Rodrigo César de Castro Miranda

Um algoritmo para pesquisa aproximada de padrões baseado no método de Landau e Vishkin e uso de arranjos de sufixos para reduzir o uso de espaço

Brasília

2006

Rodrigo César de Castro Miranda

Um algoritmo para pesquisa aproximada de padrões baseado no método de Landau e Vishkin e uso de arranjos de sufixos para reduzir o uso de espaço

Descrevemos uma variante do algoritmo de Landau e Vishkin que substitui o uso de árvores de sufixos nesse algoritmo por arranjos de sufixos, com o objeto de diminuir o espaço utilizado.

Orientador:
Mauricio Ayala-Rincón

Universidade de Brasília
Mestrado em Informática

Brasília

2006

Dedico esta dissertação a meus pais,
pelo carinho e exemplo de honestidade e trabalho,
ao meu irmão, pela amizade e apoio,
e a minha esposa, pelo amor, pelo apoio
e por compartilhar comigo os dias de sol e os dias de chuva.

Agradecimentos

Dedico meus sinceros agradecimentos para:

– o professor doutor Maurício Ayala-Rincón, pela orientação, pela paciência, pelo exemplo, pelo incentivo, pelo trabalho, e pela amizade;

– aos colegas Daniel Sobral, Leon Sólon, Marcus Yuri, Ricardo Lima, Rinaldi Neto, Thomas Sant'Ana, e Wilton José, pelo apoio e inúmeras discussões e sobre computação, o universo e o número 42;

– aos colegas Rinaldi Maya Neto e Wilton José Pereira dos Santos, pela contribuição na revisão do trabalho;

– a equipe da secretaria de pós-graduação do CIC;

– a minha família, pelo apoio e compreensão;

– a todos os colegas do Mestrado em Informática da UnB.

Resumo

A pesquisa aproximada de padrões em um texto é um problema importante para a ciência da computação. A pesquisa de algoritmos eficientes para solucionar esse problema influencia o desenvolvimento de aplicações em áreas como biologia computacional e pesquisa textual em grandes massas de dados (como a *web*, por exemplo). Mas para o tratamento de volumes de informação da magnitude envolvida nessas aplicações, o uso eficiente de tempo e espaço é uma condição essencial. A solução mais conhecida para esse problema é um algoritmo de programação dinâmica com complexidade $\theta(mn)$ para duas palavras P e T de comprimento m e n. Landau e Vishkin desenvolveram um algoritmo que usa árvores de sufixos para acelerar a computação de caminhos da tabela de programação dinâmica que correspondem às ocorrências de um padrão em um texto com no máximo k diferenças, cuja complexidade de tempo e espaço está em $\theta(kn)$. Nesse algoritmo as árvores de sufixos são utilizadas para permitir o cálculo em tempo constante do comprimento do maior prefixo comum entre quaisquer dois sufixos de P e T. Propusemos e implementamos uma variação do algoritmo de Landau e Vishkin que usa arranjos de sufixos para esse cálculo, melhorando o uso de espaço e mantendo um desempenho similar, e apresentamos a relação de custo e benefício de cada alternativa examinada. Com isso, desenvolvemos um mecanismo que torna possível substituir o uso de árvores de sufixos por arranjos de sufixos em determinadas aplicações, com ganho no uso de espaço, o que permite processar um volume maior de informações. A modificação realizada não é trivial, pois os algoritmos e estruturas de dados utilizadas são complexos, e os parâmetros de desempenho e uso de espaço rigorosos.

Abstract

Approximate pattern matching in an important problem in computer science. The research of efficient solutions for this problem influences the development of applications in disciplines such as computational biology and searching the web, and in order to be able to handle such massive ammounts of information the efficient use of computational resources is a necessary condition. The most known solution for the approximate pattern matching problem is a dynamic programming algorithm which has $\theta(mn)$ complexity given two strings P and T of length m and n. Landau and Vishkin developed a $\theta(kn)$ algorithm which uses suffix trees for a faster computation of paths along the dynamic programming table that correspond to matches of a pattern in a text with at most k differences. In this algorithm the suffix trees are used for a constant-time calculus of the longest common extension of any two suffixes of P and T. We proposed and implemented a variation of Landau and Vishkin's algorithm which uses suffix arrays for this calculus, improving the space requirements of the algorithm while keeping a similar running time performance, and present the costs and benefits of each algorithm. In order to achieve this we developed a technique that makes it possible to replace the use os suffix trees for suffix arrays in certain applications with an improved memory usage that allows the processing of a larger ammount of information. The modifications done were not trivial ones, as the algorithms and data structures involved are very complex, and the parameters for accepted running time performance and space usage are very rigorous.

Sumário

1 Introdução

A pesquisa aproximada de padrões em um texto é um problema importante para a ciência da computação com aplicações para a biologia computacional, bancos de dados de textos, e a pesquisa de textos na *web*. A pesquisa de algoritmos eficientes para solucionar esse problema influencia o desenvolvimento de aplicações em áreas como biologia computacional e pesquisa textual de grandes massas de dados como a *web* e bancos de dados de textos (como a pesquisa da jurisprudência de um tribunal, por exemplo). Mas para o tratamento de volumes de informação da magnitude envolvida nessas aplicações o uso eficiente de tempo e espaço é uma condição essencial.

O problema básico para a pesquisa aproximada de padrões é o problema de distância de edição entre duas palavras P e T, que foi proposto por Vladimir I. Levenshtein em 1965, e trata de encontrar o número mínimo de operações que transformariam P em T ou vice-versa. A solução mais conhecida para esse problema é um algoritmo de programação dinâmica com complexidade $\theta(mn)$ quando o comprimento de P e T são m e n, respectivamente. Levenshtein mostrou que a distância de edição de P e T será obtida a partir das distâncias de edição entre os prefixos de comprimento $m-1$ e $n-1$ de P e T, e entre esses prefixos e P e T, o que nos nos dá uma relação de recorrência que podemos resolver de forma eficiente com um algoritmo de programação dinâmica.

O algoritmo de pesquisa de padrões aproximados é uma variação do algoritmo de distância de edição. Nesse caso buscamos subpalavras de T cuja distância de edição com respeito a P é a menor possível. Do ponto de vista algorítmico a diferença é a condição inicial do algoritmo.

Em 1970 Needleman e Wunsch(1) adaptaram o algoritmo de Levenshtein para o processamento de seqüências biológicas. O problema de distância de edição é um problema de *minimização* no qual se busca o conjunto mínimo de operações para transformar uma palavra na outra, que não leva em conta informações da Biologia sobre a evolução de seqüências biológicas. Needleman e Wunsch propuseram uma medida de similaridade ou

semelhança entre duas seqüências, que é calculada de forma similar à distância de edição, mas transformando o cálculo numa *maximização* na qual se busca ter a máxima similaridade. O algoritmo de Needleman e Wunsch é conhecido com algoritmo de *alinhamento global*.

Smith e Waterman(2) posteriormente propuseram uma modificação ao algoritmo de Needleman e Wunsch que é adequada para encontrar regiões de grande similaridade entre duas seqüências. O algoritmo de Smith e Waterman é conhecido como algoritmo de *alinhamento local*. Ambos algoritmos usam a estrutura básica do algoritmo de distância de edição.

Na forma tradicional, o algoritmo de cálculo de distância de edição usa uma tabela de programação dinâmica de tamanho $n + 1 \times m + 1$, e dessa forma sua complexidade de espaço está em $\theta(mn)$. Pode-se usar uma técnica desenvolvida por Hirschberg(3) para alterar a complexidade de espaço do algoritmo para $\theta(n)$, dobrando o tempo de execução.

Landau e Vishkin(4) desenvolveram e apresentaram em 1986 um algoritmo que usa árvores de sufixos para acelerar a computação de caminhos da tabela de programação dinâmica de $\theta(mn)$ para $\theta(nk)$ onde k é a quantidade máxima de diferenças. O uso de espaço do algoritmo de Landau e Vishkin é $\theta(kn)$, mas assim como na programação dinâmica podemos modificar o algoritmo para executar usando espaço $\theta(n)$ para calcular as posições onde P está em T com no máximo k diferenças e um algoritmo $\theta(km)$ para calcular cada seqüência de operações de transformação ou alinhamento.

Mesmo para a variante em que o uso de espaço é $\theta(n)$, o algoritmo de Landau e Vishkin usa um fator multiplicador de n que é grande, em parte por causa do uso de árvores de sufixos para calcular os índices que permitem acelerar a computação da tabela de programação dinâmica. Árvores de sufixos são estruturas de dados que formam um índice de todos os sufixos de um texto ou palavra, e que podem ser construídas em espaço e tempo linear. Apesar da complexidade linear no uso de espaço, mostramos que é possível diminuí-lo utilizando uma estrutura de dados mais simples que também é um índice de todos os sufixos de uma palavra, chamada de arranjo de sufixos.

Propusemos e implementamos uma variação do algoritmo de Landau e Vishkin que usa arranjos de sufixos para esse cálculo, melhorando o uso de espaço e mantendo um desempenho similar ao do algoritmo original (em média 25% mais lento que a versão original para cadeias de DNA quando a memória do computador é suficiente para toda a execução do algoritmo) mas que diminui o uso de espaço. Para o caso de DNA e RNA, podemos obter ganhos de 26 a 29% com respeito ao uso de espaço, o que possibilita pro-

cessar palavras que não seriam processadas com o algoritmo original baseado em árvores de sufixos.

O algoritmo proposto foi inicialmente descrito em (5) como um resumo estendido, e posteriormente publicado como artigo completo. Neste trabalho expandimos essa descrição e apresentamos os ajustes necessários para realizar a computação do *LCE* (*Longest Common Extension*, definida na seção 2.1) por meio de arranjos de sufixos utilizando árvores cartesianas, e a descrição da implementação e experimentos. [1]

Desenvolvemos um mecanismo para consultas em tempo constante do comprimento do maior prefixo comum de dois sufixos quaisquer de uma palavra, o que torna possível substituir o uso de árvores de sufixos por arranjos de sufixos em aplicações que precisem desse cálculo, como é o caso do algoritmo de Landau e Vishkin.

A modificação realizada não é trivial. Foi preciso desenvolver a forma de calcular o comprimento do maior prefixo comum de dois sufixos de uma palavra em tempo constante após processamento linear de um arranjo de sufixos, diminuindo o uso de espaço. Isso foi possível com a construção de uma árvore cartesiana que é processada para consultas de *LCA* (*Lowest Common Ancestor*, definida na seção 2.1) em tempo constante. Apesar do aparente aumento da quantidade de informação (inserimos uma quantidade maior de estruturas de dados e etapas de processamento) na prática foi possível manter a ordem de complexidade de execução e diminuir efetivamente o espaço utilizado.

O restante deste trabalho está dividido da seguinte forma.

- No capítulo 2 definimos o problema e apresentamos conceitos teóricos que serão utilizados ao longo do trabalho.
- No capítulo 3 apresentamos o algoritmo de Landau e Vishkin e as técnicas para sua implementação.
- No capítulo 4 apresentamos nossas modificações no algoritmo de Landau e Vishkin.
- No capítulo 5 apresentamos a implementação realizada e a análise dos dados experimentais obtidos
- No capítulo 6 concluímos o trabalho e apresentamos algumas direções futuras.

[1] A implementação estará disponível a partir da página `http://www.mat.unb.br/~ayala/TCgroup/`

2 Definição do Problema

2.1 Preliminares

Neste trabalho usaremos os termos *palavra*, *cadeia de caracteres*, *seqüência*, *texto* e *padrão* como equivalentes à palavra inglesa *string* que em computação é a palavra mais comum para descrever uma seqüência ordenada de caracteres. Os nomes *texto* e *padrão* servirão para identificar o papel específico de cada palavra no contexto que estiver sendo descrito. Representaremos palavras com letras maiúsculas como P e T, e caracteres como p, t em letras minúsculas, sendo que o caractere t_i será o i-ésimo caractere da palavra T. As formas caligráficas $\mathcal{B}$, $\mathcal{T}$ ou $\mathcal{C}$ serão usadas para nomear árvores.

2.1.1 Palavras

Dadas as *palavras* $T = t_1...t_n$ e $P = p_1...p_m$ de comprimento $|T| = n$ e $|P| = m$, $m \leq n$, sobre um alfabeto Σ apresentamos as definições:

- ε é a *palavra* vazia, e $|\varepsilon| = 0$.
- P é uma *subpalavra* de T se $m \leq n$ e $p_1...p_m = t_i...t_{i+m-1}$ para algum $i \geq 1$ e $i + m - 1 \leq n$. Se $m < n$ dizemos que P é uma *subpalavra própria* de T.
- P é um *prefixo* de T se $m \leq n$ e $p_i = t_i$ para $1 \leq i \leq m$. Se $m < n$ então dizemos que P é um *prefixo próprio* de T.
- P é um *sufixo* de T se $p_1...p_m = t_i...t_{i+m-1}$ para $i + m - 1 = n$ e $i \geq 1$. Se $i > 1$ então dizemos que P é um *sufixo próprio* de T. Também dizemos que $T_i = t_i...t_n$ onde $i \geq 1$ é o *i-ésimo sufixo* de T (ou seja, o sufixo de T que começa na posição i).
- O *maior prefixo comum* ou *LCP* (*Longest Common Prefix*) de T e P é a maior *palavra* $L = l_1...l_k$ tal que $0 \leq k \leq m$ e $l_1 \ldots l_k = p_1 \ldots p_k = t_1 \ldots t_k$. Se $k = 0$ então $L = \varepsilon$.

- Além disso usamos a notação $LCP_{P,T}$ para indicar o LCP de P e T. Indicamos ainda $LCP_{P,T}(i,j)$ como sendo o maior prefixo comum de P_i e T_j. Quando P e T forem óbvios pelo contexto, usaremos apenas $LCP(i,j)$.

