Universidade Federal de Pernambuco
Centro de Informática

Graduação em Engenharia da Computação

Construção online de wavelet tree no formato da árvore de Huffman

Ícaro Julião Monteiro Guerra

Trabalho de Graduação

Recife
25 de setembro de 2023

Universidade Federal de Pernambuco
Centro de Informática

Ícaro Julião Monteiro Guerra

Construção online de wavelet tree no formato da árvore de Huffman

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
25 de setembro de 2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Monteiro Guerra, Ícaro Julião.
Construção online de wavelet tree no formato da árvore de Huffman / Ícaro Julião Monteiro Guerra. - Recife, 2023.
30 p. : il., tab.

Orientador(a): Paulo Gustavo Soares da Fonseca
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2023.

1. Wavelet tree. 2. Código de Huffman. 3. Algoritmo dinâmico. I. Soares da Fonseca, Paulo Gustavo. (Orientação). II. Título.

000 CDD (22.ed.)

Agradecimentos

Agradeço primeiramente aos meus pais, por se dedicarem durante todos esses anos para que pudesse usufruir de uma educação de qualidade, que me proporcionou ingressar na Universidade e concluir a graduação com êxito.

Agradeço à Aline, minha parceira de vida, que me acompanhou nos altos e baixos desse trabalho, sempre me motivando a acreditar no meu sucesso.

Agradeço ao meu orientador, Paulo Fonseca, por se mostra sempre solícito para dar as coordenadas e ajudar a concluir este último desafio.

E por último, quero agradecer a mim mesmo, por nunca ter desistido apesar das adversidades, pois todo o esforço está sendo colhido agora.

Resumo

A *wavelet tree* é uma estrutura de dados bastante utilizada para representar longas cadeias de caracteres de forma otimizada e permitir consultas de *rank* e *select*. Alguns estudos já exploraram como modificar o formato dessa árvore para melhorar a complexidade de consultas e de espaço de armazenamento e mostram que o formato da árvore de Huffman é ótimo no quesito memória. Apesar de ser uma estrutura conhecida, poucos algoritmos de construções *online* foram propostos e este trabalho tem como objetivo propor um algoritmo *online* para a construção da *wavelet tree* no formato da árvore de Huffman, que tem como principal vantagem o fato de que o texto de referência não precisa ser armazenado em memória principal, o que pode ser um requisito desejável, ou mesmo necessário, em aplicações de *streams* de dados. O método proposto consiste na adaptação de uma algoritmo para construção online do Código de Huffman, representado por uma árvore estritamente binária com topologia variável. A mudança na forma da árvore implica em modificações nos bitvectors representados pelos nós, com impacto significativo no desempenho do algoritmo. O método foi implementado e testado em textos de diferentes características, e os resultados sugerem que pode representar uma alternativa viável, especialmente em memória, porém previsivelmente menos eficiente em tempo com respeito à construção offline tradicional.

Palavras-chave: Wavelet tree, árvore de Huffman, dinâmico, online

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

Introdução

A representação de textos de forma eficiente é um dos problemas mais clássicos na computação. Um dos algoritmos mais famosos e utilizados para representar dados de maneira eficiente é a codificação de Huffman, uma técnica proposta por Huffman [9] para determinar o código de prefixo de custo mínimo de um texto. Na construção desse código, é feita a construção de uma estrutura auxiliar chamada árvore de Huffman que é utilizada para descobrir o código de cada caractere do texto.

Nos anos 70, Faller [4] e Gallager [6] desenvolveram independentemente um algoritmo de compressão dinâmico baseado no método de Huffman. No livro de Crochemore e Rytter [1], temos a descrição do algoritmo, que gera o código de um texto sem conhecimento prévio do seu alfabeto e que pode ter um uso de memória menor que o algoritmo proposto inicialmente se levarmos em conta que não é necessário armazenar o texto para processamento.

A *wavelet tree* é uma estrutura de dados sucinta proposta por Grossi, Grupta e Vitter [7] com uso em diversas áreas [12, 5]. Essa estrutura é utilizada para representar uma grande quantidade de caracteres com em espaço comprimido e suporta responder diferentes tipos de consulta em seu conteúdo de forma eficiente, como por exemplo saber quantos caracteres de um certo tipo existe entre duas posições do texto. Por ser inicialmente proposta como uma árvore binária balanceada, Grossi et al. [8] nota que as consultas funcionariam independente do formato da árvore e propõe mudanças no formato para otimizar as consultas. Já Mäkinen e Navarro [10], propõem a modificação do formato da árvore para o mesmo formato da árvore de Huffman para conseguir uma compressão de espaço ainda melhor.

O trabalho de Fonseca e Silva [2] propõe um algoritmo para a construção *online* da *wavelet*

tree no formato de uma árvore binária balanceada que possui performance similar em questão de tempo de processamento comparado aos algoritmos mais utilizados, mas uma melhoria no espaço utilizado por não precisar armazenar o texto como um todo e utilizar pouca memória extra além do necessário para a estrutura.

Este trabalho tem como objetivo propor um algoritmo de construção dinâmico da *wavelet tree* com o formato da árvore de Huffman, não necessitando armazenar o texto em memória nem saber o alfabeto com antecedência.

CAPÍTULO 2

Métodos

Neste capítulo, detalhamos o algoritmo para construção de uma *Wavelet tree* dinâmica no formato da árvore da codificação de Huffman. Primeiramente é necessário entender como funciona a codificação de Huffman e sua construção dinâmica, para então modificarmos essa construção para incluir as informações necessárias para formação da *wavelet tree*.

2.1 Codificação de Huffman

A codificação de Huffman é uma técnica de compactação que tem como entrada um texto T, um alfabeto A de tamanho m, uma lista de pesos positivos $F = (f_i > 0 \mid 0 \leq i < m)$, em que f_i é igual à frequência do caractere a_i em T, e tem como saída um código binário de prefixo de custo mínimo, chamado de código de Huffman. Para entender o que isso significa, definimos:

1. Um código é binário quando o alfabeto de saída é igual à $\{0,1\}$. Isso significa que existe uma lista de valores $R = (r_i \in \{0,1\}^* \mid 0 \leq i < m)$ onde r_i é a representação, ou também chamado de código, do símbolo a_i no alfabeto de saída.

2. É considerado um código de prefixo quando o código de cada símbolo de A em R não é um prefixo de nenhum outro. Por exemplo, com $m = 5$, uma possível representação do alfabeto para ser um código de prefixo seria $R = \{00, 01, 10, 110, 111\}$, enquanto que a representação $R = \{00, 01, 10, 11, 001\}$ é inválida porque 00 é prefixo de 001. Códigos de prefixos também têm a característica de que são unicamente decodificados.

3. Um código é de custo mínimo quando utiliza a representação R que minimiza

$$C(R,F) = \sum_{i=0}^{m-1} len(r_i) \cdot f_i, \tag{2.1}$$

dentre todas as representações possíveis, onde $len(x)$ é igual ao tamanho ou quantidade de *bits* da representação x.

Para gerar a codificação de Huffman do texto T, é necessário construir a árvore de Huffman. A árvore de Huffman é uma árvore estritamente binária em que cada símbolo a_i corresponde a um nó folha de peso f_i, todo nó interno da árvore tem peso igual a soma do peso de seus dois filhos, cada aresta possui um símbolo de $\{0,1\}$ e o código r_i pode ser obtido pelo caminho da raiz até o nó folha de a_i, concatenando os símbolos presentes nas arestas do caminho. É importante notar também que duas arestas que conectam ao mesmo nó pai não podem ter o mesmo símbolo. Sem perda de generalidade, assumimos que a aresta da subárvore esquerda terá o símbolo 0, e a aresta da subárvore direita terá símbolo 1.

