrica corresponde à proporção da quantidade de caracteres que estão contidos em alguma ocorrência de algum minimiser em relação à quantidade total de caracteres da referência. Formalmente, é definida como

$$Cobertura = \frac{\sum_{i=0}^{|T|} \Delta(\mathbf{H}, i)}{|T|} \tag{4.2}$$

em que

$$\Delta(\mathbf{H}, i) = \begin{cases} 1, & \text{se } \exists (X, k, pos), pos \in \mathbf{H}_k[X] \text{ e } i \in [pos, pos + k) \\ 0, & \text{caso contrário.} \end{cases}$$

Essa métrica não é simplesmente a soma do tamanho de cada ocorrência, pois podem existir sobreposições entre elas e um caractere teria uma contribuição possivelmente maior que 1 para a cobertura.

## 4.3 MÉTRICAS DO MAPEAMENTO

Nesta seção, descrevemos as métricas utilizadas para análise de estatísticas do mapeamento de *reads* à referência.

**Acurácia**

A acurácia é a porcentagem de *reads* corretamente mapeadas para o trecho do qual foram extraídas originalmente.

Uma *read* é considerada corretamente mapeada quando a quantidade de caracteres mapeados fora do trecho de sua origem não ultrapassa 10% da quantidade de caracteres mapeados dentro do trecho. Damos esta pequena tolerância para casos que inserções são introduzidas no ínicio ou no final da *read*. Pode ser que, por coincidência, os caracteres inseridos sejam iguais aos caracteres encontrados na fronteira externa ao trecho.

**Cobertura do mapeamento**

Esta métrica mede a proporção da quantidade de caracteres que estão contidos em alguma ocorrência de algum minimiser da h.i.s. em relação à quantidade total de caracteres do padrão. Note que esta métrica não é a proporção entre o peso da h.i.s. e a quantidade total de caracteres do padrão, pois pode haver sobreposições entre as ocorrências da h.i.s..

## 4.4 AVALIAÇÃO

Os experimentos foram realizados em um computador no sistema operacional Windows 10, com um processador Intel® Core™ i5-8400 @2.80GHz, e com RAM de 16GB 2400MHz.

### 4.4.1 Avaliação do índice

Analisamos como as métricas se comportam de acordo com os parâmetros $w$ e $K$ de indexação. Primeiramente, analisamos o índice com $k$-mers de tamanho único, e, depois, com múltiplos tamanhos.

Na Tabela 4.1, fixamos o valor de $w$. Notamos que, quanto mais $k$ se aproxima de $w$, maior é a cobertura. Ou seja, o espaço entre uma ocorrência e outra diminui. Esse resultado vai de acordo com o que é mostrado em Roberts et al [3]. Além disso, observamos que, ao aumentar $k$, a quantidade de minimisers distintos aumenta exponencialmente, enquanto a densidade diminui exponencialmente. Isso é consequência de que existem $\ell^k$ possíveis $k$-mers. Para $k = 4$, a quantidade de possíveis minimisers é muito pequena ($4^4 = 256$), quando comparada a $k = 14$ ($4^{14} \approx 10^8$). Isso afeta diretamente a densidade de ocorrência por minimiser. Com poucos possíveis $k$-mers, a densidade é muito alta, mas cai exponencialmente conforme $k$ aumenta. No entanto, para $k \geq 20$, o crescimento da quantidade de minimisers diminui bastante. Nesse ponto, a densidade se aproxima de 1, ou seja, a quantidade de minimisers depende só da quantidade de amostras feitas em $T$.

| $k$ | Quantidade de minimisers | Densidade de ocorrência | Cobertura (%) |
|---|---|---|---|
| 4 | 231 | 115.213 | 21 |
| 6 | 2.744 | 9.079 | 29 |
| 8 | 25.203 | 980 | 37 |
| 10 | 229.545 | 107 | 44 |
| 14 | 8.855.274 | 2,78 | 58 |
| 20 | 20.686.698 | 1,19 | 75 |
| 26 | 21.930.904 | 1,12 | 87 |
| 30 | 22.458.983 | 1,09 | 93 |