- A *maior extensão comum* ou LCE (*Longest Common Extension*) de T e P é o comprimento do maior prefixo comum de P e T: $|LCP_{P,T}|$. Usaremos a notação $LCE_{P,T}$ para indicar o LCE de P e T, e $LCE_{P,T}(i,j)$ como sendo o comprimento de $LCP_{P,T}(i,j)$. Quando P e T forem óbvios pelo contexto, usaremos apenas $LCE(i,j)$.

Para efeitos dos algoritmos apresentados neste trabalho, chamaremos as *palavras* T e P de *texto* e *padrão*, respectivamente.

2.1.2 Árvores e o LCA

Uma *árvore* $\mathcal{T}$ é um conjunto de nós (ou vértices) e arestas que possui as seguintes propriedades:

- uma aresta liga exatamente dois nós.

- Um *caminho* em $\mathcal{T}$ é uma lista de nós distintos tal que dois nós em seqüência são unidos por uma aresta, e nenhuma aresta se repete. O comprimento de um caminho é o número de nós presentes nesse caminho.

- Existe exatamente um caminho entre dois nós quaisquer de $\mathcal{T}$.

Uma outra forma de descrever uma árvore é como um grafo conectado sem ciclos.

Podemos designar um nó de $\mathcal{T}$ como sua raiz, o que transforma a árvore em uma estrutura hierárquica. A definição de árvores apresentada por Knuth em (6) explicita essa estrutura hierárquica.

Para efeitos deste trabalho, todas as árvores possuem uma raiz. Assim dizemos que para um nó v qualquer, os nós no caminho entre v e a raiz são os nós *ancestrais* de v (note que v é ancestral de si mesmo). Um *ancestral próprio* de v é um ancestral de v que não é o próprio v.

Para todo nó v, exceto a raiz, temos exatamente um ancestral próprio w que está ligado a v por uma aresta. Dizemos que w é o nó *pai* de v, e que v é um nó *filho* de w. Dizemos que dois nós são irmãos se são filhos do mesmo nó.

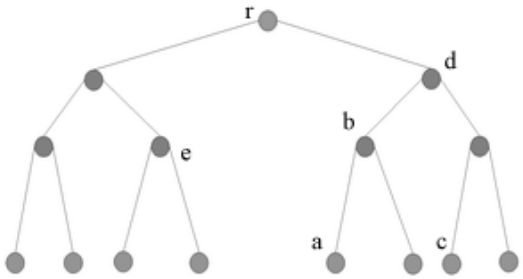


Figura 1: exemplo de LCA

A raiz da árvore não possui ancestrais próprios e portanto não possui nó pai. Além disso, a raiz é ancestral próprio de todos os demais nós da árvore.

Chamamos de *folha* um nó que não possui nós filhos, e de *nós internos* todos os demais nós.

Dizemos ainda que cada nó v da árvore $\mathcal{T}$ define uma *sub-árvore*, que é a árvore formada pelos nós dos quais v é ancestral, e as arestas que ligam esses nós. A raiz da sub-árvore de v é o nó v.

Uma *árvore binária* é uma árvore em que cada nó possui no máximo 2 filhos.

Uma *árvore binária completa* é uma árvore binária em que todos seus nós que não são folhas possuem exatamente 2 filhos, e além disso o comprimento dos caminhos da raiz até cada folha é o mesmo. Uma árvore binária completa com p folhas terá exatamente $2p - 1$ nós, e p será da forma $p = 2^x$ para algum $x \in \mathbf{N}$.

Definição 2.1.1 (LCA)**.** *Dada a árvore $\mathcal{T}$, o* LCA *(*Lowest Common Ancestor*) de dois nós v e w de $\mathcal{T}$ é o nó x que é ancestral de v e w tal que nenhum outro nó na sub-árvore de x também seja ancestral de v e w.*

Dados os nós v e w da árvore $\mathcal{T}$, indicaremos o LCA de v e w por $LCA_{\mathcal{T}}(v, w)$. Onde for óbvio qual a árvore sendo referenciada, usaremos a notação $LCA(v, w)$.

Exemplo 2.1.1 (LCA)**.** *Na figura 1 temos uma árvore binária completa, e alguns nós rotulados. Da definição de LCA, temos que:*

- $LCA(a, b) = b$.
- $LCA(a, c) = LCA(b, c) = d$.
- $LCA(d, e) = LCA(c, e) = LCA(b, e) = LCA(a, e) = r$.

- $LCA(a, d) = LCA(b, d) = d$.

2.2 Distância de Edição

O conceito de distância de edição leva ao problema de que tratamos neste trabalho.

Definição 2.2.1 (Distância de Edição)**.** *A distância de edição entre duas* palavras $P = p_1...p_m$ *e* $T = t_1...t_n$ *é a quantidade mínima de operações necessárias para transformar* P *em* T *ou vice-versa, onde as operações possíveis são:*

- *Substituição* quando o caractere p_i de P é substituído por um caractere t_j de T.
- *Inserção* quando um caractere p_i de P é inserido na posição j de T.
- *Remoção* quando o caractere p_i é removido de P.

Quando o $p_i = t_j$ dizemos que é um *casamento* ou *pareamento* (*match*).

A distância de edição entre as palavras P e T é uma medida do grau de diferença entre elas, mas além da distância de edição nos interessa saber a seqüência de operações que transformariam uma dessas palavras na outra, que chamamos de *transcrito de edição.*

Para ilustrar os conceitos de distância e transcrito de edição apresentamos o exemplo 2.2.1 que segue:

Exemplo 2.2.1 (Transcrito de edição)**.** *Dadas as palavras* P=`TGCCATA` *e* T=`ATCCCTGAT`*, o transcrito de edição de* P *em* T *é a seqüência de operações:*

1. *Inserimos o caractere "A" na posição 1:* `ATGCCATA`*.*
2. *Substituímos o caractere na posição 3 ("G") por "C":* `ATCCCATA`*.*
2. *Removemos o caractere na posição 6 ("A"):* `ATCCCTA`*.*
4. *Inserimos o caractere "G" na posição 7:* `ATCCCTGA`*.*
5. *Inserimos o caractere "T" após a posição 8:* `ATCCCTGAT` $= T$*.*

Observe que com 5 operações podemos transformar P *em* T*, e vice-versa. Em especial, o cálculo da a distância de edição apresentado a seguir indica que esse é o número mínimo de operações para* P *e* T *dados (baseado na tabela de programação dinâmica apresentada na figura 2).*

A distância de edição entre T e P pode ser encontrada por meio de uma relação de recorrência sobre as distâncias de edição de prefixos de T e P, ou seja, a distância de edição $D(i,j)$ entre $p_1...p_i$ e $t_1...t_j$ será calculada das distâncias $D(i-1,j-1)$ entre $p_1...p_{i-1}$ e $t_1...t_{j-1}$, $D(i-1,j)$ entre $p_1...p_{i-1}$ e $t_1...t_j$ e $D(i,j-1)$ entre $p_1...p_i$ e $t_1...t_{j-1}$ utilizando a relação de recorrência:

$$D(i,j) = \begin{cases} i+j & \text{se } j=0 \text{ ou } i=0, \\ \mathit{mínimo} \begin{cases} D(i-1,j-1)+d \\ D(i-1,j)+1 \\ D(i,j-1)+1 \end{cases} & \text{caso contrário} \\ \text{onde } d=0 \text{ se } p_i = t_j \text{ ou } 1 \text{ se } p_i \neq t_j \end{cases}$$

Essa relação de recorrência pode ser calculada por um algoritmo de programação dinâmica de complexidade $\theta(nm)$ que utiliza uma tabela de programação dinâmica de tamanho $(n+1) \times (m+1)$, também denotada por D. Os alinhamentos ou transcritos de edição podem ser recuperados utilizando ponteiros de retorno (*traceback*) que formam um caminho na tabela de programação dinâmica.

Para recuperar a seqüência de operações que formam o transcrito de edição basta seguir os ponteiros de retorno a partir da célula $D(m,n)$.

Na célula $D(i,j)$ da tabela de programação dinâmica um ponteiro para a célula superior significa que removemos o caractere p_i de P, um ponteiro para a célula à esquerda significa que inserimos o caractere t_j em P, e um ponteiro na diagonal significa um casamento de p_i e t_k ou então uma substituição se $p_i \neq t_j$. Seguimos os ponteiros de retorno até chegarmos a uma célula $D(0,k)$ ou $D(k,0)$. Nesse ponto removemos os k primeiros caracteres do padrão ou do texto, respectivamente.

O algoritmo de programação dinâmica consiste em construir uma tabela para os valores $D(i,j)$ conforme a relação de recorrência acima tal que o valor da célula $D(i,j)$ será o valor $D(i,j)$ na relação de recorrência. Na figura 2 temos um exemplo das tabelas de distância de edição e de ponteiros de retorno para as palavras P=TGCCATA e T=ATCCCTGAT utilizadas no exemplo 2.2.1. Cada célula $D(i,j)$ da tabela possui o valor da distância de edição de $p_1 \ldots p_i$ e $t_1 \ldots t_j$ ($i=0$ ou $j=0$ são condições iniciais para o cálculo). O ponteiro de retorno indica qual célula na tabela foi escolhida para calcular o valor da célula corrente.

		T	A	T	C	C	C	T	G	A	T
	P	0	1	2	3	4	5	6	7	8	9
T	1	1	1	2	3	4	5	6	7	8
G	2	2	2	2	3	4	5	5	6	7
C	3	3	3	2	2	3	4	5	6	7
C	4	4	4	3	2	2	3	4	5	6
A	5	4	5	4	3	3	3	4	4	5
T	6	5	4	5	4	4	3	4	5	4
A	7	6	5	5	5	5	4	4	4	5

		T	A	T	C	C	C	T	G	A	T
	P	0	1	2	3	4	5	6	7	8	9
T	1		↖								
G	2			↖							
C	3				↖						
C	4					↖					
A	5					↑					
T	6						↖	←			
A	7								↖	←	

Figura 2: Tabela de programação dinâmica e ponteiros de retorno

Uma técnica descrita por Hirschberg(3) em 1975 permite que o cálculo da relação de recorrência seja feito usando espaço $\theta(n)$ ao invés de $\theta(nm)$, dobrando o tempo de execução. Essa técnica consiste em encontrar um ponto no alinhamento ótimo entre P e T. Esse ponto ótimo divide a tabela de programação dinâmica em dois subproblemas que juntos têm a metade do tamanho do problema original. Então esses subproblemas são resolvidos recursivamente.

Exemplo 2.2.2 (Tabela de programação dinâmica e ponteiros de retorno)**.** *Na figura 2 temos a tabela de programação dinâmica calculada para o exemplo 2.2.1, e a mesma tabela com os ponteiros de retorno desenhados, indicando quais as operações que devem ser executadas para transformar P em T.*

O algoritmo de alinhamento global de Needleman e Wunsch usa uma relação de recorrência semelhante à de distância de edição. Seja $\Sigma' = \Sigma \cup \{-\}$ onde o caractere "–" representa um espaço. Então dada uma função de pontuação $\delta : \Sigma' \times \Sigma' \to \Re$, a *similaridade* S entre duas seqüências P e T é dada pela relação de recorrência:

$$S(i,j) = \begin{cases} 0 & \text{se } j = 0 \text{ e } i = 0, \\ S(i-1,0) + \delta(p_i, -) & \text{se } j = 0 \text{ e } 1 \le i \le m, \\ S(0,j-1) + \delta(-, t_j) & \text{se } i = 0 \text{ e } 1 \le j \le n, \\ \textit{máximo} \begin{cases} S(i-1,j-1) + \delta(p_i, t_j) \\ S(i-1,j) + \delta(-, t_j) \\ S(i,j-1) + \delta(p_i, -) \end{cases} & \text{caso contrário} \end{cases}$$

Essa relação de recorrência pode ser calculada por um algoritmo de programação dinâmica de complexidade $\theta(mn)$ similar ao algoritmo para a distância de edição, usando uma tabela de programação dinâmica que aqui também será denotada por S.

A função de pontuação δ indica o quão provável ou aceitável supomos uma substituição específica ou um espaço. Por exemplo, se $\delta(e,i) = 2$ e $\delta(e,a) = -1$ isso quer dizer que uma substituição de e por i é mais aceitável no cálculo que estamos realizando que uma substituição de e por a. Para o alinhamento de seqüências moleculares δ costuma indicar a probabilidade da ocorrência de uma mutação que transforme uma base ou aminoácido em outro (7).

O algoritmo de Needleman e Wunsch procura a *máxima* similaridade entre duas seqüências, enquanto o algoritmo de distância de edição busca a *mínima* diferença. Além disso o algoritmo de Needleman e Wunsch usa uma função de pontuação que pode atribuir pontuações distintas a pares distintos de caracteres. Podemos dizer que o algoritmo de Needleman e Wunsch resolve um problema mais genérico que inclui o problema de distância de edição. Dada a função δ apropriada podemos usar o cálculo de similaridade para calcular a distância de edição entre duas palavras P e T. Observe que se fizermos $\delta(a,b) = \{0 \text{ se } a = b, -1 \text{ caso contrário}\}$, obteremos $S(m,n)$ tal que $D(m,n) = -S(m,n)$.

Como o algoritmo de Needleman e Wunsch não usa o conceito de edição da palavra, ao invés do *transcrito de edição* usamos o conceito de *alinhamento* para representar as semelhanças entre as seqüências.

Definição 2.2.2 (Alinhamento)**.** *Um* alinhamento *de duas seqüências P e T é uma matriz $2 \times l$ ($l \geq m, n$) tal que a primeira linha da matriz contém os caracteres de P na ordem em que aparecem em P mesclados com $l - m$ espaços (representados por "–") e a segunda linha contém os caracteres de T na ordem em que aparecem em T mesclados com $l - n$ espaços tal que nenhuma coluna da matriz possui espaços em ambas as linhas (8).*

Na representação visual de um alinhamento podemos desenhar uma linha extra entre as linhas do alinhamento com um caractere "|" sempre que os caracteres c_1 e c_2 de uma coluna $\begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$ do alinhamento forem iguais, e em branco caso contrário. Isso permite visualizar melhor as situações em que foi feita uma operação de substituição. Usaremos o conceito de alinhamento ao invés de transcrito de edição, pois é um conceito mais genérico e mais cômodo.