Essa árvore pode ser construída seguindo um algoritmo guloso simples. Inicialmente, utilizamos cada símbolo de A e seu respectivo peso em F para construir um conjunto de nós folhas. Então, realizamos um processo guloso, onde os dois nós de menor peso são retirados do conjunto e um novo nó interno é criado, que possui peso igual a soma dos pesos dos dois nós retirados. Esse novo nó interno tem a aresta da subárvore esquerda conectada ao nó retirado de menor peso, e a aresta da subárvore da direita conectada ao outro nó retirado e esse novo nó é então adicionado de volta ao conjunto. Após $n-1$ passos apenas um nó sobrará no conjunto, a raiz de nossa árvore de Huffman, com peso igual a $\sum_{i=0}^{n-1} f_i$. Um exemplo para a árvore de Huffman da palavra *abracadabra* pode ser visto na Figura 2.1.

Para codificar um texto T de tamanho n utilizando a árvore de Huffman, criamos a lista de códigos R a partir das folhas das árvores, e depois simplesmente concatenamos os códigos dos caracteres de T. Para $T = abracadabra$, com $A = \{a,b,c,d,r\}$, usando a árvore da Figura 2.1 temos que $R = \{0,10,1100,1101,111\}$, e como resultado o código de Huffman $H(T) = 0 \cdot 10 \cdot$

$111 \cdot 0 \cdot 1100 \cdot 0 \cdot 1101 \cdot 0 \cdot 10 \cdot 111 \cdot 0$, isto é, $H(T) = 01011101100011010101110$.

A decodificação de um código de Huffman é simples: criamos um ponteiro, inicialmente na raiz da árvore de Huffman, e processamos o código símbolo por símbolo. Para cada novo símbolo do código, o ponteiro desce na árvore utilizando a aresta que possui mesmo valor. Quando esse ponteiro alcança um nó folha, com um caractere e um peso associado, temos a certeza que esse caractere é o caractere da posição atual e o ponteiro volta para a raiz. Esse processo é realizado até o fim do código, e é válido porque código de Huffman é um código de prefixo. Decodificando o código do exemplo anterior, $H = 01011101100011010101110$ e utilizando a Figura 2.1 de referência, iniciamos o ponteiro na raiz e realizamos os seguintes passos:

1. $h_0 = 0$, então o ponteiro desce para a esquerda. Como chegamos em um nó folha com o ponteiro, temos que $T_0 = a$. Ponteiro volta para raiz.
2. $h_1 = 1$, ponteiro desce para a direita.
3. $h_2 = 0$, ponteiro desce para esquerda e $T_1 = b$. Ponteiro volta para raiz.
4. $h_3 = 1$, ponteiro desce pra direita.
5. $h_4 = 1$, ponteiro desce pra direita.
6. $h_5 = 1$, ponteiro desce pra direita e $T_2 = r$. Ponteiro volta para raiz.
7. $h_6 = 0$, ponteiro desce pra esquerda e $T_3 = a$. Ponteiro volta para raiz.
8. $h_7 = 1$, ponteiro desce pra direita.
9. $h_8 = 1$, ponteiro desce pra direita.
10. $h_9 = 0$, ponteiro desce pra esquerda.
11. $h_{10} = 0$, ponteiro desce pra esquerda e $T_4 = c$. Ponteiro volta para raiz.

12. $h_{11} = 0$, ponteiro desce pra esquerda e $T_5 = a$. Ponteiro volta para raiz.
13. $h_{12} = 1$, ponteiro desce pra direita.
14. $h_{13} = 1$, ponteiro desce pra direita.
15. $h_{14} = 0$, ponteiro desce pra esquerda.
16. $h_{15} = 1$, ponteiro desce pra direita e $T_6 = d$. Ponteiro volta para raiz.
17. $h_{16} = 0$, ponteiro desce pra esquerda e $T_7 = a$. Ponteiro volta para raiz.
18. $h_{17} = 1$, ponteiro desce pra direita.
19. $h_{18} = 0$, ponteiro desce pra esquerda e $T_8 = b$. Ponteiro volta para raiz.
20. $h_{19} = 1$, ponteiro desce pra direita.
21. $h_{20} = 1$, ponteiro desce pra direita.
22. $h_{21} = 1$, ponteiro desce pra direita e $T_9 = r$. Ponteiro volta para raiz.
23. $h_{22} = 0$, ponteiro desce pra esquerda e $T_{10} = a$. Ponteiro volta para raiz.
24. Fim do código H.

No final, temos que $T = abracadabra$, conforme esperado.

2.1.1 Construção da árvore de Huffman dinâmica

Descrevemos como funciona a construção da árvore de Huffman de maneira estática, ou seja: tendo acesso ao texto T inteiro e calculando a frequência de cada caractere antes do início da construção. Porém, aplicações que utilizam *stream* de dados podem não possuir acesso ao texto por completo, e por isso se vê necessário o uso de um algoritmo dinâmico para construção da árvore de Huffman, descrito em [1].

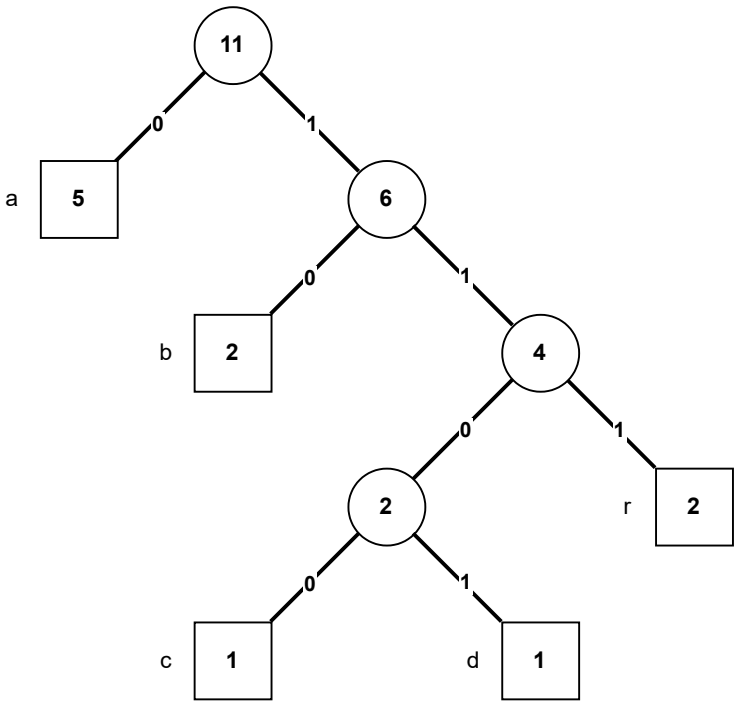

Figura 2.1 Exemplo da árvore de Huffman estática construída sobre *abracadabra*.