**Tabela 4.1** Métricas do índice para referência de 400MB, $w = 40$ e $K = \{k\}$.

Na Tabela 4.2, fixamos $K$. Em relação à cobertura, observamos a mesma coisa acontecer que na Tabela 4.1. Conforme $w$ se aproxima de $k$, a cobertura aumenta, até que chega em 100% quando $w = k$. A densidade de ocorrência praticamente não se altera, enquanto que a quantidade de minimisers diminui proporcionalmente ao aumento de $w$. Isso é reflexo direto do impacto de $w$ na taxa de amostragem, já que a taxa de amostragem é inversamente proporcional a $w$.

| $w$ | Quantidade de minimisers | Densidade de ocorrência | Cobertura (%) |
|---|---|---|---|
| 15 | 33.130.002 | 1,806 | 100 |
| 20 | 25.668.864 | 1,798 | 93 |
| 25 | 21.054.825 | 1,795 | 83 |
| 30 | 17.855.397 | 1,795 | 75 |
| 35 | 15.510.875 | 1,798 | 67 |
| 40 | 13.681.992 | 1,798 | 61 |
| 45 | 12.251.417 | 1,810 | 56 |
| 50 | 11.093.901 | 1,821 | 52 |

**Tabela 4.2** Métricas do índice para referência de 400MB e $K = \{12\}$.

Por fim, analisamos, o comportamento dessas métricas quando usamos múltiplos tamanhos de $k$-mers. Como os minimisers e suas ocorrências são obtidos de forma independente, então, ao combiná-los, a quantidade de minimisers é simplesmente a soma dos valores individuais de cada $k$. Entretanto, a cobertura não é obtida de forma independente, pois uma ocorrência pode ter sobreposição com outra. Como explicado na Seção 3.1.1, ao usar uma ordem diferente para cada valor de $k$, os minimisers de comprimentos diferentes são independentes uns dos outros. Isso diminui bastante a sobreposição entre eles, como está evidenciado pelo aumento de cobertura de 75% para 96% e de 64% para 79% nos exemplos da Tabela 4.3.

Para testar o tempo de construção do índice, variamos vários parâmetros que o afetam. No primeiro experimento, variamos o tamanho de $T$. Os resultados podem ser vistos na Figura 4.1. É notável o comportamento linear no tempo de construção no índice, assim

| $K$ | Quantidade de minimisers | Densidade de ocorrência | Cobertura (%) |
|---|---|---|---|
| {10} | 229.545 | 107 | 44 |
| {15} | 13.681.992 | 1,8 | 61 |
| {20} | 20.686.698 | 1,2 | 75 |
| {10, 15, 20} | 34.598.235 | 2,1 | 96 |
| {8} | 25.203 | 980 | 37 |
| {16} | 17.084.357 | 1,4 | 64 |
| {8, 16} | 17.109.560 | 2,9 | 79 |

**Tabela 4.3** Métricas do índice com tamanhos variados de $k$-mers para referência de 400MB e $w = 40$.

como esperado, já que seu custo é $O(n)$ para $|K| = 1$. Além disso, vimos que $T$ de 400MB foi indexado em $27,2s$.

No segundo, variamos o tamanho dos $k$-mers indexados. Os resultados podem ser vistos na Figura 4.2. Notamos que, inicialmente, o tempo de construção aumenta, mas estabiliza. Podemos atribuir este aumento à *hash table* **H**. Como a densidade de ocorrência por minimiser diminui para próximo de 1 rapidamente como vimos na Tabela 4.1, a quantidade de entradas em **H** aumenta rapidamente. O maior tamanho de **H**, e o acesso a muitas entradas distintas tem um impacto direto na cache, aumentando o tempo de execução do algoritmo.

No terceiro experimento, variamos o tamanho da janela $w$. Os resultados podem ser vistos na Figura 4.3. Notamos que acontece o oposto de quando aumentamos o tamanho de $k$. Neste caso, como a janela é maior, diminuimos a amostragem de minimisers, ou seja, temos menos ocorrências de minimisers ao todo. Isso diminui a quantidade de acessos e o tamanho de **H**.