O alinhamento pode ser recuperado seguindo os ponteiros de retorno de forma similar ao transcrito de edição. Na célula $S(i,j)$ da tabela de programação dinâmica, um ponteiro para a célula superior significa um espaço na segunda linha da coluna, um ponteiro para a célula à esquerda significa um espaço na primeira linha da coluna, e um ponteiro na diagonal significa uma coluna sem espaços com p_i na primeira linha e t_j na segunda.

Podemos também construir um alinhamento de duas palavras P e T a partir do transcrito de edição de P em T percorrendo os caracteres de P e T na ordem em que aparecem da seguinte forma:

- Suponha que já temos as k primeiras colunas do alinhamento, e a próxima coluna será a coluna $k+1$ e vamos processar os caracteres p_i e t_j tal que a célula $D(i,j)$ se encontra no caminho formado pelos ponteiros de retorno. Se $k = 0$ então $i = j = 1$.
  - Se $p_i = t_j$ então acrescentamos a coluna $C_{k+1} = \begin{pmatrix} p_i \\ t_j \end{pmatrix}$ e fazemos $i = i+1$, $j = j+1$ e $k = k+1$.
  - Se fazemos a substituição de p_i por t_j, então acrescentamos a coluna $C_{k+1} = \begin{pmatrix} p_i \\ t_j \end{pmatrix}$ e fazemos $i = i+1$, $j = j+1$ e $k = k+1$.
  - Se removemos o caractere p_i então acrescentamos a coluna $C_{k+1} = \begin{pmatrix} p_i \\ - \end{pmatrix}$ e fazemos $i = i+1$ e $k = k+1$.
  - Se inserimos o caractere t_j então acrescentamos a coluna $C_{k+1} = \begin{pmatrix} - \\ t_j \end{pmatrix}$ e fazemos $j = j+1$ e $k = k+1$.
- Repetimos as operações até que todos os caracteres de P e T estejam no alinhamento, e todas as operações do transcrito de edição tenham sido representadas.

Exemplo 2.2.3 (Alinhamento)**.** *Dadas P e T como no exemplo 2.2.1 e o transcrito de edição gerado de P para T, o alinhamento correspondente seria:*

```
P:   - T G C C A T - A -
       |   | |   |   |
T:   A T C C C - T G A T
```

2.3 Pesquisa aproximada de padrões com k erros

O problema da pesquisa aproximada de padrões com k erros entre um padrão P e um texto T é o problema de encontrar todos os pares de posições (i,j) em T tal que a

distância de edição entre P e $t_i...t_j$ é no máximo k. O caso especial em que $k = 0$ é o problema de encontrar todas as ocorrências de P em T.

Esse problema pode ser resolvido pelo algoritmo de programação dinâmica usado para o problema da distância de edição com uma alteração simples: na condição inicial definimos que $D(i, 0) = 0$ para cada $0 \leq i \leq n$. As ocorrências de P em T serão os caminhos que iniciem na linha 0 e terminem na linha m da tabela de programação dinâmica. Um alinhamento gerado a partir desses caminhos é chamado de *alinhamento semi-global*

3 Algoritmo de Landau e Vishkin

Landau e Vishkin(4) desenvolveram um algoritmo de complexidade $\theta(kn)$ para o problema da pesquisa aproximada de padrões com k erros, melhorando dessa forma a complexidade da solução de programação dinâmica $\theta(nm)$, onde n e m são os comprimentos do texto e do padrão, respectivamente. O algoritmo é dividido em duas fases: uma fase de pré-processamento e uma fase de iteração. A apresentação mostrada aqui segue a de Gusfield(9).

Na fase de pré-processamento o algoritmo de Landau e Vishkin constrói uma árvore de sufixos $\mathcal{T}$ para as palavras P e T (concatenadas) e processa $\mathcal{T}$ para que seja possível calcular o LCA (seção 2.1.2) de quaisquer de suas folhas em tempo $O(1)$. Na sua fase de iteração o algoritmo de Landau e Vishkin usa a observação de que ocorrências do padrão no texto serão representados por caminhos ao longo das diagonais da tabela de programação dinâmica (representando casamentos) intercalados com trechos na vertical, horizontal e diagonais que representem erros. Assim o algoritmo percorre as diagonais da tabela de programação dinâmica fazendo saltos em tempo constante ao longo das diagonais, e o comprimento de cada salto é calculado a partir do LCA na árvore de sufixos de suas folhas correspondentes aos sufixos envolvidos de P e T.

Apesar das suas propriedades teóricas, não temos conhecimento de uma aplicação prática do algoritmo de Landau e Vishkin na análise de seqüências biológicas.

3.1 Fase de Pré-processamento

Na fase de pré-processamento construímos uma árvore de sufixos $\mathcal{T}$ para P concatenada a T e a processamos para que possamos calcular o LCA (ver seção 2.1.2) de duas folhas quaisquer em $O(1)$.

Esse processamento faz um mapeamento dos nós de $\mathcal{T}$ para os nós de uma árvore binária completa $\mathcal{B}$ implícita, para a qual calculamos facilmente o LCA de dois nós em

tempo $O(1)$, dessa forma obtendo um cálculo de LCA em tempo $O(1)$ para uma árvore qualquer.

3.1.1 Árvores de sufixos

A *árvore de sufixos* é uma estrutura de dados desenvolvida por Weiner(10) que forma um índice de todos os sufixos de uma palavra, permitindo consultas rápidas às suas subpalavras e a informações da sua estrutura. Weiner(10) e McCreight(11) mostraram como é possível construir uma árvore de sufixos usando espaço e tempo linear, o que tornou o uso da estrutura de dados mais prático, e Ukkonen(12) desenvolveu um algoritmo que constroi uma árvore de sufixos de forma incremental (*on-line*). Posteriormente Kurtz(13) mostrou que apesar de usar espaço da ordem $\theta(n)$ o fator multiplicador de n pode ser alto, e desenvolveu técnicas para buscar uma diminuição desse fator.

Uma árvore de sufixos $\mathcal{T}$ para a palavra $T = t_1...t_n$ sobre um alfabeto Σ é uma árvore que possui as seguintes propriedades:

- $\mathcal{T}$ possui exatamente n folhas, numeradas de 1 a n.
- Cada nó interno de $\mathcal{T}$, exceto possivelmente pela raiz, possui pelo menos dois nós filhos.
- Cada aresta $\mathcal{T}$ é rotulada por uma subpalavra de T, tal que para um nó v os rótulos das arestas que ligam v a seus filhos se diferenciam pelo menos por seus caracteres iniciais.
- Para uma folha i de $\mathcal{T}$, a concatenação dos rótulos das arestas no caminho da raiz de $\mathcal{T}$ até i, na ordem em que são visitadas, é o sufixo T_i de T.
- A concatenação dos rótulos das arestas no caminho da raiz até o nó que é o $LCA_{\mathcal{T}}(v, v')$ de duas folhas v e v' de $\mathcal{T}$ nos dá o maior prefixo comum de T_v e $T_{v'}$.

Um caractere chamado *sentinela* que não pertence a Σ é concatenado a T para garantir que $\mathcal{T}$ possui exatamente $n + 1$ folhas. Denotamos os caracteres sentinelas como \$ e #, tal que \$ $\neq$ #. Na figura 3 apresentamos a árvore de sufixos $\mathcal{T}$ para a palavra $T = GATGACCA\$$.

A árvore de sufixos para uma palavra de comprimento n pode ser construída em tempo $\theta(n)$ utilizando espaço $\theta(n)$, como descrito por McCreight(11) e Ukkonen(12). O

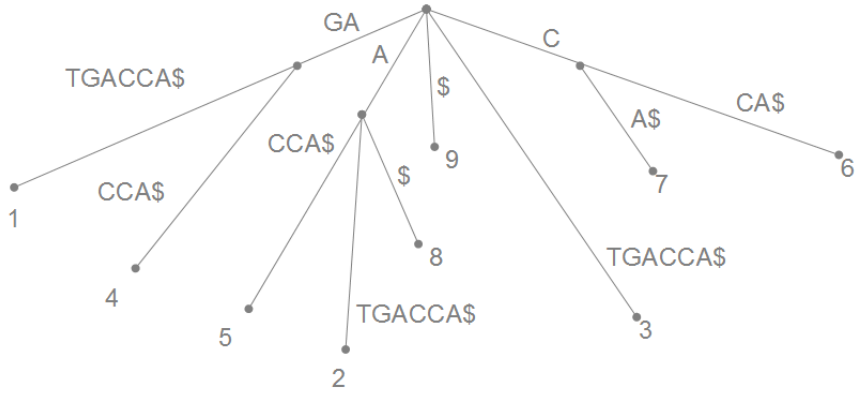


Figura 3: Árvore de sufixos para a palavra GATGACCA$

algoritmo de McCreight adiciona os sufixos de T à árvore $\mathcal{T}$ um após o outro, adicionando primeiro o sufixo T_1, seguido do sufixo T_2 e assim sucessivamente até que o todos os sufixos de T tenham sido adicionados a $\mathcal{T}$. O algoritmo de Ukkonen constrói a árvore de sufixos $\mathcal{T}$ adicionando os prefixos de T na ordem crescente de seu comprimento, construindo primeiro a árvore de sufixos $\mathcal{T}_1$ para a palavra t_1, depois adicionando o caractere t_2 para obter a árvore de sufixos $\mathcal{T}_2$ para a palavra t_1t_2, depois acrescentando t_3 para obter a árvore de sufixos $\mathcal{T}_3$ para a palavra $t_1t_2t_3$, e assim sucessivamente até que é construída a árvore $\mathcal{T}$ para $t_1 \ldots t_n$ a partir da árvore $\mathcal{T}_{n-1}$ para $t_1 \ldots t_{n-1}$. Por construir a árvore de sufixos processando um caractere de T por vez, na ordem em que aparecem na palavra, dizemos que o algoritmo de Ukkonen é um algoritmo *on-line*. Apesar dos algoritmos de McCreight e de Ukkonen parecerem algoritmos muito distintos, Kurtz e Giegerich(14) mostraram que são na verdade muito parecidos.

Para permitir a construção usando tempo e espaço $\theta(n)$ rotulamos cada aresta de $\mathcal{T}$ com o índice em T do caractere inicial de seu rótulo e com o comprimento do rótulo (ou então o índice do seu caractere final em T). Assim cada rótulo consome o espaço constante de 2 números inteiros para sua representação.

Para a construção em tempo $\theta(n)$ ambos os algoritmos usam *ponteiros de sufixos* (*sufix links*) para acelerar a atualização dos rótulos.

Seja $\bar{x}$ a palavra formada pela concatenação dos rótulos das arestas no caminho da raiz de $\mathcal{T}$ até o nó x, na ordem em que são visitados, note que se x é a raiz de $\mathcal{T}$ então $\bar{x} = \epsilon$. Então seja o nó x tal que $\bar{x} = \alpha\beta$ onde α seja um caractere e β uma palavra (possivelmente a palavra vazia ϵ). Então o ponteiro de sufixos desse nó apontará para o nó interno y tal que $\bar{y}$ é prefixo de β (se y existir).

A primeira utilidade da árvore de sufixos é a pesquisa exata de padrões. Dadas as palavras T e P, e árvore de sufixos $\mathfrak{T}$ construída a partir de T, a consulta para saber se P é subpalavra de T é $\theta(m)$. A consulta pode ser feita da seguinte forma:

1. Fazemos o nó x igual a raiz de $\mathfrak{T}$.
2. Se x é uma folha, então P não é subpalavra de T.
3. Selecionamos a aresta saindo de x para o nó y filho de x tal que o seu rótulo se inicia com o caractere p_1, se não existe essa aresta então P não é uma subpalavra de T.
4. Caso contrário seja $R = r_1...r_k$ o rótulo da aresta selecionada.
5. Comparamos r_2 com p_2, r_3 com p_3 e assim sucessivamente até que todo os caracteres do R ou de P tenham sido comparados, ou até que encontremos i ($1 < i \leq m$ e $1 < i \leq k$) tal que $r_i \neq p_i$
6. Se encontramos i descrito no passo anterior, então P não é subpalavra de T.
7. Se $m \leq k$ e não encontramos i no passo 5 acima, então P é uma subpalavra de T, e y é o nó mais próximo da raiz tal que P é subpalavra de $\bar{y}$.
8. Caso contrário, $m > k$, então fazemos $x = y$, $P = P_{k+1}$ e voltamos ao passo 2.

Encontrado P em $\bar{y}$, todas as folhas na sub-árvore de x representam os sufixos de T do qual P é um prefixo. Além disso, a quantidade de folhas na sub-árvore de y é a quantidade de vezes que P está em T.

Na prática, a implementação e construção eficiente de árvores de sufixos é complexa. Kurtz(13) mostra que apesar de uma árvore de sufixos usar espaço que é linear com respeito ao comprimento da palavra, muitas implementações são pouco eficientes no uso de espaço e possuem um fator multiplicador de n que é grande demais (maior que $24n$ bytes na maioria das implementações). No mesmo artigo Kurtz analisa em detalhe as informações necessárias para construir uma árvore de sufixos e descreve técnicas de implementação que melhoram o uso de espaço da árvore de sufixos de forma que o espaço utilizado seja de $10n$ a $12n$ bytes para palavras de até 128 milhões de caracteres.

Árvores de sufixos são muito utilizadas para aplicações em biologia computacional, como nos sistemas MUMMER (15) que faz alinhamento de seqüências biológicas, e RE-Puter (16) que encontra repetições em seqüências biológicas, ambos capazes de trabalhar

com massas de dados do tamanho de genomas inteiros. Para poder processar palavras desse porte, a implementação de árvore de sufixos do sistema MUMMER usa cerca de $15n$ bytes para cadeias de DNA.