Teorema 2.1 (*Propriedade dos irmãos*)**.** *Seja V uma árvore binária completa, F uma lista de pesos positivos para cada nó da árvore, e p a quantidade de folhas da árvore, onde o peso de todo nó interno é igual a soma dos pesos de seus filhos. Então V é uma árvore de Huffman se somente se seus nós podem ser organizados numa sequência $S = (x_1, x_2, ..., x_{2p-1})$ de nós da árvore tal que:*

1. *A sequência de pesos $(f_{x_1}, f_{x_2}, ..., f_{x_{2p-1}})$ está em ordem crescente e*
2. *Para qualquer i $(1 \leq i < p)$, os nós consecutivos x_{2i-1} e x_{2i} são irmãos (possuem o mesmo pai)*

A prova pode ser vista em [1]

A ideia do algoritmo dinâmico de Huffman é atualizar os nós da árvore de Huffman para cada mudança de frequência de um caractere de forma que a propriedade descrita no Teorema 2.1 se mantenha. Pode ser necessária tanto a criação de novos vértices, no caso de um caractere que apareceu pela primeira vez, como também a troca de posição entre nós. Pelas mudanças de nó na árvore, o código de um caractere pode mudar ao longo do processamento de um mesmo texto e, portanto, se vê necessário a reconstrução da árvore com o mesmo algoritmo no momento de decodificação.

A árvore de Huffman dinâmica possui um nó a mais que a árvore de Huffman estática: um nó de peso 0 que será representado por um caractere que não aparecerá em T - no nosso caso utilizaremos #. Esse será o primeiro nó da árvore e único nó de peso 0. Pelo seu peso ser 0, ele sempre será o primeiro da sequência S do Teorema 2.1 e todos os novos nós folha criados serão vizinhos à ele inicialmente. Outra diferença do algoritmo estático é que precisamos definir uma função $g(c)$, que retorna um código binário único para todos os caracteres em A. Esse código será usado apenas na primeira aparição de c no texto.

Seja $T[l : r] = t_l \cdots t_r$ e suponha que processamos o prefixo $T[0 : i]$. Temos V^i, a árvore de Huffman para esse prefixo mais um nó folha de peso 0 para o caractere # que representa todos

os caracteres ainda não vistos e um código H^i. Ao ler o próximo caractere $c = T[i+1]$, temos duas possibilidades:

1. c apareceu anteriormente em $T[0:i]$. O código de c é encontrado igual a versão estática da árvore de Huffman: o valor das arestas no caminho da raiz até a folha.

2. c nunca apareceu antes em $T[0:i]$, então não possui uma folha em V^i. Como essa é a primeira vez que c aparece no texto e não possui um código próprio ainda, utilizaremos o código de # concatenado com $g(c)$.

Após conseguir o código atual de c, atualizaremos V^i para V^{i+1} e teremos um H^{i+1} para $T[:i+1]$. Com isso, podemos descrever como atualizar V^i para V^{i+1}. Caso c não tenha uma folha em V, iremos criar uma nova folha x_{2p} de peso 1 para representar c e um nó interno x_{2p+1} que será pai de x_{2p} e x_0. Essa operação é igual a modificar a sequência S de $(x_0, x_2, ..., x_{2p-1})$ para $(x_0, x_{2p}, x_{2p+1}, x_2, ..., x_{2p-1})$ e isso mantém a propriedade dos irmãos válida para a sequência, já que $f_{x_0} = 0$ e $f_{x_{2p}} = f_{x_{2p+1}} = 1$. Por outro lado, se c já faz parte da árvore, começamos incrementando em 1 o peso f_i correspondente ao nó folha que representa c. Em ambos os casos, se a propriedade 1 do Teorema 2.1 for quebrada com esse incremento, procuraremos o maior j tal que $f_{s_i} > f_{s_j}$, $i < j$ e x_j não é pai de x_i e faremos a troca de posição desses nós, caso exista tal j. Esse processo é repetido sucessivamente para o pai de x_i, que tem seu peso incrementado em 1, até a raiz e pode ser implementado em $O(p)$, que também significa dizer que será $O(A)$. Ao final desse processo, teremos V^{i+1}. Um exemplo da troca de nós pode ser visto na Figura 2.2.

2.2 Wavelet Tree

Seja T um texto de tamanho n sobre um alfabeto A de tamanho m. Para todo subconjunto $A' \subseteq A$, seja $proj(T, A')$ a sequência não contínua, possivelmente vazia, de todos caracteres

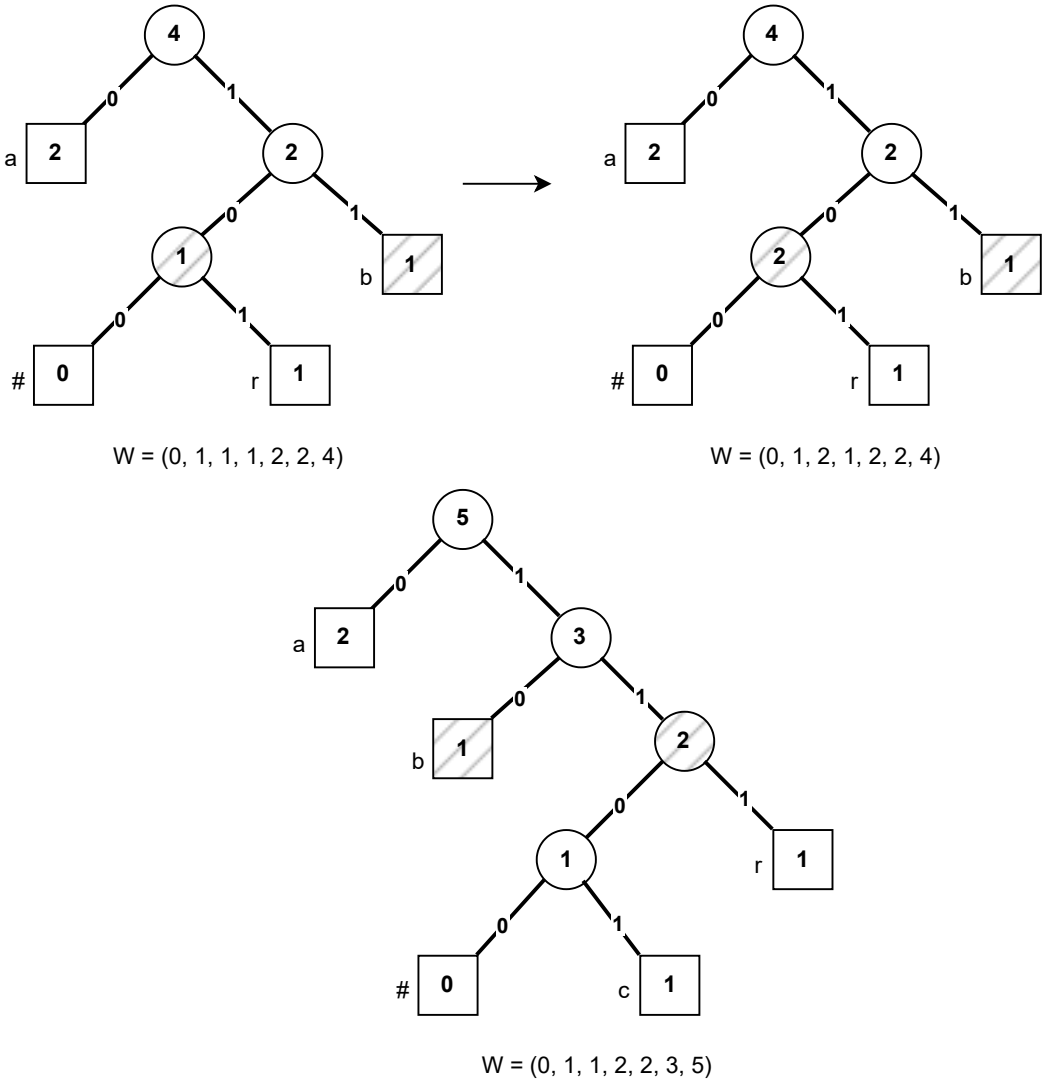

Figura 2.2 Exemplo da transformação da árvore de Huffman no processamento da palavra *abra* para *abrac*. Os nós trocados estão marcados de cinza e W é a lista de pesos ordenados da propriedade dos irmãos.