Após analisar esses três experimentos, notamos que apesar do algoritmo ter uma complexidade de tempo $O(n)$, a sua eficiência ainda está relacionada a $w$ e $k$, devido a fatores de desempenho da *hash table*. Além disso, nosso algoritmo é capaz de indexar uma referência de 400MB em 27,2s.

### 4.4.2 Avaliação do mapeamento

Avaliamos as métricas do mapeamento de 1000 *reads*, cada uma gerada a partir de um trecho aleatório de tamanho 1000 da referência. Os erros introduzidos nas *reads* foram distribuídos igualmente entre inserções, remoções e substituições. Definimos o valor de $\epsilon$ da restrição exposta na Equação 3.7 como $\epsilon = 5$. Comparamos índices contendo minimisers de tamanho único com índices contendo minimisers de tamanhos variados.

A acurácia e a cobertura do mapeamento de cada padrão pelo Algoritmo 2, além do tamanho de sua entrada, podem ser vistos na Tabela 4.4. Notamos que, ao combinar os minimisers de tamanho 20, 25 e 30, a acurácia aumenta em relação ao resultado individual de todos eles, mesmo apesar de $|\hat{T}_{\{20,25,30\}}| \approx 0.3 \cdot |\hat{T}_{\{20\}}|$ nas duas taxas de erros. Além de a acurácia aumentar, a cobertura do mapeamento também aumenta em

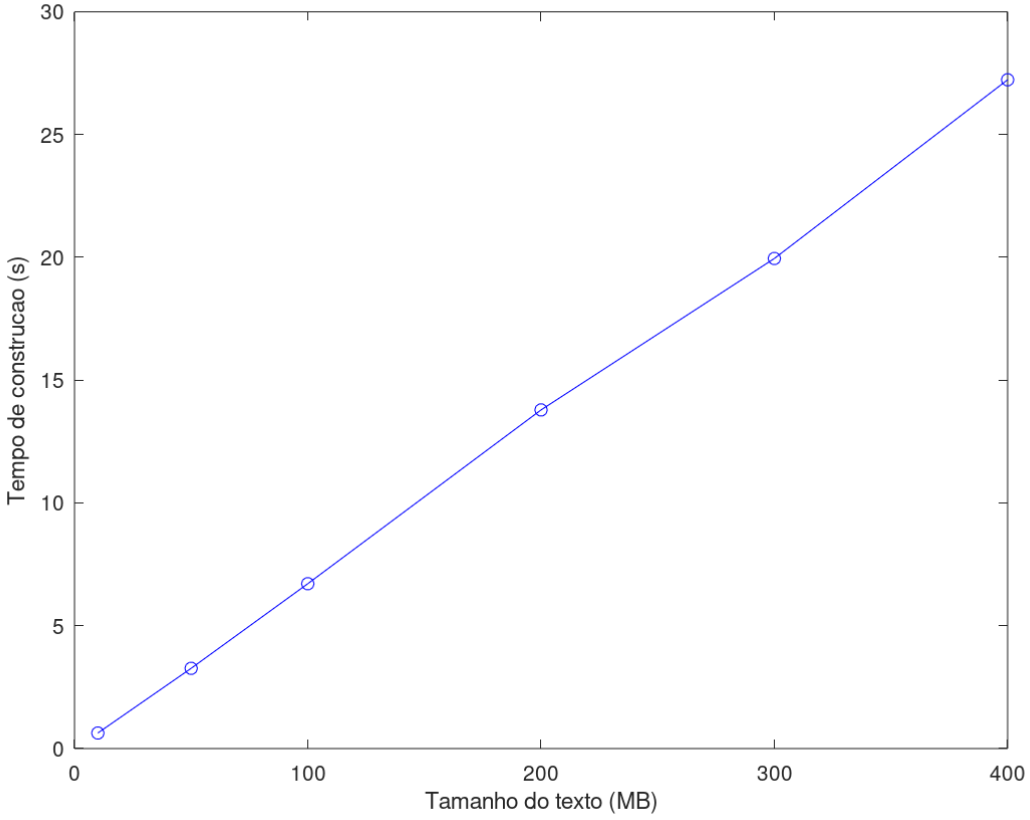


**Figura 4.1** Tempo de construção do índice ao variar o tamanho da referência. Texto indexado com $w = 20$ e $K = \{15, 20, 30\}$.