3.1.2 Cálculo do LCA em $O(1)$ para uma árvore binária completa

O algoritmo de Landau e Vishkin usa um cálculo de LCA em uma árvore binária completa com complexidade $O(1)$ para calcular o LCA em uma árvore genérica em $O(1)$. Nesta seção apresentamos o cálculo para a árvore binária completa $\mathcal{B}$.

Seja $\mathcal{B}$ a árvore binária completa com p folhas (ver seção 2.1.2). Como $\mathcal{B}$ é completa, $\mathcal{B}$ possui $n = 2p - 1$ nós, e todas as suas folhas estão à mesma distância da raiz e cada sub-árvore dos nós filhos da raiz tem exatamente $p - 1$ nós. Rotulamos cada nó de $\mathcal{B}$ com um número binário de (d+1) bits, onde $d = \log_2 p$. Esse rótulo é obtido com uma visita *em-ordem* (17) de cada nó v de $\mathcal{B}$, numerando cada nó na medida em que esse seja visitado. Para efeitos de notação, vamos identificar um nó de $\mathcal{B}$ com seu rótulo.

Seja $h(v)$ a função que retorna a altura do nó v, ou seja o número de nós presentes no caminho v até as folhas da sub-árvore da qual é raiz. Calculamos $h(v)$ como sendo a posição (contada a partir do bit mais à direita) do bit igual a 1 menos significativo (mais à direita) do rótulo de v. Diremos que o bit i de v é o bit na posição i contada a partir do bit mais à direita de v.

Note que $\mathcal{B}$ possui p folhas e $2p - 1$ nós. Como $\mathcal{B}$ é completa então a sub-árvore de cada um dos filhos da raiz (se existirem) terá $\frac{p}{2}$ folhas e $p - 1$ nós. Seja v a raiz de $\mathcal{B}$. Se $p = 1$ $\mathcal{B}$ possui $2p-1 = 1$ nó e $h(v) = 1$. Se $p = 2$ folhas então v tem exatamente 2 filhos, $\mathcal{B}$ possui $2p - 1 = 3$ nós e $h(v) = 2$. Se $p = 4$ então $\mathcal{B}$ possui $2p - 1 = 7$ nós, e $h(v) = 3$. É fácil verificar que se v é a raiz de $\mathcal{B}$, então $\mathcal{B}$ terá $2^{h(v)} - 1$ nós, e $2^{h(v)-1}$ folhas. Como a numeração dos nós é feita *em-ordem*, então temos que se $i = h(v)$, numeramos os nós na sub-árvore do filho à esquerda de v com os valores do intervalo $1 \ldots 2^{i-1} - 1$, numeramos v com 2^{i-1}, e os nós na sub-árvore do filho à direita de v com os valores do intervalo $2^{i-1} + 1 \ldots 2^{i+1} - 1$. Note que $h(v)$ depende apenas de v, e que é o valor exatamente na metade do intervalo que define os valores dos rótulos na sua sub-árvore, e que o tamanho desse intervalo pode ser calculado de $h(v)$.

Assim, seja v a raiz de uma sub-árvore qualquer de $\mathcal{B}$. Então a sua sub-árvore possui $2^{h(v)} - 1$ nós, e seus rótulos serão os rótulos v_i onde $v - (2^{h(v)-1} - 1) \leq v_i \leq v + (2^{h(v)-1} - 1)$,

independentemente do valor de v e do tamanho de $\mathcal{B}$ (pois $\mathcal{B}$ é completa e os rótulos numerados *em-ordem*).

Podemos verificar se dados dois nós v e w, $v \neq w$, se um é ancestral do outro da seguinte forma:

- Seja $h_v = h(v)$ e $h_w = h(w)$. Se $h_v = h_w$ então nem v é ancestral de w e nem w é ancestral de v.
- Suponha que $h_v > h_w$, então para verificar se v é ancestral de w verificamos se $v - (2^{h_v - 1} - 1) \leq w \leq v + (2^{h_v - 1} - 1)$.
- Em caso afirmativo v é ancestral de w, caso contrário nem v é ancestral de w e nem w é ancestral de v.

Exemplo 3.1.1 (Verificação se um de dois nós é ancestral do outro)**.** *Tomemos como exemplo a árvore binária completa ilustrada na figura 4(b).*

Seja $v = 10$ e $w = 3$, $h_v = h(10) = 2$ e $h_w = h(3) = 1$. Como $h_v > h_w$, e verificamos que **não é verdade** *que $10 - (2^{2-1} - 1) \leq 3 \leq 10 + (2^{2-1} - 1)$, então v não é ancestral de v e nem w é ancestral de v.*

Agora seja $v = 6$ e $w = 4$, $h_v = h(6) = 2$ e $h_w = h(4) = 3$. Como $h_v < h_w$, e verificamos que $4 - (2^{3-1} - 1) \leq 6 \leq 4 + (2^{3-1} - 1)$, então w é ancestral de v.

Para encontrar o LCA x de dois nós v e w de $\mathcal{B}$ verificamos primeiro se v está na sub-árvore de w ou vice-versa. Se não for esse o caso, então seja o nó $x = LCA(v, w)$. Sem perda de generalidade, podemos dizer que x será um nó tal que v está na sub-árvore de seu filho à esquerda, e w na sub-árvore de seu filho à direita, ou seja $x - (2^{h(x)-1} - 1) \leq v < x$ e $x > w \leq x + (2^{h(x)-1} - 1)$, e $h(x) > h(v)$ e $h(x) > h(w)$.

Observe que uma maneira de encontrar x a partir de v e w é subir a árvore a partir de v ou w, até chegarmos num nó que é ancestral de v e de w ao mesmo tempo. Para isso precisamos primeiro descobrir como encontrar o nó v' que é pai de v manipulando o rótulo de v:

Proposição 3.1.1. *Seja $h(v)$ a altura do nó v. O nó v' ancestral de v será o nó cujo rótulo é igual ao rótulo de v com o bit $h(v) + 1$ alterado para 1, o bit $h(v)$ alterado para 0.*

Demonstração. Suponha que $h(v) = 1$. Então o último bit de v é 1. Como $\mathcal{B}$ é completa, $h(v') = 2$, e vale $v' - (2^{h(v')-1} - 1) \leq v \leq v' + (2^{h(v')-1} - 1)$, de onde obtemos que

$v = v' - 1$ ou $v = v' + 1$. Como $h(v) = 1$ então os últimos 2 bits de v podem ser 01 ou 11. Da numeração *em-ordem* sabemos que se $v < v'$ então os últimos 2 bits de v serão 01. Nesse caso $v = v' - 1 => v' = v + 1$, de forma que os últimos 2 bits de v' serão 10 e vale a proposição. Se $v > v'$ então os últimos 2 bits de v serão 11. Nesse caso $v = v' + 1 => v' = v - 1$, de forma que os últimos 2 bits de v' serão 10 e vale a proposição.

Se v' é ancestral de v, então $h(v') = h(v) + 1$, e $v' - (2^{h(v)} - 1) \leq v \leq v' + (2^{h(v)} - 1)$. Da definição de h sabemos que v' tem seus últimos $h(v)$ bits com valor 0, precedido por um bit 1. Resta mostrar que os bits à esquerda do bit $h(v')$ são iguais em v e v'. Suponha que exista um bit $b > h(v')$ em v á esquerda de h(v') que seja diferente do mesmo bit b em v'. Se esse bit for 1 em v e 0 em v', então temos que $v > v'$. Ora, sabemos que a sub-árvore a direita de v' tem $2^{h(v')-1} - 1$ nós, e o maior rótulo de um nó nessa subárvore será $v' + 2^{h(v')-1} - 1$. Como todos os bits à direita do bit $h(v')$ são zero, e o bit mais significante de $2^{h(v')-1}$ é o bit $h(v) = h(v') - 1$, então $v > v' + (2^{h(v')-1} - 1)$ e v não pode estar na sub-árvore de v' o que é uma contradição. De forma análoga, se o bit b for 0 em v e 1 em v', $v < v'$ e v estaria na sub-árvore à esquerda de v'. Mas temos então que $v < v' - (2^{h(v)} - 1)$ o que é uma contradição e temos que o bit $b > h(v')$ possui o mesmo valor em v e v'.

□

Como sabemos achar o pai de v a partir de v, é fácil calcular $x = LCA(v, w)$. Escolhemos o nó dentre v e w que tem a menor altura, e subimos na árvore até encontrarmos o seu ancestral que possui altura igual à do outro nó. Feito isso, subimos na árvore a partir de ambos os nós, um nível por vez, até que o ancestral encontrado subindo a árvore a partir de v e w seja o mesmo. Esse procedimento é correto porque cada nó possui exatamente um único nó pai (exceto a raiz, que não possui nenhum) e a raiz é ancestral de todos os nós.

Para verificar que não é necessário subir a árvore a partir de cada nó para chegar no LCA de v e w, observe que esse ancestral x de v e w tem no seu rótulo todos os bits à esquerda do bit $h(x)$ iguais aos bits correspondentes em v e w, e que exatamente nessa posição os bits em v e w serão diferentes, indicando que v está na sub-árvore à esquerda de x (pois $x - (2^{h(x)-1} - 1) \leq v < x$) e w está na sub-árvore à direita de x (pois $x > w \leq x + (2^{h(x)-1} - 1)$). Então basta descobrir essa posição $h(x)$ para podermos descobrir x a partir de v ou w. Como os bits em v e w à esquerda do bit $h(x)$ são iguais, e o bit $h(x)$ é diferente, basta fazer uma operação *OU-EXCLUSIVO* ou *XOR* dos rótulos de v e x e procurar a posição do primeiro bit com valor 1 a partir da esquerda (primeira

posição onde há divergência entre os bits de v e w). Isso vai nos dar a posição do bit $h(x)$. Pela definição de h sabemos que todos os bits à direita dessa posição terão valor 0. Além disso da proposição 3.1.1 temos que os bits à esquerda dessa posição serão iguais aos bits nessas posições em v e w, então basta pegar o rótulo de v ou de w, fazer o bit $h(x)$ igual 1 e completar com $h(x) - 1$ bits 0 à direita do bit $h(x)$, e com isso encontramos o x (rótulo do nó que é LCA de v e w).

Assim, rotulada a árvore, podemos encontrar o LCA x de dois nós v e w de $\mathcal{B}$ em $O(1)$ da seguinte forma:

i. Sejam $h_v = h(v)$ e $h_w = h(w)$

ii. Se $h_v < h_w$ então se $w - (2^{h_w - 1}) + 1 \leq v \leq w + (2^{h_w - 1}) - 1$ então $x = w$

iii. Senão se $v - (2^{h_v - 1} + 1) \leq w \leq v + (2^{h_v - 1} - 1)$ então $x = v$

iv. Se nem (*ii.*) e nem (*iii.*) valem, então calculamos x:

   - $x' = v$ XOR w
   - $k =$ posição do bit 1 mais à esquerda de x'
   - $x'' = v$ com seus bits deslocados $d + 1 - k$ bits para a direita
   - $x''' = x''$ com o seu bit menos significante alterado para 1
   - $x = x'''$ com seus bits deslocados $d + 1 - k$ para a esquerda.

Dessa forma calculamos o rótulo x do nó que é o LCA de v e w em $O(1)$.

Observação 3.1.1 (Operações bit-a-bit em tempo constante)**.** *Algumas das operações bit-a-bit que tratamos como constantes são na realidade $\theta(log_2 n)$. Na prática, fixamos n para essas operações como sendo o maior valor inteiro que a arquitetura da máquina suporta. Uma vez fixado n independente do tamanho da entrada para o algoritmo, a complexidade dessas operações bit-a-bit pode ser analisada como $O(1)$.*

As operações em questão são a pesquisa da posição do bit 1 mais significativo ou menos significativo de um número inteiro, que usam consultas em tempo $O(1)$ em uma tabela.

Exemplo 3.1.2 (Cálculo do LCA em uma árvore binária completa)**.** *Seja a $\mathcal{B}$ a árvore binária completa como na figura 4 (a). $\mathcal{B}$ possui $p = 8$ folhas, e $n = 2p - 1 = 15$ nós, $d = \log_2 p = 3$. Primeiramente visitamos todos os nós de $\mathcal{B}$* em-ordem *e rotulamos os nós*

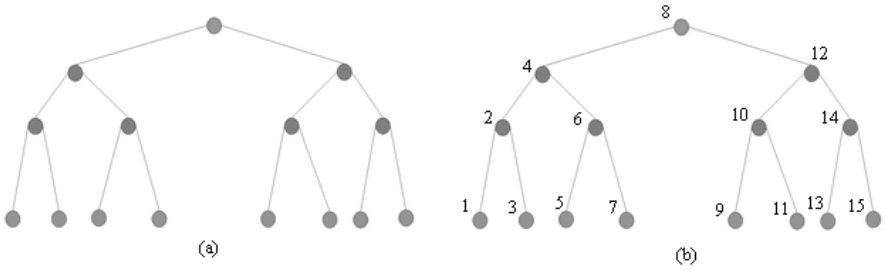


Figura 4: Cálculo em $O(1)$ de LCA em árvore binária completa

com um número de $d+1 = 4$ *bits na ordem em que são visitados, de forma que os rótulos ficam como na figura 4 (b).*

Dados dois nós v *e* w *de* $\mathcal{B}$*, temos dois casos distintos:*

- v *é ancestral de* w*, ou seja* $LCA_{\mathcal{B}}(v, w) = v$*.*
- $LCA_{\mathcal{B}}(v, w) = x$*, onde* $x \neq v$ *e* $x \neq w$*.*

No caso de w *ser ancestral de* v*, simplesmente chamamos* v *de* w *e* w *de* v*.*

Caso 1. Vamos escolher os nós $v = 4 = 0100$ *e* $w = 7 = 0111$*. Então* $h_v = h(0100) = 3$*, e* $h_w = h(0111) = 1$*. Como* $h_w < h_v$ *então fazemos* $k = 2^{h_v - 1} = 2^2 = 4$ *e verificamos que* $v - k + 1 \leq w \leq v + k - 1$*, logo o* $LCA_{\mathcal{B}}(v, w) = v = 4$*.*

Caso 2. Vamos escolher os nós $v = 9 = 1001$ *e* $w = 14 = 1110$*. Nesse caso* $h_v = h(1001) = 1$ *e* $h_w = h(1110) = 2$*. Como* $h_v < h_w$*, fazemos* $k = 2^{h_w - 1} = 2$ *e verificamos que* $w - k + 1 > v$ *e logo* $LCA_{\mathcal{B}}(v, w) = x \neq v \neq w$*. Para encontrar* x *então seguimos com o algoritmo:*

- $x' = v$ *XOR* $w = 1001$ *XOR* $1110 = 0111$*.*
- $k = 2$
- $x'' = 0010$ *(*1001 *com seus bits deslocados* $3 + 1 - 2 = 2$ *bits para a direita)*
- $x''' = 0011$ *(*0010 *com o seu bit menos significante alterado para 1)*
- $x = 1100$ *(*0011 *com seus bits deslocados* $3 + 1 - 2$ *para a esquerda)*

Logo, para o caso 2, temos corretamente que $LCA_{\mathcal{B}}(v, w) = 12$*.*

3.1.3 Pré-processamento para cálculo do LCA de uma árvore qualquer

Para calcular o LCA de duas folhas de uma árvore de sufixos $\mathcal{T}$ com complexidade $O(1)$ construimos um mapeamento de $\mathcal{T}$ para a árvore binária completa $\mathcal{B}$.