$t_i \in A'$. Por exemplo, seja $T = abracadabra$ e $A' = \{a,c,d\}$, $proj(T,A') = aacadaa$.

Definição 2.1. A *wavelet tree* (WT) para o texto T sobre o alfabeto A pode ser definida recursivamente como

- Caso $|A| = 1$, então a árvore é apenas um nó folha de rótulo a_0.
- Se não, a árvore possui um nó raiz interno, que representa A. Esse nó possui dois filhos: o filho a esquerda é definido como a WT para o texto $proj(T,A_0)$ sobre o alfabeto A_0 e o filho a direita é definido como a WT para o texto $proj(T,A_1)$ sobre o alfabeto A_1, onde $A_0 \cup A_1 = A$. Também armazenamos no nó um *bitvector* B de tamanho $|T|$, onde $b_i = 0$ se $t_i \in A_0$ ou $b_i = 1$ se $t_i \in A_1$.

A WT é uma estrutura que provê consultas de *access*, *rank* e *select*. Seja $W(T,A)$ a WT sobre T em A e $T[l,r] = t_l \cdots t_r$, definimos que $W.access(i)$ é igual t_i, $W.rank(c,i)$ é igual ao número de ocorrências de c em $T[0:i]$ e $W.select(c,j)$ é igual à posição em T que o caractere c aparece pela $j-$ésima vez. Então, se $i = W.select(c,j)$, temos $W.rank(c,i) = j$.

Na construção da WT, podemos definir diferentes formas de particionar A. Uma das formas mais utilizadas é a de dividir A na metade a cada iteração. Essa forma de particionar A gera uma WT balanceada, que, como demonstrado em [11], pode utilizar de $n\lg(m) + o(n\lg(m))$ bits de memória e complexidade de consulta $O(\lg(m))$. Outra forma de particionar A, explorado em [10], é a de dividir A de forma que a árvore resultante tenha o mesmo formato da árvore de Huffman. É importante notar que, para essa divisão do alfabeto, é também necessário armazenar em cada nó interno uma lista D, onde, para todo $c \in A$, $d_c = 0$ se $c \in A_0$ e $d_c = 1$ se $c \in A_1$. Essa lista D será usada para percorrer a árvore até a folha de um caractere específico. Essa abordagem utiliza $|H|(1 + o(1))$ bits de memória e tem complexidade de consulta $O(\xi)$, onde ξ é a entropia de ordem zero do texto e é definida como

$$\xi = -\sum_{i=0}^{m} \frac{f_i}{n} \log\left(\frac{f_i}{n}\right)$$

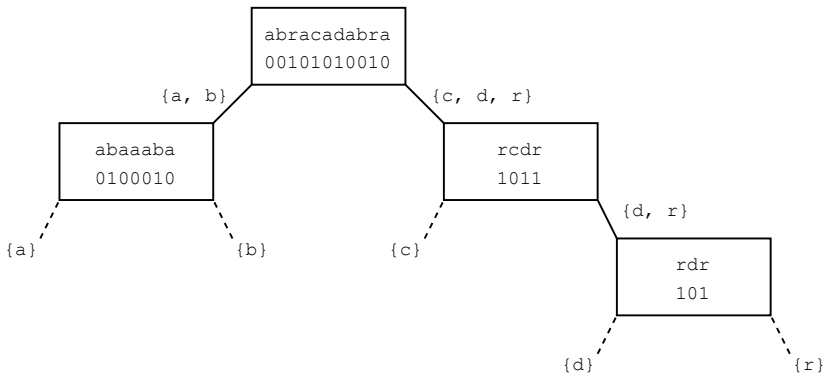

Figura 2.3 A *wavelet tree* balanceada do texto *abracadabra* sobre o alfabeto a,b,c,d,r. Apenas os *bitvectors* são armazenados. O texto e alfabeto são para fins ilustrativos.

Um exemplo de uma WT balanceada e uma WT no formato de Huffman podem ser visto nas Figura 2.3 e Figura 2.4, respectivamente.

2.2.1 Construção dinâmica da *wavelet tree* no formato da árvore de Huffman

Para a construção da WT no formato da árvore de Huffman de forma dinâmica, modificaremos a construção descrita em Seção 2.1.1 para não só manter o formato da árvore, como também armazenar informação em cada nó interno sobre o *bitvector* B e a lista D descritos anteriormente.

Suponha que processamos o prefixo $T[0:i]$. Temos W^i, a WT com formato da árvore de Huffman para esse prefixo mais um caractere # que representa todos os caracteres ainda não vistos. Ao ler o próximo caractere $c = T[i+1]$, atualizaremos W^i para W^{i+1}. Primeiramente, adicionaremos o novo *bit* referente à c nos *bitvectors* necessários. Caso c já tenha uma folha em W^i, percorremos a árvore iniciando da raiz até a folha e, para cada nó interno do caminho, concatenamos em B um novo *bit* de valor igual à d_c. Caso c não tenha um nó folha em W^i, sabemos que, pela construção da árvore de Huffman dinâmica, o nó folha referente a c será criado vizinho ao nó folha de #. Então iremos percorrer da raiz até a folha de #, concatenando

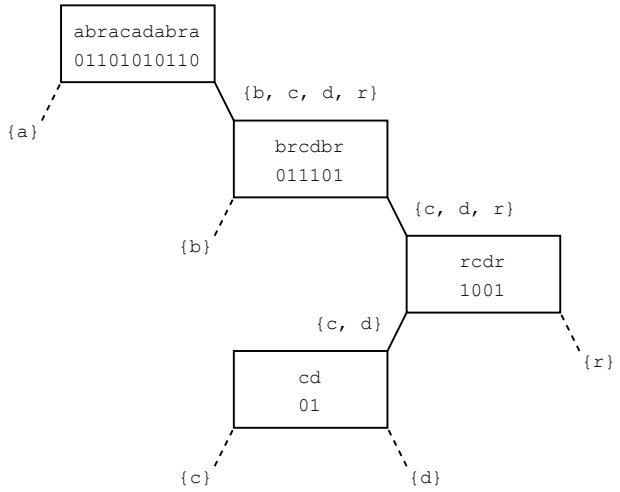

Figura 2.4 A *wavelet tree* no formato Huffman do texto *abracadabra* sobre o alfabeto a,b,c,d,r. Apenas o código de Huffman de cada caractere e os *bitvectors* são armazenados. O texto e alfabeto são para fins ilustrativos.

em B um novo *bit* igual à $d_\#$ e adicionando $d_c = d_\#$. Agora, seguindo a construção da árvore de Huffman dinâmica, incrementamos em 1 o peso da folha de c caso ela já exista ou criamos os dois novos nós - um interno e a folha referente à c, e os adicionamos na árvore. Esse novo nó interno terá, além do peso igual à 1, $B = 0$ e $d_c = 0, d_\# = 1$.

Após essa modificação, a árvore de Huffman pode sofrer mudanças no seu formato para manter a propriedade dos irmãos. As mudanças no formato da árvore são realizadas por uma sequência de trocas entre dois nós e cada troca de posição pode modificar o alfabeto que um ou mais nós internos representam e, por consequência, modificar tanto B quanto D desses nós. Para entender quais nós serão modificados em cada troca, é necessário entender o Teorema 2.2.