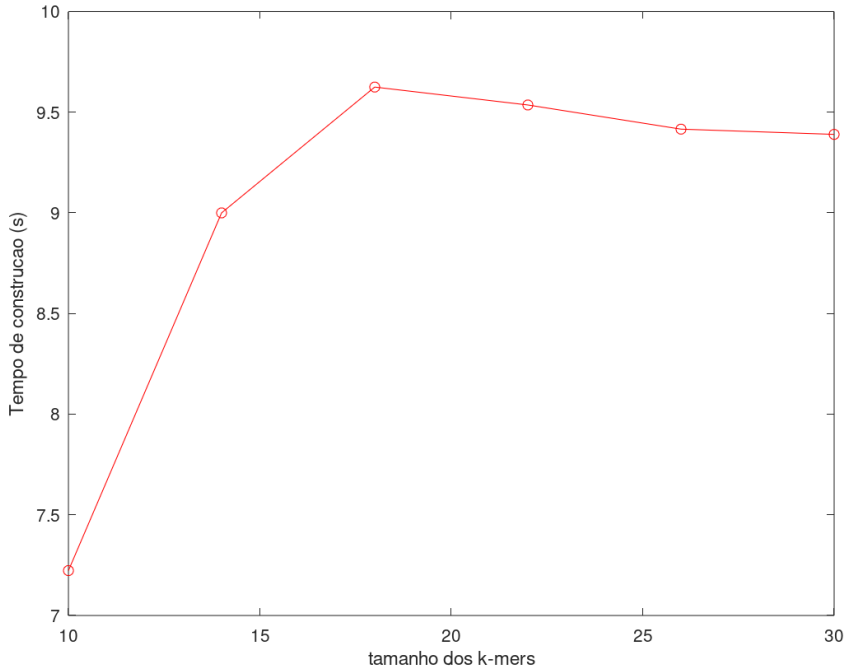


**Figura 4.2** Tempo de construção do índice ao variar o valor de $k$. Referência indexada de 400MB com $w = 100$.

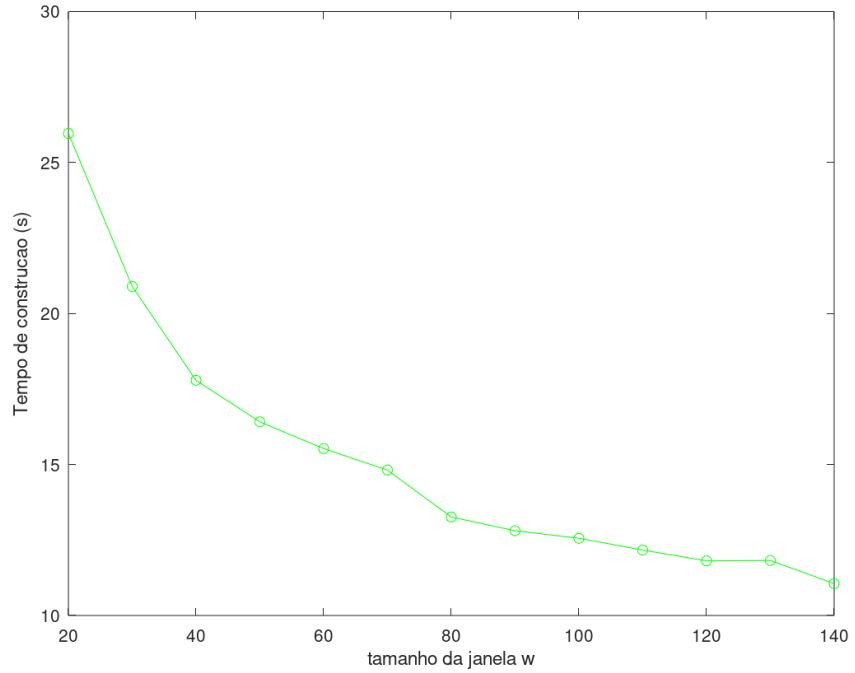


**Figura 4.3** Tempo de construção do índice ao variar o valor de $w$. Referência indexada de 400MB com $K = \{15, 20\}$.