Visitamos cada nó de $\mathcal{T}$ numa visita em *pré-ordem* (17), rotulando cada nó com a ordem em que é visitado, e vamos identificar um nó com o seu rótulo.

Definimos uma função I como um mapeamento de nós da árvore genérica $\mathcal{T}$ na árvore binária completa $\mathcal{B}$, ou seja, dado o nó v de $\mathcal{T}$, $I(v) = w$, onde w é um nó de $\mathcal{B}$, e calculamos w da seguinte forma:

Se v é uma folha em $\mathcal{T}$, então $w = v$ (o rótulo do nó w em $\mathcal{B}$ será o rótulo v). Se v não for uma folha, então w será o rótulo de um nó v' na sub-árvore de v tal que para todo outro nó x na sub-árvore de v, $h(v') > h(x)$.

A definição de I é bem formada porque garante a unicidade de w na sub-árvore de v. Observe que a altura de um nó com rótulo w em $\mathcal{B}$ não depende do tamanho de $\mathcal{B}$, mas é determinada pela quantidade de zeros à direita do bit 1 menos significativo de w. Além disso observe que se existirem dois nós v' e v'' na sub-árvore de v tal que $h(v) = h(v')$, então teremos um nó w onde $v \leq w \leq v'$ tal que $h(w) > h(v)$.

A função I particiona $\mathcal{T}$ em conjuntos de nós que são mapeados para o mesmo nó $I(v)$ de $\mathcal{B}$. Dizemos que a *cabeça* de cada uma dessas partições é o nó v de $\mathcal{T}$ na partição que está mais próximo da raiz de $\mathcal{T}$.

O cálculo de I pode ser feito em tempo $O(n)$ visitando cada nó v de $\mathcal{T}$ em pré-ordem com o seguinte algoritmo recursivo:

Algoritmo 1 Cálculo do mapeamento $I(v)$

1. Se v é uma folha, então $I(v) = v$.
2. Caso Contrário, sejam $w_1 \ldots w_k$ os k filhos de v:
   2.1 Fazemos $I_i = I(w_i)$ para $1 \leq i \leq k$ (executando este algoritmo recursivamente para cada filho w_i de v).
   2.2. Seja $h_0 = h(v)$ e $h_i = h(I_i)$ para $1 \leq i \leq k$.
   2.3. Seja m o índice tal que h_m é o máximo de $h_0 \ldots h_k$.
   2.4. Se $m = 0$ então $I(v) = v$, senão $I(v) = I_m$

Além de I, precisamos de mais informação da estrutura de $\mathcal{T}$. Assim calculamos as funções L e A, onde $L(v)$ é a cabeça da partição $I(v)$ e $A(v)$ é o número binário tal que o i-ésimo bit de $A(v)$ é 1 se o nó v possui um ancestral u em $\mathcal{T}$ tal que $h(I(u)) = i$, e 0

caso contrário. Ou seja, L indica o nó mais próximo da raiz de $\mathcal{T}$ que é mapeado para o mesmo nó $I(v)$ em $\mathcal{B}$ e A codifica parte da estrutura hierárquica de $\mathcal{T}$ da raiz até o nó v.

L pode ser calculado juntamente com I. Para isso acrescentamos um passo 3 ao algoritmo 1 no qual fazemos $L(I(v)) = v$.

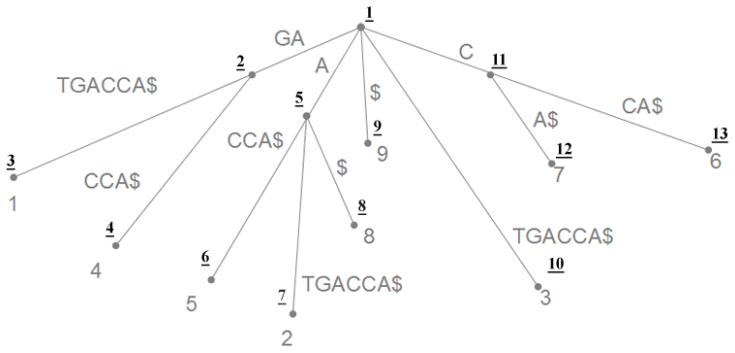
Tendo calculado I, calculamos A com uma visita em *pré-ordem* aos nós de $\mathcal{T}$, sendo que se v é nó filho de v', então $A(v)$ será o valor de $A(v')$ com o bit $h(I(v))$ com o valor 1.

Como cada uma das funções é calculada por um percurso em *pré-ordem* de $\mathcal{T}$ que percorre exatamente 2 vezes cada nó para rotular cada nó e calcular I e L, e 1 vez cada nó para calcular A, então a complexidade total é $\theta(n)$ para uma árvore com n nós.

Exemplo 3.1.3 (Mapeamento da árvore de sufixos $\mathfrak{T}$ para a árvore binária completa $\mathfrak{B}$)**.** *Seja a árvore de sufixos $\mathcal{T}$ mostrada na figura 5. Observe que a numeração da visita em* pré-ordem *de cada nó está representada por um número sublinhado próximo ao nó. Vamos mostrar o cálculo de I, A e L para a sub-árvore do nó com rótulo $v = 5$ (representado por $\underline{5}$ na figura).*

Mostramos primeiro o cálculo de I e L para a sub-árvore de v:

- *Primeiro calculamos $I(v_i)$ para cada filho de v_i, visitando cada um em* pré-ordem*:*
  - *Seja $v_1 = 6$. Como v_1 é uma folha então $I(6) = 6$, e fazemos $L(I(v_1)) = L(6) = 6$.*
  - *Seja $v_2 = 7$. Como v_2 é uma folha então $I(7) = 7$, e fazemos $L(I(v_2)) = L(7) = 7$.*
  - *Seja $v_3 = 8$. Como v_3 é uma folha então $I(8) = 8$, e fazemos $L(I(v_3)) = L(8) = 8$.*
  - *Como não há mais filhos de v, voltamos para v para calcular $I(v)$ e $L(v)$.*
- *Fazemos $h_0 = h(v) = h(5) = 1$, $h_1 = h(v_1) = h(6) = 2$, $h_2 = h(v_2) = h(7) = 1$ e $h_3 = h(v_3) = h(8) = 4$.*
- *Escolhemos i tal que h_i é máximo. No caso, $i = 3$ pois $h_3 = 4$ é a maior altura dentre os valores selecionados.*
- *Como $3 = i \neq 0$, fazemos $I(v) = I(v_i) = I(8) = 8$.*
- *Fazemos $L(I(v)) = L(8) = v = 5$.*



v	v (binário)	h(v)	l(v)	L(v)	A(v)
1	00001	1	8	1	1000
2	00010	2	4	2	1100
3	00011	1	3	3	1101
4	00100	3	4	4	1100
5	00101	1	8	1	1000
6	00110	2	6	6	1010
7	00111	1	7	7	1001
8	01000	4	8	1	1000
9	01001	1	9	9	1001
10	01010	2	10	10	1010
11	01011	1	12	11	1100
12	01100	3	12	11	1100
13	01101	1	13	13	1101

Figura 5: Mapeamento da árvore de sufixos $\mathcal{T}$ para a árvore binária completa $\mathcal{B}$

O cálculo para os demais nós se faz de maneira similar. Neste exemplo, em especial, observe que $I(1)$ *será 8.*

Calculados I *e* L*, vamos ilustrar agora o cálculo de* A *para a sub-árvore de* v*. Da figura, sabemos que o pai de* v *é a própria raiz da árvore, o nó 1. Note que* $I(1) = 8$ *e* $L(8) = 1$*.*

Vamos calcular os valores de A *para a raiz e a sub-árvore de* $v = 5$*.*

- *Como o nó 1 é a raiz da árvore (não possui pai), o valor de* $A(1)$ *será o número binário 0000 com o bit* $h(I(1))$ *alterado para o valor 1. Assim* $A(1) = \mathbf{1000} = 8$*.*
- *Como o nó 1 é o pai de* v*, Calculamos* $A(v) = A(5)$ *como sendo* $A(1)$ *com o bit* $h(I(v))$ *alterado para 1. Observe que temos* $I(v) = 8$ *e* $h(8) = 4$*, logo* $A(5) = \mathbf{1000} = 8$*.*
- *Agora calcular* $A(v_i)$ *para cada filho* v_i *de* v*:*
  - *Seja* $v_1 = 6$*.* $I(6) = 6$*, e* $h(6) = 2$*, fazemos* $A(6)$ *como sendo* $A(5)$ *com o bit 2 com valor 1. Assim,* $A(6) = \mathbf{1010} = 10$*.*
  - *Seja* $v_2 = 7$*.* $I(7) = 7$*, e* $h(7) = 1$*, fazemos* $A(7)$ *como sendo* $A(5)$ *com o bit 1 com valor 1. Assim,* $A(7) = \mathbf{1001} = 9$*.*
  - *Seja* $v_3 = 8$*.* $I(8) = 8$*, e* $h(8) = 4$*, fazemos* $A(8)$ *como sendo* $A(5)$ *com o bit 4 com valor 1. Assim,* $A(8) = \mathbf{1000} = 8$*.*

Dessa forma ilustramos o cálculo de I*,* L *e* A*. Na figura 5 mostramos as tabelas* I*,* L *e* A *completas para a árvore* $\mathcal{T}$*.*

3.1.4 Cálculo do LCA em O(1) em uma árvore de sufixos

Uma vez calculados I, L e A, podemos calcular o LCA de dois nós v e w de $\mathcal{T}$ em $O(1)$ usando a seguintes propriedade:

Propriedade 3.1.1. *Se z é ancestral de x em $\mathcal{T}$ então $I(z)$ é ancestral de $I(x)$ em $\mathcal{B}$.*

Assim, seja z o LCA de v e w em $\mathcal{T}$, e seja o $b = LCA_{\mathcal{B}}(I(v), I(w))$ o LCA de $I(v)$ e $I(w)$ em $\mathcal{B}$. Como $\mathcal{B}$ é uma árvore binária completa, b pode ser encontrado em $O(1)$ como vimos na subseção 3.1.2.

Podemos usar b para encontrar a altura de $I(z)$ a partir da informação da estrutura de $\mathcal{T}$ armazenda em A. Observe que $A(v)$ codifica a altura $h(I(v'))$ de cada ancestral v' de v. $A(w)$ armazena a mesma informação para w. Ora, $z = LCA_{\mathcal{T}}(v, w)$ por ser ancestral de v e de w terá a altura do seu mapeamento $j = h(I(z))$ mapeada em $A(v)$ e em $A(w)$, ou seja, o bit j terá o valor 1 em $A(v)$ e em $A(w)$. Para encontrar j observamos que os primeiros bons candidatos são as alturas mapeadas em ambos $A(v)$ e $A(w)$ (pois o bit j estará com o valor 1 em ambos). Além disso, pela propriedade 3.1.1 sabemos que a altura de j será no mínimo a altura de b. Então j será a posição do primeiro bit à esquerda do bit $i = h(b)$ que tem o valor 1 em $A(v)$ e $A(w)$.

Se $h(b) = i$, então podemos encontrar $j = h(I(z))$ em $O(1)$ da seguinte forma:

- Seja $x = A(v)$ AND $A(w)$ (observe que dessa forma deixamos com valor 1 apenas os bits que marcam as alturas comuns em $\mathcal{B}$ dos ancestrais de v e w).
- $x' = x$ com seus bits deslocados para a direita e de volta $i - 1$ bits (zeramos os bits à direita do (i)-ésimo bit).
- $j = h(x')$

Sabendo a altura do mapeamento de z, podemos encontrar os nós $\bar{v}$ e $\bar{w}$ que são ancestrais de v e w, respectivamente, que possuem o mesmo mapeamento que z. Para isso encontramos $I(x)$, onde x é um nó em uma partição cuja cabeça $L(I(x))$ é um nó filho de $\bar{v}$. Assim, ao encontrar $I(x)$ encontramos $\bar{v}$ facilmente. De forma análoga encontramos $\bar{w}$. Perceba que $\bar{v}$ é ancestral de v tal que $I(\bar{v}) = I(z)$ assim como $\bar{w}$ é ancestral de w tal que $I(\bar{w}) = I(z)$. Então pela numeração em pré-ordem, $z = min(\bar{v}, \bar{w})$.