Teorema 2.2. *Seja $path(x_1,x_2)$ o menor caminho entre dois nós x_1 e x_2 da árvore. Definimos $LCA(x_1,x_2)$ como sendo o ancestral comum mais profundo (ou lowest common ancestor) entre x_1 e x_2, ou seja, o nó mais próximo da raiz no caminho entre x_1 e x_2. Sejam x_i e x_j os nós da árvore que serão trocados e $x_k = LCA(x_i,x_j)$. O conjunto de nós que precisam ter o $bitvector$ B e lista D atualizados é igual a $path(x_k,x_i) \cup path(x_k,x_j) \setminus \{x_i,x_j\}$.*

Demonstração. Para cada nó v da árvore, sendo A_i o subconjunto de A que o vértice i representa, podemos classificá-lo em um dos grupos em relação a um nó $x \in \{x_i, x_j\}$:

1. $A_v \subseteq A_x$. Isso significa que v faz parte da subárvore de x. É fácil ver que não precisamos modificar v nesse caso pois esse nó será movido com x na troca.

2. $A_v \cap A_x = \emptyset$. É trivial ver que não precisamos modificar v nesse caso.

3. $A_x \subset A_v$. Podemos dividir esse caso em três:

   $v \in (path(r, x_k) \setminus \{x_k\})$ Nesse caso v é ancestral de x_k, logo não precisamos mudar os dados em v porque ambos x_i e x_j continuarão em sua subárvore e continuarão pertencendo ao mesmo filho direto.

   $v = x_k$ Sendo v o $LCA(x_i, x_j)$, precisamos atualizar seus dados pois houve uma inversão nos *bits* que representam os caracteres em A_{x_i} e A_{x_j}

   $v \notin (path(r, x_k) \setminus \{x_k\})$ Neste caso, v é ancestral de x porém ainda descendente de x_k. Então, como v possui apenas um dos nós trocados como filho, é necessário retirar as informações do nó que sairá de sua subárvore e adicionar as informações do novo nó. □

   Nas Figura 2.5, Figura 2.6 e Figura 2.7 temos exemplos desses três casos para a WT da palavra *abracadabra*.

Sabendo quais nós precisam ser modificados, agora descrevemos como serão modificados. Sejam x_i e x_j os nós da árvore que serão trocados, $x_k = LCA(x_i, x_j)$ e v um dos nós que será modificado. Sem perda de generalidade, suponha que v é um ancestral de x_i diferente de x_k. O bitvector B_v representa as posições de T com caracteres em A_v e a lista D_v que mapeia cada caractere em A_v para uma das arestas, na direção de sua folha. Por exemplo, na Figura 2.4, o nó de $T = brcdbr$ possui $B = 011101$ e $D = \{b : 0, c : 1, d : 1, r : 1\}$. Com a remoção de x_i da subárvore de v, A_v, B_v e D_v precisam ser modificados para não mais conter os caracteres

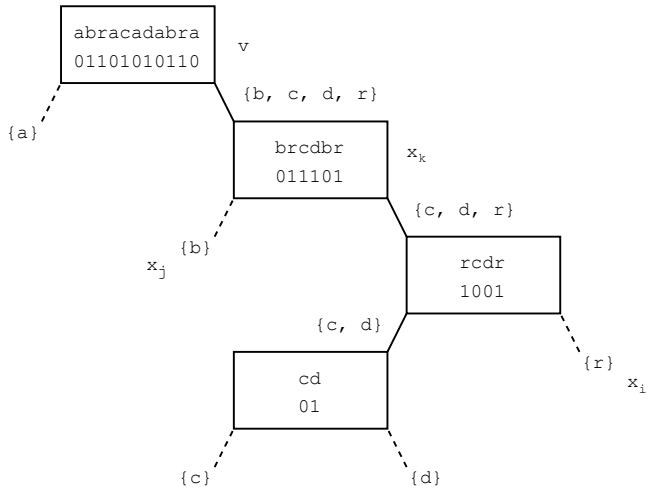

Figura 2.5 Exemplo quando o vértice v não será modificado na troca dos vértices x_i e x_j, onde $LCA(x_i, x_j) = x_k$.

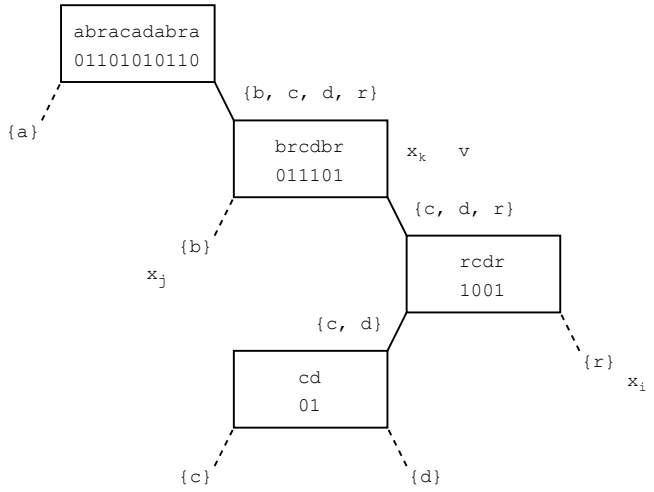

Figura 2.6 Exemplo quando o vértice v será modificado na troca dos vértices x_i e x_j, onde $LCA(x_i, x_j) = x_k$ e $x_k = v$.

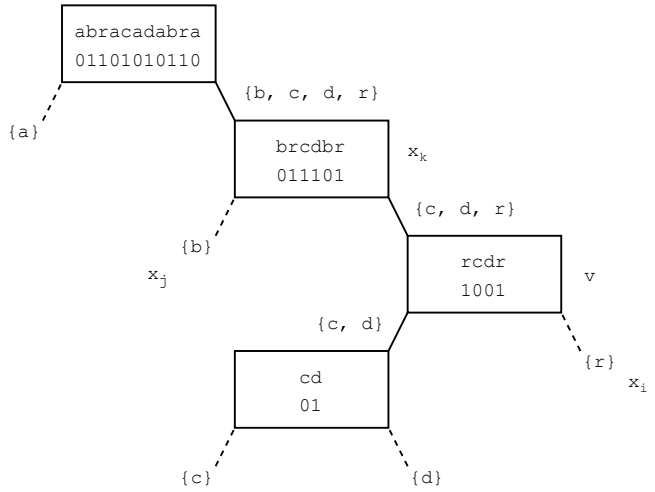

Figura 2.7 Exemplo quando o vértice v será modificado na troca dos vértices x_i e x_j, onde $LCA(x_i, x_j) = x_k$.

presentes em A_{x_i}. Com a adição de x_j, é necessário adicionar os caracteres de A_{x_j} em A_v, B_v e D_v. A adição e remoção em A_v e D_v são triviais de serem realizadas, mas a modificação de B_v é mais complexa pois não sabemos qual caractere é referente a um *bit* específico, então precisamos criar uma maneira de achar as posições relevantes para realizar tanto a remoção quanto a adição.

Para o caso que $v = x_k$, não é necessário fazer a operação de remoção ou adição de *bits*, pois não houve remoção ou adição de nós em sua subárvore. Mas é necessário inverter o *bit* em D_v e B_v para todo $c \in A_{x_i} \cup A_{x_j}$. Para a inversão de *bits* em B_v, é necessário utilizar os dados das posições encontrados na remoção dos *bits* dos nós no caminho até x_i e x_j.