aproximadamente 20%. No segundo cenário, em que temos um erro a cada 6,7 caracteres em média, o nosso método mostra uma acurácia de 85%. Note que precisa haver pelo menos um casamento de minimisers de tamanho pelo menos 20 entre $P$ e $T$ para que haja a possibilidade de um mapeamento correto. Já no primeiro cenário, em que temos um erro a cada 20 caracteres, a acurácia é próxima de 100%.

A ferramenta *Minimap*, que utiliza uma heurística para o mapeamento, foi executada com o índice mais sensível ($K = \{20\}$). Ela apresenta uma acurácia de 99% para uma taxa de erro de 5%, similar à nossa implementação. Entretanto, para uma taxa de erro de 15%, a acurácia cai para 52%, inferior à nossa ferramenta. Esses são resultados preliminares e mais experimentos são necessários para se quantificar a diferença entre as duas ferramentas, tanto em questão de acurácia, quanto em questões de desempenho, como memória e tempo de execução.

| | Taxa de erro em $P$ | | | | | |
|---|---|---|---|---|---|---|
| | 5% | | | 15% | | |
| $K$ | Acur. (%) | Cober. (%) | $\lvert\hat{T}\rvert$ | Acur. (%) | Cober. (%) | $\lvert\hat{T}\rvert$ |
| {20} | 98,8 | 38,2 | 14197 | 78,2 | 5,0 | 2611 |
| {25} | 98,9 | 29,5 | 2519 | 59,1 | 3,7 | 219 |
| {30} | 98,9 | 23,3 | 574 | 32,0 | 3,5 | 13 |
| {20, 25, 30} | 99,0 | 46,1 | 4189 | 85,0 | 5,9 | 883 |

**Tabela 4.4** Métricas do mapeamento de *reads* com diferentes taxas de erro. Índice da refêrencia de 400MB construído com $w = \min_{k \in K} k$. $|\hat{T}|$ é o tamanho médio da $(w, K)$-projeção de $P$ em $T$. A cobertura média diz respeito às reads corretamente mapeadas.

Por fim, vimos que usar um índice que combina vários minimisers, aumenta a acurácia e diminui bastante a quantidade de sementes, i.e., o tamanho de $\hat{T}$, em relação ao índice mais sensível (com menor $k$). Além disso, nosso método de mapeamento possui uma boa acurácia, mesmo em cenários com muito ruído em comparação aos tamanhos dos minimisers usados.

CAPÍTULO 5

# CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, propusemos um método para resolver uma variação do problema de casamento aproximado de texto baseado num índice de minimisers de comprimento variável e na heurística *seed&extend*. As principais características/contribuições do trabalho são

1. A utilização de um índice de minimisers de comprimento variável de forma a combinar a especificidade de sementes maiores, sem a consequente perda de sensibilidade, que é garantida pelas sementes menores, potencializada pela adoção de ordens de $k$-mers distintas para cada valor de $k$.

2. Um algoritmo *online* eficiente para a construção desse índice.

3. A definição formal do problema combinatório de encontrar subconjunto ótimo de sementes, ou seja, de maior comprimento total, respeitando a ordem relativa entre as suas ocorrências no texto e no padrão, assim como as distâncias entre elas.

4. Um algoritmo exato polinomial para o problema do item anterior, e a sua utilização para o casamento aproximado via *seed&extend* acrescido de algumas otimizações.

Os resultados experimentais aqui reportados revelam que o algoritmo de construção do índice apresenta um resultado prático que confirma a previsão teórica de ser linear no comprimento do texto indexado. A técnica de paralelização, de fato, reduz o tempo de indexação por um fator constante (dados não reportados). A utilização de minimisers de comprimento variável, aliada à escolha das maiores sementes possíveis em cada janela, durante o mapeamento, diminui a quantidade de ocorrências das sementes no texto em relação ao índice mais sensível (com menor $k$), tornando o mapeamento mais rápido e acurado, revelando a eficácia do método proposto.