Podemos encontrar $\bar{v}$ da seguinte forma:

- Seja $\bar{v}$ o nó ancestral de v que está na mesma partição de z (ou seja, $I(\bar{v}) = I(z)$):
  - se $h(I(v)) = h(I(z))$ então $\bar{v} = v$.
  - caso contrário $\bar{v} \neq v$ $(h(v) < j)$:
    - seja x o ancestral de v tal que $\bar{v}$ seja o nó pai de x. Então $h(I(x)) = k$ é a maior posição de um bit 1 em $A(v)$ que é menor que j.
    - encontramos $I(x) =$ bits de $I(v)$ à esquerda da posição k seguida de um bit 1 e completada com zeros.
    - O nó $\bar{v}$ será o nó pai de $L(I(x))$.
- calculamos $\bar{w}$ de forma similar a $\bar{v}$.
- se $\bar{v} < \bar{w}$ então o LCA de v e w é $\bar{v}$ caso contrário é $\bar{w}$.

Exemplo 3.1.4 (Cálculo do LCA em uma árvore de sufixos $\mathfrak{T}$)**.** *Seja a árvore de sufixos $\mathfrak{T}$ mostrada na figura 5. Sejam os nós $v = 6$ e $w = 8$. Vamos encontrar o $z = LCA(v, w)$.*

- *Em primeiro lugar fazemos $I_v = I(v) = 6$ e $I_w = I(w) = 8$, e calculamos $b = LCA_{\mathcal{B}}(I_v, I_w)$ como vimos na seção 3.1.2. Como $h(I_v) < h(I_w)$ verificamos que $8 - (2^{h(8)-1} - 1) \leq 6 \leq 8 + (2^{h(8)-1} - 1)$, de forma que o $LCA_{\mathcal{B}}(6, 8) = 8$.*
- *Fazemos $i = h(b) = h(8) = 4$.*
- *Encontramos em seguida $j = h(I(z))$ a partir de b. Calculamos $x = A(v)$ AND $A(w) =$ **1010** AND **1000** = **1000**, e zeramos os bits à direita do bit i e obtemos $x' =$ **1000**, e $j = h(x') = 4$.*
- *Vamos encontrar $\bar{v}$ e $\bar{w}$:*
  - *Observe que $h(v) < j$. Então seja x ancestral de v tal que o pai de x seja $\bar{v}$. Calculamos $k = h(I(x))$ como sendo a posição do bit 1 mais significante de $A(v)$ que está à direita do bit j. Como $A(v) =$ **1010** então $k = 2$.*
  - *Fazemos $I(x) = I(v)$ com o bit na posição k igual a 1 e os bits à direita da posição k iguais a 0. Como $I(v) =$ **0110** $= 6$, então $I(x) =$ **0110** $= 6$.*
  - *O nó $\bar{v}$ será o nó pai do nó $L(I(x))$. Como L(6)=6, $\bar{v} = 5$.*
  - *Como $h(w) = j$, então $\bar{w} = w = 8$.*
- *Como $\bar{v} < \bar{w}$, então $z = \bar{v} = 5$.*

Dessa forma ilustramos a busca do LCA de dois nós 6 e 8 de $\mathfrak{T}$, e encontramos o nó 5.

3.2 Fase de iteração

Na fase de iteração do algoritmo construímos caminhos seguindo as diagonais da tabela de programação dinâmica.

Na tabela de programação dinâmica D, supondo que o caractere t_j do texto rotula a coluna j da tabela, e o caractere p_i do padrão rotula a linha i, uma ocorrência de P em T forma um caminho sem ciclos que percorre a tabela iniciando na primeira linha e terminando na última linha.

Mais formalmente, dizemos que:

- Uma *diagonal* d da tabela de programação dinâmica D são todas as células $D(i, j)$ tal que $j - i = d$.
- A *diagonal principal* é a diagonal 0 de D composta pelas células $D(i, i)$ onde $0 \leq i \leq m \leq n$.
- Duas células $D(i, j)$ e $D(i', j')$ são ditas *adjacentes* se $i' \neq i$ ou $j' \neq j$ e além disso $i \leq i' \leq i + 1$ e $j \leq j' \leq j + 1$.
- Um *caminho* na tabela de programação dinâmica é uma seqüência de células $C_0 \ldots C_k$ onde para qualquer $k' \in \{0 \ldots k - 1\}$, $C_{k'}$ e $C_{k'+1}$ são adjacentes.
- Se a célula $C_{k+1} = D(i, j)$ é a célula que segue a célula $C_k = D(i - 1, j - 1)$ em um caminho, então dizemos que é um *descasamento* se $t_j \neq p_i$, e um *casamento* se $t_j = p_i$.
- Se a célula $C_{k+1} = D(i + 1, j)$ ou a célula $C_{k+1} = D(i, j + 1)$ segue a célula $C_k = D(i, j)$ em um caminho então dizemos que é um *espaço*.
- Um *erro* em um caminho é um descasamento ou um espaço.
- Um *d-caminho* em D é um caminho que se inicia ou na coluna 1 antes da linha $d+1$ ou na linha 1 e possui as propriedades:
  - Caminhos que iniciem na linha 1 começam com 0 erros e caminhos que iniciem na célula $C_0 = D(i, 1)$ para $1 \leq i \leq d$ iniciam com i erros.
  - Se a célula $C_k = D(i, j)$ está no caminho, então a célula $C_{k+1} = D(i', j')$ é a célula imediatamente após C_k no caminho (se existir) e $D(i', j') \in \{D(i+1, j+1), D(i, j + 1), D(i + 1, j)\}$.

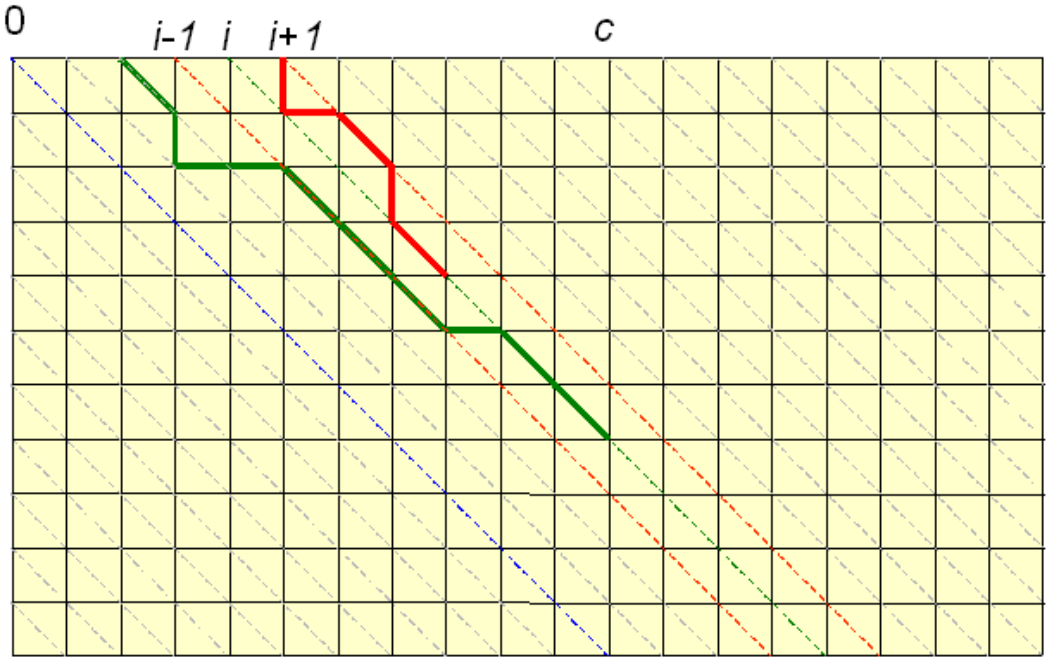


Figura 6: Caminho de maior alcance na diagonal i

- Possui exatamente d erros.
- Um d-caminho é o *de maior alcance* na diagonal i se é um d-caminho que termina em uma célula $C_k = D(r, c)$ na diagonal i e o índice c da coluna de C_k é maior ou igual ao índice da coluna da célula final de todos os outros d-caminhos que terminem na diagonal i.

Na figura 6 mostramos dois caminho que terminam na diagonal i. Na figura o caminho que se inicia na diagonal $k \leq i - 1$ é o caminho de maior alcance na diagonal i.

A fase de iteração do algoritmo de Landau e Vishkin constrói todos os 0-caminhos da tabela de programação dinâmica, e a partir desses todos os 1-caminhos, e a partir desses todos os 2-caminhos, e assim sucessivamente até que todos os d-caminhos procurados foram construídos. Isso é feito percorrendo cada diagonal i de D e extendendo os $(d-1)$-caminhos nas diagonais $i - 1$, i, e $i + 1$ para a diagonal i como será mostrado a seguir.

3.2.1 Construção e extensão de *d*-caminhos

Usaremos a notação $LCE(i, j)$ para indicar o $LCE_{P,T}(i, j)$ (ver seção 2.1).

Um d-caminho é construído em D da seguinte forma.

Se $d = 0$ então um 0-caminho que inicia na diagonal i é o caminho que começa na

célula $D(1, i)$ e é estendido na diagonal i até a célula $D(j, i + j)$, onde j é o LCE(1,i+1).

Um d-caminho começando na célula $D(d, 1)$, para $0 < d \leq m$ é estendido até a célula $D(d + j, j)$, onde j é o $LCE(d, 1)$.

Se $d > 0$ então um d-caminho é construído a partir de um $(d-1)$-caminho cuja célula final $C_k = D(r, c)$ está na diagonal i estendendo-o inicialmente para a célula $D(r', c')$ na diagonal i' de uma de três formas:

- o caminho é estendido uma célula para a direita para a célula $D_{r',c'} = D(r, c + 1)$ na diagonal $i' = i + 1$, significando a inserção de um espaço no padrão na posição r;
- o caminho é estendido uma célula para baixo para a célula $D_{r',c'} = D(r + 1, c)$ na diagonal $i' = i - 1$, significando a inserção de um espaço no texto na posição c;
- o caminho é estendido uma célula na diagonal $i' = i$ para a célula $D_{r',c'} = D(r + 1, c + 1)$, significando um descasamento entre t_c e p_r.

Após estender o $(d - 1)$-caminho para a célula (r', c') na diagonal i', o caminho é estendido l células na diagonal i' onde $l = LCE(r', c')$.

3.2.2 A iteração

O algoritmo de Landau e Vishkin, apresentado no algoritmo 2, percorre cada diagonal i da tabela de programação dinâmica, construindo os d-caminhos que são de maior alcance em cada diagonal, começando com todos os 0-caminhos, e depois desses calculando todos os 1-caminhos e assim sucessivamente até que todos os k-caminhos sejam encontrados. Os k'-caminhos (onde $0 \leq k' \leq k$) que terminem na linha m da tabela de programação dinâmica são ocorrências de P em T com no máximo k diferenças.

Em cada iteração calculamos o d-caminho de maior alcance na diagonal i da seguinte forma:

- Se $d = 0$ então o 0-caminho que inicia na diagonal i é o 0-caminho de maior alcance em i.
- Se $d > 0$, encontramos o d-caminho de maior alcance em i a partir dos $(d - 1)$-caminhos de maior alcance nas diagonais $i - 1$, i e $i + 1$ da seguinte forma:

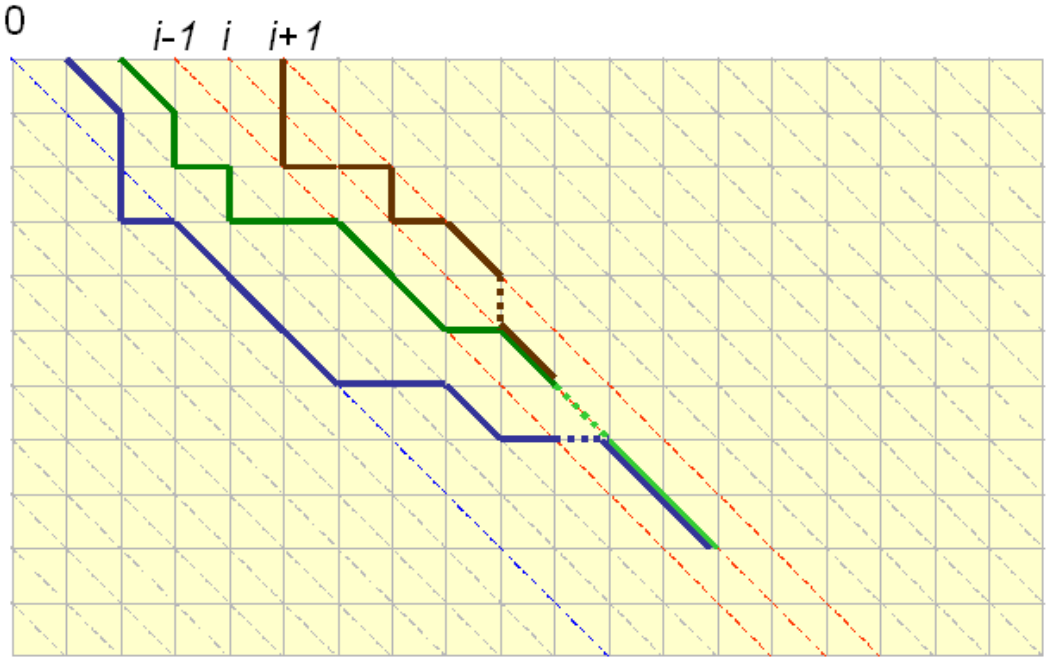


Figura 7: Extensão de caminhos para a diagonal i

- Estendemos o $(d-1)$-caminho de maior alcance na diagonal $i-1$ uma célula para a direita para a célula $D(i_h, j_h)$ na diagonal i e então o estendemos $l_h = LCE(i_h, j_h)$ células ao longo da diagonal i como descrito na subseção 3.2.1.
- De forma similar estendemos o $(d-1)$-caminho de maior alcance na diagonal $i+1$ uma célula para a baixo para a célula $D(i_v, j_v)$ na diagonal i e então o estendemos $l_v = LCE(i_v, j_v)$ células ao longo de i.
- Estendemos também o $(d-1)$-caminho de maior alcance na diagonal i uma célula ao longo da diagonal i para a célula $D(i_d, j_d)$, e então o estendemos $l_d = LCE(i_d, j_d)$ células ao longo de i.
- O d-caminho de maior alcance na diagonal i é escolhido dos três construídos acima como sendo aquele que tem o maior índice na coluna de sua célula final.