2.2.1.1 Remoção de *bits*

Para a remoção, começamos de x_i e subimos na árvore até x_k, removendo os *bits* necessários no caminho. Vamos utilizar a função $select(b, j)$ do *bitvector*, que retorna o índice no vetor do j-ésimo *bit* b. Por exemplo, para o *bitvector* $B = 110011$ temos

- $B.select(1,0) = 0$
- $B.select(0,0) = 2$
- $B.select(1,3) = 5$

A ideia é utilizar a informação de um nó para atualizar o *bitvector* do pai e assim sucessivamente até chegar ao *LCA*. Para atualizar o pai, é necessário saber as posições em seu *bitvector* que correspondem à caracteres em A_{x_i}. Seja *ant* um nó em $path(x_i, x_k)$, *cur* o pai de *ant* e P um vetor onde p_i é igual ao índice em B_{cur} do i-ésimo caractere de $proj(T, A_{x_i})$. Inicialmente $ant = x_i$, *cur* é o pai de x_i e $P = (p_i = i \mid 0 \leq i < |proj(T, A_{x_i})|)$, que são os índices em B_{ant} correspondentes aos caracteres de A_{x_i}. Seja y o valor da aresta utilizado para subir de *ant* para *cur* (0=esquerda, 1=direita). Para modificarmos B_{cur} vamos primeiro atualizar o vetor P, que está com os índices de B_{ant}, para ter os índices de B_{cur}. Sabendo que em B_{cur} todo *bit* igual à y se refere à valores em A_{ant}, para atualizarmos os índices em P basta fazer $(p_i = B_{cur}.select(y, p_i) \mid 0 \leq i < |proj(T, A_{x_i})|)$ e em seguida remover todos esses índices de B_{cur}. Após a remoção, atualizamos *ant* para ser igual à *cur* e *cur* para ser o novo pai de *ant*. Esse processo é repetido até que *cur* seja igual à x_k. Quando *cur* for igual à x_k, faremos o mesmo processo para calcular P mas ao invés de fazermos a remoção dos *bits*, faremos a inversão como descrito na demonstração do Teorema 2.2. A figura Figura 2.8 mostra um exemplo do cálculo de P em uma troca de nós.

Quando *cur* for igual a x_k, faremos o mesmo processo para calcular *pos* mas ao invés de fazermos a remoção dos *bits*, faremos a inversão.

2.2.1.2 Adição de *bits*

A adição é realizada após a remoção e será feita do jeito inverso, de cima para baixo, começando em x_k e descendo até x_j, que será a nova posição de x_i. Vamos utilizar a função $rank(b, j)$ do *bitvector*, que retorna quantos *bits* iguais a b tem até a posição j. Para o *bitvector* $B = 110011$ temos

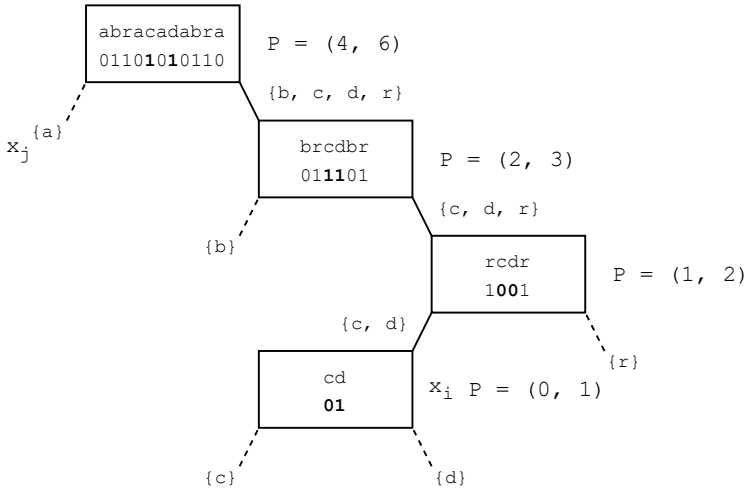

Figura 2.8 Exemplo de valores de P na remoção dos *bits* de x_i na troca com x_j. Os *bits* em negrito serão removidos.

- $B.rank(1,0) = 1$
- $B.rank(0,2) = 1$
- $B.rank(1,5) = 4$
- $B.rank(0,0) = 0$

Seja *ant* um nó em $path(x_k, x_j)$, *cur* o filho de *ant* também em $path(x_k, x_j)$. Seja L um vetor onde l_i é igual posição onde devemos adicionar o i-ésimo *bit* de $proj(T, A_{x_i})$ em B_{cur}. Precisamos processar L para todos os nós no caminho de x_k e x_j e adicionar os *bits* nas posições específicas. Inicialmente, $ant = x_k$ e *cur* é o filho de x_k mais próximo de x_j. Para o valor inicial de L, usaremos a informação de P calculada na inversão dos *bits* de x_k. Sendo y o valor da aresta utilizado no caminho de *ant* para *cur* (0=esquerda, 1=direita), para saber onde inserir os novos *bits* em B_{cur}, basta saber quantos *bits* iguais a y aparecem antes em B_{ant}. Então, $L = (l_i = B_{x_k}.rank(y, p_i) - 1 \mid 0 \leq i < |proj(T, A_{x_i})|)$. Adicionamos em B_{cur} o *bit* y na posição l_i e atualizamos $L = (l_i = B_{cur}.rank(y, l_i) - 1 \mid 0 \leq i < |proj(T, A_{x_i})|)$. Esse processo é repetido até que *cur* seja igual à x_j, que não precisa ser modificado. A figura Figura 2.9 mostra um

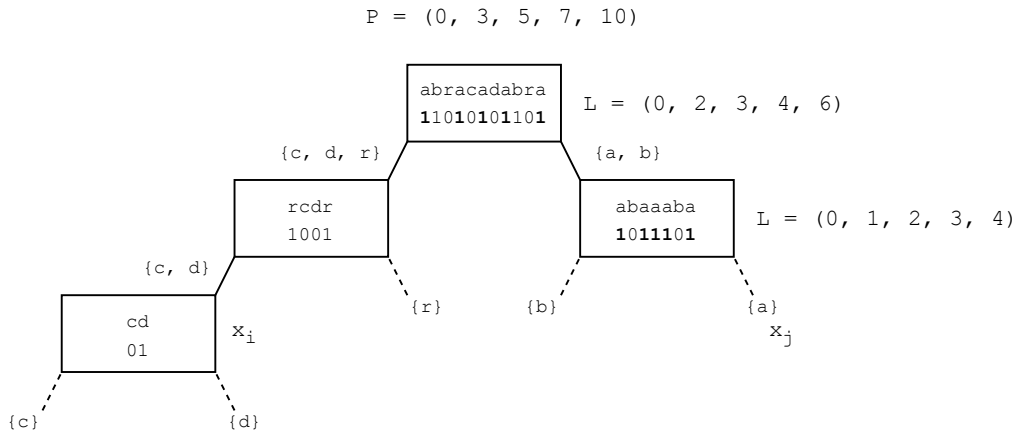

Figura 2.9 Exemplo de valores de L na adição dos *bits* de x_j na troca com x_i. Os *bits* em negrito serão adicionados.

exemplo do cálculo de L em uma troca de nós.

2.3 Implementação

Com as operações de remoção e adição dos *bits* definidas, finalizamos as modificações necessárias do algoritmo de Huffman dinâmico para suportar também uma *wavelet tree* dinâmica. A complexidade de uma atualização é $O(m^2 * p * r)$, onde m é o tamanho do alfabeto processado, p a quantidade de *bits* trocados e r é a maior complexidade entre as operações *rank*, *select*, *insert* e *remove* do *bitvector* utilizado.

A implementação em C++ do algoritmo descrito aqui pode ser encontrada no repositório abaixo

`https://github.com/icarog/tcc`

Alguns detalhes importantes da implementação: utilizamos o *bitvector* dinâmico descrito por Prezza [14] e melhorado por Dönges et al [3]. Fizemos uma otimização para decidir se vale

mais a pena iterar por todo o *bitvector* ao invés de fazer várias consultas de *rank* e *select*. Além da *wavelet tree*, é possível retornar o código de Huffman dinâmico do texto.