## 5.1 MELHORIAS FUTURAS

Uma fonte de melhoria mais facilmente realizável no curto prazo diz respeito à paralelização de trechos do código. Por exemplo, conforme discutido na Seção 3.4, durante o mapeamento, a projeção (conjunto de ocorrências das sementes no texto) é particionada em seções que são tratadas independentemente. Esse processamento poderia, então, ser facilmente paralelizado. Mais ainda, embora tenhamos discutido algoritmos para mapeamento de um padrão no texto, na prática as aplicações envolvem o casamento de um extenso conjunto de padrões que, em princípio, podem ser mapeados em paralelo.

Outro aspecto menos imediato deve-se ao fato de que o problema descrito na Seção 3.2.3 introduz uma restrição para manter compatíveis as distância entre sementes no padrão e no texto. Da forma como foi definida, $\epsilon$ é uma constante que não

depende da separação dessas sementes. Na verdade, se a taxa de erros de inserções for diferente taxa de erros de remoções, então, talvez estejamos sendo muito restritivos com duas ocorrências muito separadas, ou muito lenientes com duas ocorrências próximas.

**Exemplo 5.1.** Suponha que $|P| = 1000$ e que a taxa de inserções é duas vezes a taxa de remoções. Suponha que existem três ocorrências de minimisers em $P$, nas posições 0, 100, e 900, e que também ocorrem em $T$.

Se dissermos que $\epsilon = 20$, então ocorrências dos dois primeiros minimisers nas posições 500 e 615 do texto seriam consideradas compatíveis pois $|(615 - 100) - (500 - 0)| = 15 < 20$. Ou seja, estamos considerando razoável 15 inserções a mais do que remoções para dois minimisers a uma distância 100. Mantendo a mesma proporção, deveria ser razoável aceitar $15 \cdot 9 = 135$ inserções a mais que remoções caso estejamos considerando os minimisers nas posições 0 e 900. Ainda assim, só aceitamos no máximo 20.

Note que os experimentos realizados neste trabalho, foram feitos usando uma taxa de erros de inserção igual à taxa de erros de remoção, e por isso, não é esperado que as distâncias relativas entre duas sementes varie muito, independente da separação entre elas no texto/padrão. Uma melhoria futura consiste em flexibilizar a taxa de erro $\epsilon$ conforme a separação entre as componentes da âncora.

# REFERÊNCIAS BIBLIOGRÁFICAS


[1] Costa, Arthur Latache Pimentel Gesteira: *Casamento aproximado de padrões baseado em índices de minimizers de comprimento variável*. Trabalho de Conclusão de Curso, Centro de Informática, Universidade Federal de Pernambuco, julho 2019. (resumo), 1.1, 2.4

[2] Schleimer, Saul, Daniel S. Wilkerson e Alex Aiken: *Winnowing: Local Algorithms for Document Fingerprinting*. Em *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, página 76–85, New York, NY, USA, 2003. Association for Computing Machinery, ISBN 158113634X. https://doi.org/10.1145/872757.872770. 1.1, 2.4

[3] Roberts, Michael, Wayne Hayes, Brian R. Hunt, Stephen M. Mount e James A. Yorke: *Reducing storage requirements for biological sequence comparison*. Bioinformatics, 20(18):3363–3369, julho 2004, ISSN 1367-4803. https://doi.org/10.1093/bioinformatics/bth408. 1.1, 2.2.1, 2.4, 2.4, 4.4.1

[4] Li, Heng: *Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences*. Bioinformatics, 32(14):2103–2110, março 2016, ISSN 1367-4803. https://doi.org/10.1093/bioinformatics/btw152. 1.1, 2.4

[5] Knuth, Donald E, James H Morris, Jr e Vaughan R Pratt: *Fast pattern matching in strings*. SIAM journal on computing, 6(2):323–350, 1977. https://doi.org/10.1137/0206024. 2.1.1