Na figura 7 ilustramos a extensão dos $d-1$ caminhos nas diagonais $i-1$, i e $i+1$ para a diagonal i com o acréscimo de 1 erro representado por uma linha pontilhada e a extensão ao longo da diagonal i.

Observe que uma vez que construímos uma arvore de sufixos $\mathcal{T}$ para a concatenação de P e T, todos os sufixos de P e de T serão folhas em $\mathcal{T}$. Assim, dados i e j correspondentes a P_i e T_j, respectivamente, o $LCA_{\mathcal{T}}$ das folhas correspondentes nos dá o $LCP(i,j)$, e portanto o $LCE(i,j)$. Dessa forma, o cálculo do LCE para a extensão dos d-caminhos

depende do cálculo do LCA na árvore de sufixos.

3.3 O algoritmo

No algoritmo 2 apresentamos as duas fases do algoritmo de Landau e Vishkin. Na primeira fase construímos uma árvore de sufixos generalizada $\mathcal{T}$ para P e T, e a processamos para que seja possível calcular o LCA de duas folhas quaisquer em $O(1)$. Em seguida construímos todos os 0-caminhos e executamos a iteração até encontrarmos todos os k-caminhos.

Algoritmo 2 Algoritmo de Landau e Vishkin para pesquisa aproximada de padrões

1. Construimos a árvore de sufixos generalizada $\mathcal{T}$ para P e T.
2. Processamos $\mathcal{T}$ em $O(n)$ de forma a responder consultas LCA em $O(1)$.
3. Para cada diagonal i da tabela de programação dinâmica, encontramos o seu 0-caminho de maior alcance com uma consulta de LCA entre P e T_{i+1}.
4. Para $d = 1 \ldots k$:
   4.1 Para cada diagonal i da tabela de programação dinâmica:
      4.1.1 Estendemos o $(d-1)$-caminho de maior alcance na diagonal $i-1$ uma célula para a direita para a célula $D(r,s)$ na diagonal i
      4.1.2 Estendemos esse caminho ao longo da diagonal i pela quantidade de células igual à profundidade do LCA dos sufixos correspondentes de P e T ($P[r]...P[m]$ e $T[s]..T[n]$).
      4.1.3 Estendemos o $(d-1)$-caminho de maior alcance na diagonal $i+1$ uma célula para a baixo para a célula $D(r',s')$ na diagonal i
      4.1.4 Estendemos esse caminho ao longo da diagonal i pela quantidade de células igual à profundidade do LCA dos sufixos correspondentes de P e T ($P[r']...P[m]$ e $T[s']..T[n]$).
      4.1.5 Estendemos o $(d-1)$-caminho de maior alcance na diagonal i uma célula ao longo da diagonal i para a célula $D(r'',s'')$
      4.1.6 Estendemos esse caminho ao longo da diagonal i pela quantidade de células igual à profundidade do LCA dos sufixos correspondentes de P e T ($P[r'']...P[m]$ e $T[s'']..T[n]$).
      4.1.7 Escolhemos o d-caminho de maior alcance dentre os três.
5. Cada caminho que alcançar a linha m é uma ocorrência de P em T com no máximo k erros.

3.4 Análise do algoritmo

Vamos analisar cada fase em separado.

3.4.1 A fase de pré-processamento

A fase de pré-processamento possui duas etapas distintas.

i. construímos a árvore de sufixos generalizada $\mathcal{T}$ para P e T em $\theta(n)$.

ii. pré-processamos $\mathcal{T}$ em $\theta(n)$ para o cálculo do LCA em $O(1)$

Sabemos que a árvore de sufixos pode ser construída com complexidade de espaço e tempo linear, então a etapa (i) é executada em $\theta(n+m) = \theta(n)$ (pois $n \geq m$).

A segunda etapa é executada com complexidade de tempo e espaço linear sobre o número de nós da árvore. Como o número de nós da árvore de sufixos é $\theta(n)$, então a segunda etapa também é executada em $\theta(n)$, de forma que a complexidade geral da fase de pré-processamento é $\theta(n)$.

3.4.2 A fase de iteração

A fase de iteração é composta de $k+1$ passos onde percorremos $n+k$ diagonais. Como o cálculo do LCA de duas folhas da árvore de sufixos é $O(1)$ após a fase de pré-processamento, então para cada diagonal fazemos 3 operações de extensão de caminho em tempo $O(1)$, de forma que a complexidade total do algoritmo é $\theta(kn)$.

A complexidade de espaço também é $\theta(kn)$ se for necessário apresentar os alinhamentos das ocorrências de P em T. Se somente os pares de posições inicial e final de cada ocorrência de P em T forem necessários, então o uso de espaço pode ser reduzido para $\theta(n)$ pois basta manter somente a lista das posições finais dos $(d-1)$-caminhos para calcular as posições finais dos d-caminhos.

4 Algoritmo de Landau e Vishkin Modificado para Usar Arranjos de Sufixos

Identificamos no uso de árvores de sufixos uma oportunidade de melhorar o uso de espaço do algoritmo de Landau e Vishkin. Nossa proposta é substituir o uso de uma árvore de sufixos nesse algoritmo por um arranjo de sufixos e estruturas adicionais para calcular em tempo constante os comprimentos dos maiores prefixos comuns de sufixos do texto e do padrão. A vantagem dessa modificação é diminuir o uso de memória com relação ao algoritmo original.

A maior parte da modificação foi realizada na fase de pré-processamento, e as mudanças na fase de iteração são mínimas.

4.1 Arranjo de Sufixos

O *arranjo de sufixos* é uma estrutura de dados introduzida por Manber e Myers(18) em 1989 com propósito similar ao da árvore de sufixos (ver seção 3.1.1), que é formar um índice de todos os sufixos de uma palavra para facilitar consultas às suas subpalavras.

Definição 4.1.1 (Arranjo de sufixos)**.** *Um arranjo de sufixos Pos para a palavra T é um arranjo que contém a seqüência dos sufixos de T segundo a ordem lexicográfica (18).*

Para a construção do arranjo de sufixos Pos, o alfabeto Σ precisa ser ordenado com uma ordem total. Costuma-se adicionar um caractere sentinela \$ ao fim de T para garantir que nenhum sufixo de T é prefixo de outro sufixo de T, e que possui a propriedade de ser ou maior ou menor que qualquer símbolo de Σ.

Como Pos é um arranjo de índices de posições em T, um arranjo de sufixos usa espaço $n \log n$ bits ($\theta(n)$ bytes). Tipicamente o espaço utilizado por um arranjo de sufixos é $4n$

bytes para um processador de 32-bits (para palavras com comprimento de até 4 bilhões de caracteres).

O arranjo de sufixos Pos para a palavra T pode ser construído em tempo $\theta(n)$ a partir da árvore de sufixos $\mathcal{T}$ para T. Al'em disso existem algoritmos $\theta(n)$ de construção direta – que não precisam de construir uma árvore de sufixos – como os de Ko e Aluru(19), de Kärkkäinen e Sanders(20) e Kim et al.(21). Além disso existem algoritmos com pior caso $O(n^2)$ que são mais rápidos que os algoritmos lineares para praticamente todos os casos[1].

Os algoritmos de Ko e Aluru e de Kärkkäinen e Sanders usam as idéias introduzidas por Farach(23) para chegar num algoritmo recursivo que seja $\theta(n)$ para a construção de arranjos de sufixos. A idéia básica é particionar a palavra em dois conjuntos de sufixos e ordenar um desses subconjuntos recursivamente. Feito isso o subconjunto ordenado é combinado com o subconjunto não ordenado usando características do critério de particionamento para acelerar a ordenação.

Como Pos é ordenado lexicograficamente, para pesquisar o padrão P em T usamos o algoritmo de busca binária e realizamos $\theta(m \log_2 n)$ comparações para responder se P é subpalavra de T. Além disso, em caso afirmativo, a resposta da pesquisa nos dá todos os sufixos de T do qual P é prefixo.

Acrescentamos ao arranjo de sufixos um arranjo chamado lcp. Dado o arranjo de sufixos Pos para a palavra $T = t_1...t_n$, o arranjo lcp é um arranjo de n elementos tal que $lcp(i)$ é o comprimento do maior prefixo comum de $T_{Pos(i)}$ e $T_{Pos(i+1)}$. O arranjo lcp pode ser construído em tempo linear a partir do arranjo de sufixos como descrito em (24). Um arranjo de sufixos acompanhado do arranjo lcp correspondente também é conhecido como *arranjo de sufixos melhorado* (25). Na figura 8 apresentamos o arranjo de sufixos Pos para a palavra $GATGACCA\$$, e o arranjo lcp correspondente.

Operações de pesquisa de subpalavras de que podem ser realizadas com árvores de sufixos podem ser realizadas com arranjos de sufixos com a complexidade aumentada por um fator multiplicativo $\theta(\log_2 n)$. A tabela LCP pode transformar esse fator numa soma de $\theta(\log_2 n)$ ao invés de uma multiplicação.

[1]Nas medidas de Manzini e Ferragina (22) os únicos casos em que a construção linear no pior caso foi melhor que a contrução linear no caso médio foram casos patológicos onde as palavras eram exclusivamente repetições de pequenas cadeias de caracteres (ou seja, o LCP médio entre dois sufixos adjacentes no arranjo de sufixos era muito alto).

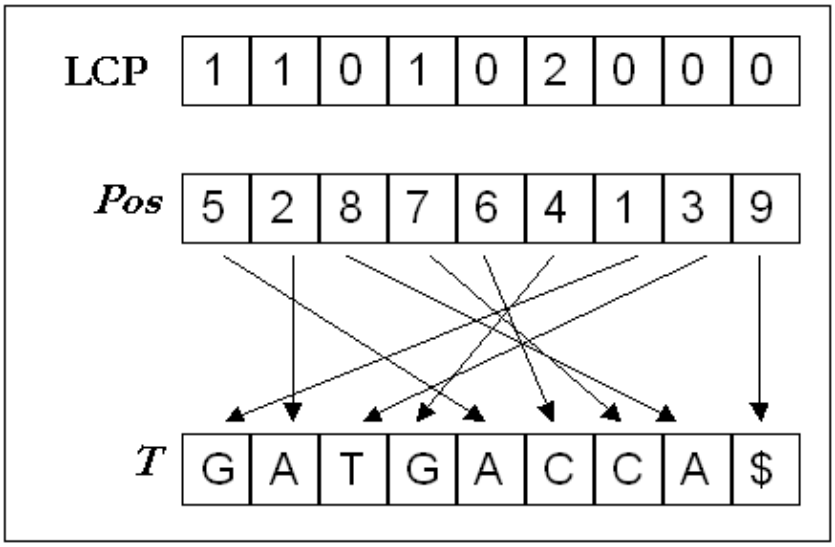


Figura 8: Arranjo de sufixos e LCP

4.2 Cálculo do Comprimento do Maior Prefixo Comum Usando Arranjos de Sufixos

O que permite o algoritmo de Landau e Vishkin manter a complexidade $\theta(kn)$ de tempo e espaço é o cálculo em tempo constante do comprimento do maior prefixo comum de sufixos do texto e do padrão. Mostramos aqui que é possível fazer esse mesmo cálculo em tempo constante usando arranjos de sufixos melhorados acrescidos de algumas estruturas de dados, mantendo a complexidade linear na fase de pré-processamento, e economizando espaço.

Dado o arranjo de sufixos melhorado Pos para a palavra $P\#T\$$, nós podemos processar o arranjo lcp correspondente e responder consultas do comprimento do maior prefixo comum de sufixos de $P\#T\$$ em tempo constante. A chave para essa operação é o teorema que segue.

Teorema 4.2.1. *A maior extensão comum $LCE_{S,S}(a,b)$ de dois sufixos S_a e S_b de S pode ser obtido do arranjo lcp da seguinte forma:*

Seja i a posição de S_a entre os sufixos ordenados de S (ou seja, $Pos(i) = a$). Seja j a posição de S_b entre os sufixos ordenados de S. Podemos assumir que $i < j$ sem perda de generalidade. Então a maior extensão comum de S_a e S_b é $LCE(a,b) = min_{i \leq k < j} lcp(k)$.

Demonstração. Sejam $S_a = s_a...s_{a+c}...s_n$ e $S_b = s_b...s_{b+c}...s_n$, e seja $c = LCE(a,b)$.

Se $i = j - 1$ então $k = i$ e $LCE(a,b) = c = lcp(i)$.

Se $i < j - 1$ então selecionamos k tal que $lcp(k)$ é o mínimo valor no intervalo $lcp(i) \dots lcp(j-1)$. Temos então dois casos possíveis:

- Se $c < lcp(k)$ temos uma contradição porque $s_a \dots s_{a+lcp(k)-1} = s_b \dots s_{b+lcp(k)-1}$ pela definição do arranjo lcp, e o fato que as entradas de lcp correspondem aos sufixos ordenados de S.
- se $c > lcp(k)$, seja $j = Pos(k)$ tal que S_j é o sufixo associado à posição k. S_k é tal que $s_j \dots s_{j+lcp(k)-1} = s_a \dots s_{a+lcp(k)-1}$ e $s_j \dots s_{j+lcp(k)-1} = s_b \dots s_{b+lcp(k)-1}$, mas como $s_a \dots s_{a+c-1} = s_b \dots s_{b+c-1}$ temos que o arranjo lcp estaria ordenado errôneamente o que é uma contradição.

Logo vale $LCE(a, b) = c = lcp(k)$

□

Dessa forma reduzimos a busca da maior extensão comum a uma consulta do valor mínimo em um intervalo do arranjo lcp. Essa consulta é conhecida por *RMQ* (*Range Minimum Query*).