CAPÍTULO 3

Resultados

3.1 Dados de teste

Para o teste do método desenvolvido, extraímos textos de Inglês, sequências de DNA e sequências de proteínas de até $3 \cdot 10^6$ caracteres do corpus Pizza&Chili[13].

3.2 Avaliação

Os experimentos foram realizados em um computador com sistema operacional Windows 10, com um processador AMD Ryzen 7 3700X 8-Core 3.60 GHz e 16GB de RAM 3200MHz.

As principais métricas avaliadas serão o tempo de execução para processar todas as atualizações na WT, quantos nós foram trocados e quantos *bits* foram modificados em uma troca de nó em todas atualizações.

Os dados coletados para os diferentes tipos de textos e tamanhos podem ser vistos na Tabela 3.1, Tabela 3.2 e Tabela 3.3. Podemos notar que a quantidade de *bits* trocados ao final de todas atualizações é maior que o tamanho total do texto em todas as tabelas e também na Figura 3.2. Também é importante observar o crescimento linear desse número de trocas ao longo do tamanho do texto, justificando o aumento do tempo de processamento na Figura 3.1. Como discutido na Seção 2.3, o processamento total do texto pode ser quadrático se o número de trocas crescer linearmente ao tamanho do texto, já que depende de ambos fatores. Tamanhos maiores não foram testados por causa do grande tempo de processamento.

Tamanho do texto	Tempo (ms)	Troca de nós	Troca de *bits*	Alfabeto
10000	80.15	2169	887493	81
50000	236.01	3734	2721971	87
100000	480.87	4515	5396661	90
500000	1958.99	6168	17437093	94
1000000	6020.87	7936	65173480	106
2000000	14985.71	8956	118227207	107
3000000	20987.89	9978	187791347	110

Tabela 3.1 Métricas para textos em inglês de diferentes tamanhos

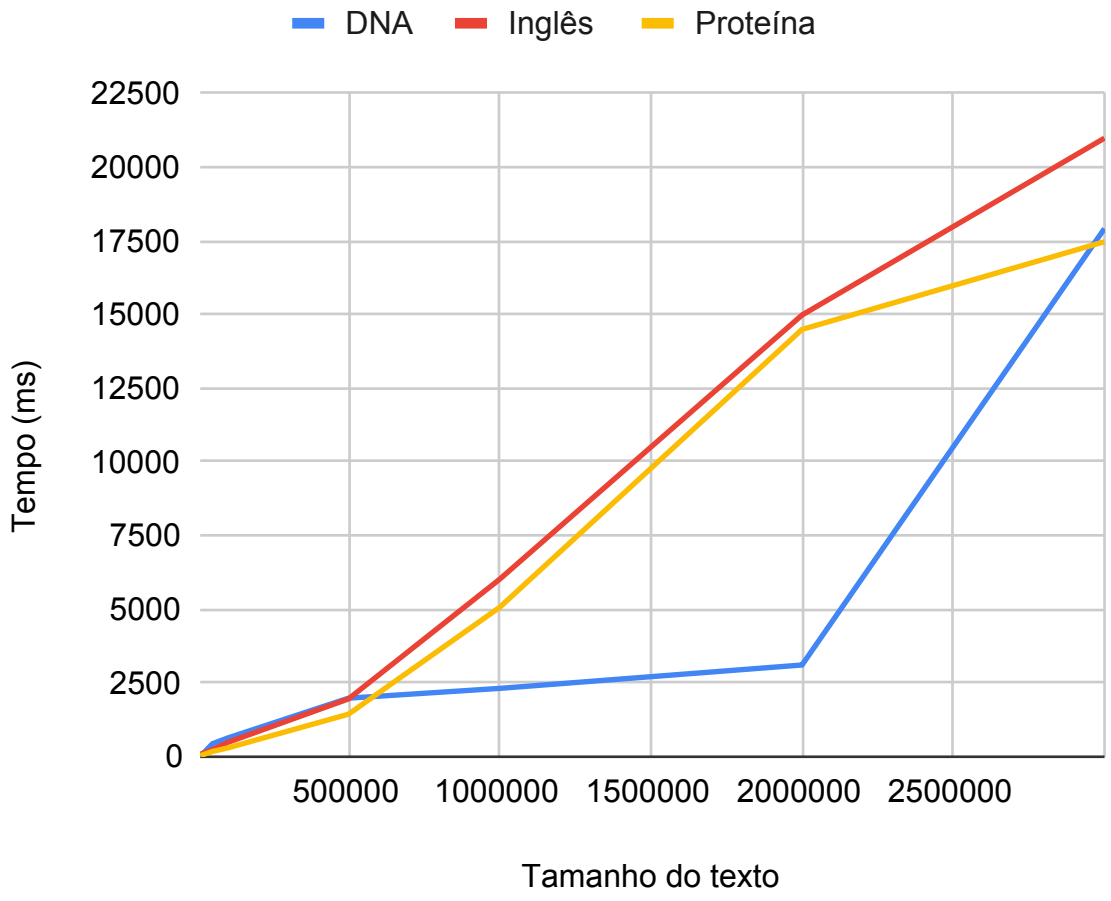

Figura 3.1 Tempo de processamento para cada tamanho de texto de inglês, DNA e proteína.

Tamanho do texto	Tempo (ms)	Troca de nós	Troca de *bits*	Alfabeto
10000	7.79	72	196331	4
50000	437.06	258	9152145	4
100000	630.80	308	12887742	4
500000	1983.54	393	42864207	4
1000000	2316.17	400	47653798	4
2000000	3109.91	409	62806019	4
3000000	17917.04	553	403465839	4

Tabela 3.2 Métricas para sequências de DNA de diferentes tamanhos

Podemos fixar o tamanho do texto e analisar como os diferentes tipos de texto se comportam. Nota-se que a sequência de DNA possui um tempo de processamento maior quando o tamanho é igual à 100000, comparado com os outros dois. Isso acontece porque, como a sequência de DNA possui um alfabeto menor, a quantidade de *bits* trocados é grande já que os *bitvectors* estão mais concentrados e possuem uma chance maior de serem um dos nós envolvidos na troca, enquanto que os outros dois tipos possuem mais nós com *bitvectors* pequenos. Esse comportamento fica mais claro quando comparado a coluna de troca de *bits*. Isso pode ser visto claramente também analisando a diferença de número de troca de *bits* na sequência de DNA entre os tamanhos 2000000 e 3000000.

Analisando os textos de tamanho 3000000, podemos ver que a sequência de proteínas foi a mais rápida dos três, mas quase igual à de DNA. Olhando apenas para o número de troca de *bits*, poderia-se estimar que a sequência de DNA deveria ser a mais lenta dos três tipos. Isso mostra o impacto que um pequeno alfabeto pode ter no tempo de processamento, tendo quase três vezes mais o número de trocas que a sequência de proteínas e o mesmo tempo de processamento.