[6] Boyer, Robert S. e J. Strother Moore: *A Fast String Searching Algorithm*. Commun. ACM, 20(10):762–772, outubro 1977, ISSN 0001-0782. https://doi.org/10.1145/359842.359859. 2.1.1

[7] Levenshtein, Vladimir I: *Binary codes capable of correcting deletions, insertions, and reversals*. Soviet physics doklady, 10(8):707–710, fevereiro 1966. 2.1.2

[8] Soni, Kapil Kumar, Rohit Vyas e Amit Sinhal: *Importance of string matching in real world problems*. International Journal Of Engineering And Computer Science, 3(6):6371–6375, 2014. 2.1.3

[9] Lesk, Arthur M: *Introduction to genomics*. Oxford University Press, 2017. 2.1.3

[10] Weiner, Peter: *Linear pattern matching algorithms*. Em *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, páginas 1–11. IEEE, 1973. https://doi.org/10.1109/SWAT.1973.13. 2.2.1

[11] Manber, Udi e Gene Myers: *Suffix Arrays: A New Method for On-Line String Searches*. SIAM Journal on Computing, 22(5):935–948, 1993. https://doi.org/10.1137/0222058. 2.2.1

[12] Fernandes, Francisco, Paulo da Fonseca, Luis Russo, Arlindo Oliveira e Ana Freitas: *TAPYR: An efficient high-throughput sequence aligner for re-sequencing applications*. EMBnet.journal, 17(B):43, março 2012, ISSN 2226-6089. https://doi.org/10.14806/ej.17.B.284. 2.3

[13] Burrows, M. e D. J. Wheeler: *A block-sorting lossless data compression algorithm*. Relatório Técnico 124, DEC, Digital Systems Research Center, Palo Alto, California, 1994. 2.3

[14] Li, Heng e Richard Durbin: *Fast and accurate short read alignment with Burrows–Wheeler transform*. Bioinformatics, 25(14):1754–1760, maio 2009, ISSN 1367-4803. https://doi.org/10.1093/bioinformatics/btp324. 2.3

[15] Delcher, Arthur L., Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White e Steven L. Salzberg: *Alignment of whole genomes*. Nucleic Acids Research, 27(11):2369–2376, janeiro 1999, ISSN 0305-1048. https://doi.org/10.1093/nar/27.11.2369. 2.3

[16] Schbath, Sophie, Véronique Martin, Matthias Zytnicki, Julien Fayolle, Valentin Loux e Jean François Gibrat: *Mapping Reads on a Genomic Sequence: An Algorithmic Overview and a Practical Comparative Analysis*. Journal of Computational Biology, 19(6):796–813, junho 2012. https://doi.org/10.1089/cmb.2012.0022, PMID: 22506536. 2.3

[17] Bentley, Jon Louis: *Decomposable searching problems*. Information Processing Letters, 8(5):244 – 251, 1979, ISSN 0020-0190. https://doi.org/10.1016/0020-0190(79)90117-0. 3.2.3

[18] Lueker, George S.: *A Data Structure for Orthogonal Range Queries*. Em *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, SFCS '78, página 28–34, USA, 1978. IEEE Computer Society. https://doi.org/10.1109/SFCS.1978.1. 3.2.3

[19] Bentley, Jon Louis: *Multidimensional Binary Search Trees Used for Associative Searching*. Commun. ACM, 18(9):509–517, setembro 1975, ISSN 0001-0782. https://doi.org/10.1145/361002.361007. 3.2.3

[20] *Abseil library*. URL: https://abseil.io. Consultado em 12/09/2019. 3.4

[21] Benzaquen, Sam, Alkis Evlogimenos, Matt Kulukundis e Roman Perepelitsa: *Swiss Tables*. URL: https://abseil.io/blog/20180927-swisstables. Consultado em 12/09/2019. 3.4

[22] *OpenMP*. URL: https://www.openmp.org. Consultado em 05/10/2019. 3.4

[23] Ferragina, Paolo e Gonzalo Navarro: *Pizza&Chilli Corpus*. URL: http://pizzachili.dcc.uchile.cl. Consultado em 15/10/2019. 4.1