Com isso, podemos notar que a complexidade do algoritmo possui termos dependentes entre si. Mais especificamente, quanto menor o alfabeto, maior as chances de ter uma troca de nós que afeta muitos *bits*, mas menor será o caminho percorrido até o LCA entre os dois nós da troca. Isso torna o cálculo do tempo real de processamento do algoritmo algo não trivial para

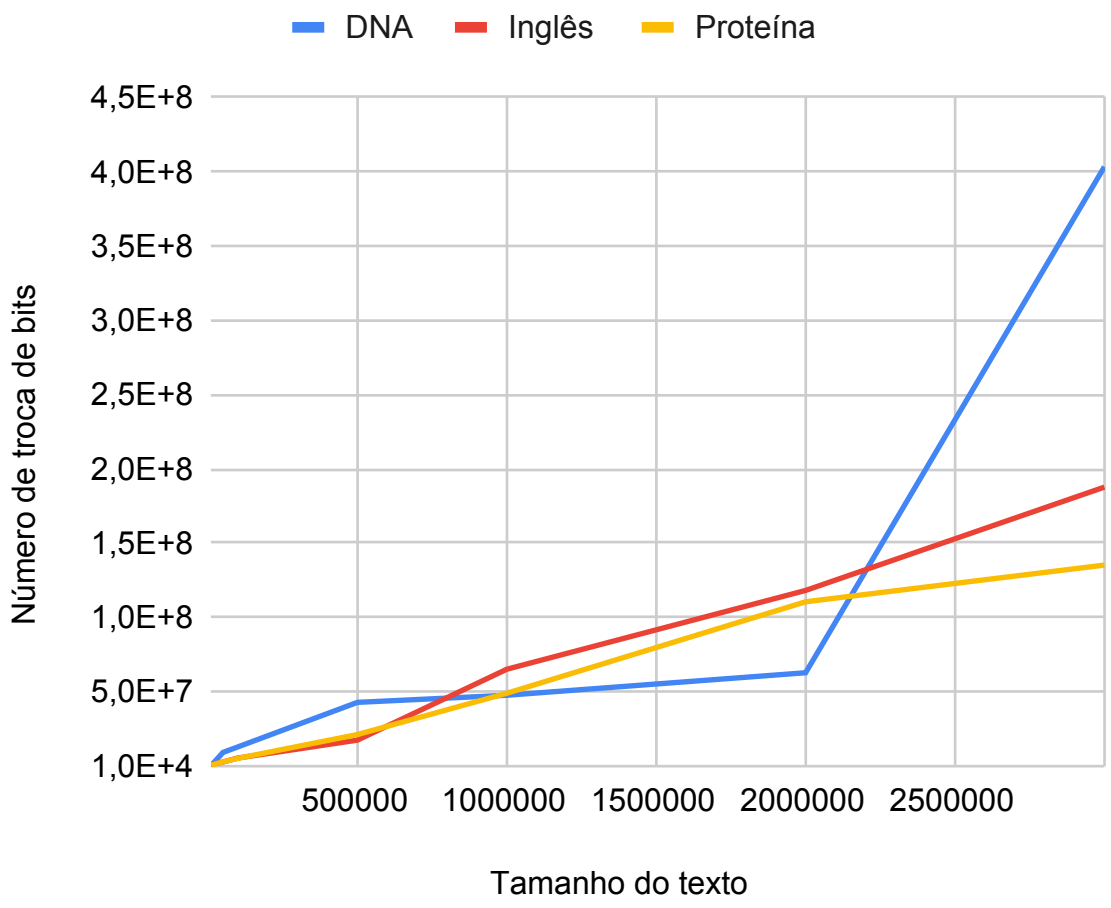

Figura 3.2 Número de troca de *bits* para cada tamanho de texto de inglês, DNA e proteína.

Tamanho do texto	Tempo (ms)	Troca de nós	Troca de *bits*	Alfabeto
10000	32.70	817	599494	20
50000	175.98	1046	2786206	20
100000	293.82	1146	5182571	20
500000	1439.92	1309	21202639	20
1000000	5066.84	1417	48987888	21
2000000	14497.23	1529	110578342	23
3000000	17472.64	1562	135297287	23

Tabela 3.3 Métricas para sequências de proteínas de diferentes tamanhos

cada tipo de texto, além de depender da quantidade de nós que será trocada que é diretamente relacionada com a entropia.

CAPÍTULO 4

Conclusões

Nesse trabalho, propusemos um algoritmo para a construção *online* de uma *wavelet tree* no formato da árvore de Huffman. A principal característica do trabalho foi a modificação do algoritmo dinâmico de construção da árvore de Huffman para também armazenar informações necessárias para ser uma *wavelet tree*.

A principal vantagem da solução proposta é o fato de não necessitar armazenar o texto em memória, que pode ser uma característica desejada ou necessária. Os resultados experimentais mostram que o tempo de processamento pode ser um problema pelo número de *bits* trocados em mudanças no formato da árvore. Uma possível solução para esse problema seria realizar as mudanças no formato da árvore após processar um bloco do texto ao invés de cada caractere. Isso poderia deixar o formato da árvore mais estável, já que mais mudanças ocorrem quando a diferença de frequência dos caracteres difere em um.

Mais melhorias podem ser feitas para alcançar melhor desempenho prático. Por exemplo, conforme discutido na Seção 2.1.1, no Teorema 2.1, dois nós de mesmo peso podem ser considerados iguais no quesito de troca de nós e, sabendo disso, podemos fazer uma escolha melhor de nós para se trocar de forma que a quantidade total de *bits* trocados no final seja a menor possível.

Outra possível melhoria seria trocar a implementação de *bitvector* utilizada para alguma mais eficiente, já que a atual não possui uma operação de *select* e *rank* otimizada para o *bit* 0.

Referências Bibliográficas

[1] Maxime Crochemore and Wojciech Rytter. Text Algorithms. Oxford University Press, Inc.198 Madison Ave. New York, NY, United States, October 1994.

[2] Paulo G. S. da Fonseca and Israel B. F. da Silva. Online Construction of Wavelet Trees. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, 16th International Symposium on Experimental Algorithms (SEA 2017), volume 75 of Leibniz International Proceedings in Informatics (LIPIcs), pages 16:1–16:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[3] Saska Dönges, Simon J. Puglisi, and Rajeev Raman. On dynamic bitvector implementations. In 2022 Data Compression Conference (DCC), pages 252–261, 2022.

[4] Newton Faller. An adaptive system for data compression. In Record of the 7-th Asilomar Conference on Circuits, Systems and Computers, pages 593–597, 1973.

[5] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. Information and Computation, 207(8):849–866, 2009.

[6] R. Gallager. Variations on a theme by huffman. IEEE Transactions on Information Theory, 24(6):668–674, 1978.

[7] Roberto Grossi, Ankur Gupta, and Jeffrey Vitter. High-order entropy-compressed text indexes. Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, 11 2002.

[8] Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. When indexing equals compression: experiments with compressing suffix arrays and applications. In SODA, volume 4, pages 636–645, 2004.

[9] David A Huffman. A method for the construction of minimum-redundancy codes. Proceedings of the IRE, 40(9):1098–1101, 1952.

[10] Veli Mäkinen and Gonzalo Navarro. New search algorithms and time/space tradeoffs for succinct suffix arrays. Technical rep. C-2004-20 (April). University of Helsinki, Helsinki, Finland, 2004.

[11] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In Alberto Apostolico, Maxime Crochemore, and Kunsoo Park, editors, Combinatorial Pattern Matching, pages 45–56, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[12] Gonzalo Navarro. Wavelet trees for all. Journal of Discrete Algorithms, 25:2–20, 2014. 23rd Annual Symposium on Combinatorial Pattern Matching.

[13] Paolo Ferragina e Gonzalo Navarro. Pizza&chili corpus. `https://pizzachili.dcc.uchile.cl/`, Acessado em 2023-01-20.

[14] Nicola Prezza. A framework of dynamic data structures for string processing. In International Symposium on Experimental Algorithms. Leibniz International Proceedings in Informatics (LIPIcs), 2017.