Para resolver a consulta de valor mínimo num intervalo utilizaremos um algoritmo baseado em *Árvores Cartesianas* apresentadas em 1984 por Gabow, Bentley e Tarjan(26). Construiremos em $\theta(n)$ uma árvore cartesiana para o arranjo lcp tal que seja possível fazer a consulta de valor mínimo de qualquer intervalo em lcp em tempo constante utilizando uma consulta ao LCA de dois nós da árvore cartesiana em tempo constante.

Definição 4.2.1 (Árvores Cartesianas)**.** *Uma árvore cartesiana $\mathcal{C}$ para a seqüência de números inteiros $x_1 \dots x_n$ é a árvore binária com nós rotulados por esses números tal que a raiz da árvore é rotulada por m onde $x_m = \min x_i \mid 1 \leq i \leq n$, a sub-árvore à esquerda é a árvore cartesiana para $x_1 ... x_{m-1}$ e a sub-árvore à direita é a árvore cartesiana para $x_{m+1} \dots x_n$. Na figura 9 apresentamos a árvore cartesiana para a seqüência de números $< 1, 1, 0, 1, 0, 2, 0, 0, 0 >$ que corresponde ao arranjo lcp na figura 8. Acima de cada nó apresentamos o seu rótulo, e abaixo o índice em lcp correspondente a esse valor.*

Proposição 4.2.1. *O menor valor num intervalo $x_i \dots x_j$ pode ser encontrado por uma busca do $LCA_{\mathcal{C}}(i, j)$ de dois nós i e j da árvore cartesiana $\mathcal{C}$ construída com os valores do intervalo.*

Demonstração. Dados os nós i e j na árvore cartesiana $\mathcal{C}$, e seja v o LCA de i e j, e suponha que $i < j$. A estrutura de $\mathcal{C}$ é tal que se um nó $v = LCA(i, j)$, então $i \leq v \leq j$,

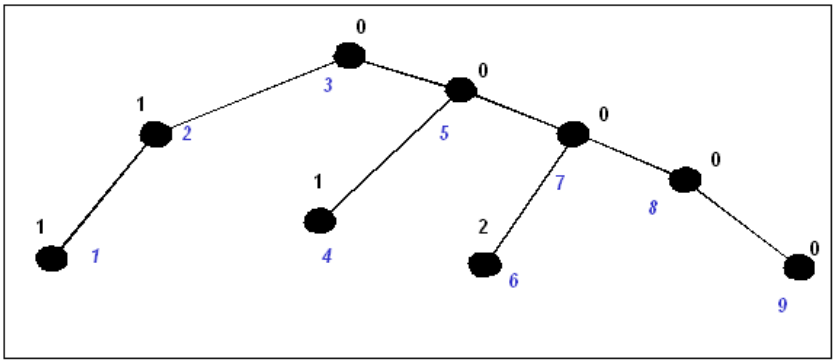


Figura 9: Árvore cartesiana

pois da construção da árvore cartesiana todo outro ancestral de v fica ou à esquerda de i e j, ou à direita de ambos. Além disso, da construção da árvore, o nó v é o nó tal que x_v é o menor valor de sua subárvore, e assim encontrar v, ancenstral comum mais profundo de i e j também significa encontrar o menor valor x_v no intervalo $x_i \dots x_j$.

□

Como já sabemos processar uma árvore qualquer em tempo linear para consultar o LCA de qualquer par de nós em tempo constante (ver seção 3.1.3), o mesmo vale para uma árvore cartesiana.

Precisamos mostrar então como construir uma árvore cartesiana em $\theta(n)$. Isso pode ser feito usando o algoritmo mostrado em (26) e (27).

Construímos a árvore cartesiana $\mathcal{C}_i$ para o arranjo $x_1, \dots, x_i$ da árvore cartesiana $\mathcal{C}_{i-1}$ para o arranjo $x_1, \dots, x_{i-1}$ da seguinte forma.

- se $i = 1$, então $\mathcal{C}_1$ é a árvore com um único nó 1 que é rotulado por x_1.
- se $i > 1$, então construímos $\mathcal{C}_i$ seguindo o caminho mais à direita da árvore desde a folha até a raiz, até encontrarmos um nó k tal que $x_i \geq x_k$.
- uma vez que encontramos o nó k, definimos que a sub-árvore à esquerda do nó i será a sub-árvore à direita de k, e fazemos o nó i como sendo a sub-árvore à direita do nó k.

Acrescentamos cada valor à árvore um valor de cada vez. A cada iteração acrescentamos o novo nó ao caminho mais a direita comparando-o com os demais nós desse caminho até encontrar um nó cujo rótulo seja menor que o seu. Observe que se o caminho mais à

direita possui k nós e se for preciso realizar $k' \leq k$ comparações para acrescentar um nó i, na próxima iteração será necessário fazer no máximo $k - k' + 1$ comparações porque todos os nós que estavam nesse caminho e eram maiores que i foram passados para a sub-árvore à esquerda de i e não serão feitas novas comparações com eles. Ou seja, sempre que for preciso fazer $n + 1$ comparações ao acrescentar um nó a $\mathcal{C}$ (com $n > 0$)), diminuimos o caminho mais a direita em n nós e conseqüentemente o número máximo de comparações para a inserção do próximo nó. Cada nó pode ser adicionado ao caminho mais à direita no máximo uma vez, e sair desse caminho no máximo uma vez, o algoritmo executa em tempo $\theta(n)$.

Finalmente, para realizar consultas do comprimento do maior prefixo comum em $O(1)$ após um pré-processamento em $\theta(n)$ fazemos:

- Construimos um arranjo de sufixos em $\theta(n)$ para o texto concatenado com o padrão, usando caracteres sentinelas.
- Construimos em $\theta(n)$ a tabela lcp para o arranjo de sufixos, e uma tabela de índices reversos R tal que $Pos(R(i)) = i$.
- Construimos em $\theta(n)$ a árvore cartesiana $\mathcal{C}$ para a tabela lcp
- Processamos $\mathcal{C}$ em $\theta(n)$ para respondermos consultas ao LCA de qualquer par de nós de $\mathcal{C}$ em $O(1)$.

Dados os sufixos i e j $(i < j)$ da palavra $P\#T\$$, o comprimento do seu maior prefixo comum vai ser o menor valor no intervalo of $[lcp(R(i)) \ldots lcp(R(j) - 1)]$, que será dado por uma consulta do ancestral comum mais profundo em $O(1)$ na árvore cartesiana $\mathcal{C}$.

4.3 O algoritmo proposto

O algoritmo proposto é o mesmo algoritmo de Landau e Vishkin, substituindo a árvore de sufixos por um arranjo de sufixos melhorado, e a consulta do LCA na árvore de sufixos por uma consulta do valor mínimo num intervalo da tabela LCP por meio de uma consulta de LCA em uma árvore cartesiana, que descrevemos como o Algoritmo 3.

Algoritmo 3 Algoritmo de Landau e Vishkin Modificado para pesquisa aproximada de padrões

1. Construímos o arranjo de sufixos melhorado Pos para $P\#T\$$.
2. Construímos a árvore cartesiana $\mathcal{C}$ para o arranjo lcp
3. Processamos $\mathcal{C}$ em $O(n)$ de forma a responder consultas LCA em $O(1)$.
4. Para cada diagonal i da tabela de programação dinâmica, encontramos o seu 0-caminho de maior alcance com uma consulta do menor valor no intervalo correspondente em lcp.
5. Para $d = 1 \ldots k$:
   5.1 Para cada diagonal i da tabela de programação dinâmica:
      5.1.1 Estendemos o $(d-1)$-caminho de maior alcance na diagonal $i-1$ uma célula para a direita para a célula $D(r, s)$ na diagonal i
      5.1.2 Estendemos esse caminho ao longo da diagonal i pela quantidade de células igual ao menor valor no intervalo correspondente de lcp dado pela consulta do LCA em $\mathcal{C}$.
      5.1.3 Estendemos o $(d-1)$-caminho de maior alcance na diagonal $i+1$ uma célula para a baixo para a célula $D(r', s')$ na diagonal i
      5.1.4 Estendemos esse caminho ao longo da diagonal i pela quantidade de células igual ao menor valor no intervalo correspondente de lcp dado pela consulta do LCA em $\mathcal{C}$.
      5.1.5 Estendemos o $(d-1)$-caminho de maior alcance na diagonal i uma célula ao longo da diagonal i para a célula $D(r'', s'')$
      5.1.6 Estendemos esse caminho ao longo da diagonal i pela quantidade de células igual ao menor valor no intervalo correspondente de lcp dado pela consulta do LCA em $\mathcal{C}$.
      5.1.7 Escolhemos o d-caminho de maior alcande dentre os três.
6. Cada caminho que alcançar a linha m é uma ocorrência de P em T com no máximo k erros.

4.4 Análise do algoritmo proposto

Somente a fase de pré-processamento foi significativamente alterada, já que a consulta ao valor mínimo num intervalo de *lcp* continua sendo dado por uma consulta ao LCA em $O(1)$, que é a mesma operação realizada no algoritmo original, acrescida de duas consultas diretas na tabela R (que contém os índices reversos que mapeiam os sufixos de P e T na suas posições em Pos).

Teorema 4.4.1 (Complexidade de tempo e espaço do algoritmo modificado)**.** *O algoritmo modificado de Landau e Vishkin para pesquisa aproximada de padrões possui complexidade de tempo e espaço* $\theta(nk)$.

Demonstração. Como comentado acima, a construção e manutenção de arranjos de sufixos pode ser feita em $\theta(n)$ tempo e espaço (19, 20, 21) assim como a construção e manutenção de árvores cartesianas. Como o pré-processamento para a consulta de LCA em tempo constante é $\theta(n)$, então a complexidade do pré-processamento com respeito ao uso de tempo e espaço também é $\theta(n)$.

O acréscimo à fase de iteração são duas consultas $O(1)$ em uma tabela, de forma que a complexidade da fase de iteração no algoritmo modificado é $\theta(kn) > \theta(n)$. Dessa forma, o algoritmo modificado também usa tempo e espaço da ordem $\theta(kn)$.

□

Apesar dos limites teóricos do algoritmo modificado coincidirem com os do algoritmo original, a nossa versão utiliza menos espaço durante o pré-processamento das palavras T e P, e também na execução completa do algoritmo.

Suponha que a implementação da árvore de sufixos seja boa o suficiente, como a implementação de Kurtz, e utilize $12n$ bytes para a representação da árvore de sufixos, construída com $\frac{3}{2}n$ nós. Então o espaço total para o pré-processamento será $12n + S\frac{3}{2}n$, onde S é o espaço por nó utilizado no pré-processamento para cálculo do ancestral comum mais profundo.

Ora, supondo que a implementação da árvore de sufixos contenha toda a informação necessária para o pré-processamento, então $S = S_I + S_L + S_A + S_{id}$, onde S_I, S_L e S_A são os espaços necessários para as funções A, I e L, e S_{id} o espaço necessário para a numeração dos nós de $\mathcal{T}$. Usando um inteiro de 32-bits (4 bytes) por entrada em cada uma dessas tabelas, temos que $S = 16$ bytes e o uso de espaço do pré-processamento com árvores de

sufixos será $36n$ bytes.

Para a versão modificada, uma vez construída a árvore cartesiana, não é mais necessária a manutenção do arranjo de sufixos, mas somente do arranjo *lcp* que usa $4n$ bytes, do índice reverso R que também usa $4n$ bytes e da árvore cartesiana que usa $8n$ bytes. Como a árvore cartesiana tem exatamente n nós, então o uso de espaço do pré-processamento para arranjos de sufixos será $4n+8n+4n+16n = 32n$ que é uma economia de $4n$ bytes sobre a versão original.

Quanto ao tempo de execução, em primeiro lugar é preciso comparar a construção da árvore de sufixos com a do arranjo de sufixos, porque as construções do arranjo *lcp* e da árvore cartesiana são muito mais rápidas que a do arranjo de sufixos. O pré-processamento da árvore cartesiana é mais rápido que o da árvore de sufixos, pois há menos nós para serem processados. Em contrapartida entendemos que a fase iteração execute um pouco mais rápido com a versão original porque nesse caso não são necessárias as duas consultas ao índice reverso do sufixo de P e do sufixo de T para os quais estamos calculando o LCE.

5 *Análise Experimental*

Como análise experimental avaliamos o comportamento da versão original e da versão modificada do algoritmo de Landau e Vishkin.

Os primeiros experimentos foram realizados com dados gerados de forma aleatória. Geramos várias palavras e padrões de tamanhos crescentes com quatros alfabetos de tamanhos distintos — $|\Sigma| = 2$, $|\Sigma| = 4$, $|\Sigma| = 26$ e $|\Sigma| = 93$ — valores que representam os tamanhos dos alfabetos binário, quartenário (RNA e DNA), alfabeto da língua inglesa e caracteres ASCII que podem ser impressos (que neste trabalho chamamos de ASCII93).

Para os dados reais, dividimos a análise em duas partes. Na primeira parte usamos textos e seqüências retiradas do projeto Gutenberg (28), do NCBI Genbank (29) e do *corpus* de Canterbury (30). Na segunda parte selecionamos seqüências biológicas de escala cromossômica (por exemplo, todo o cDNA do cromossomo 22 do *H. sapiens*) e fizemos pesquisas de porções de cDNA nessas seqüências usando as variantes que usam espaço $\theta(n)$ dos algoritmos.

5.1 Implementações utilizadas

Buscamos utilizar implementações eficientes para as estruturas de dados. Em especial, usamos para a árvore de sufixos a implementação na linguagem de programação C de Kurtz que é usada em um programa de alinhamento de genomas, e para arranjos de sufixos a implementação de Manzini e Ferragina da ordenação de sufixos de uma palavra (também escrita na linguagem C). O arranjo *lcp* foi implementado na linguagem C++ pelo autor a partir de (24), e as demais estruturas de dados e algoritmos foram implementados pelo autor em C++.

5.1.1 Árvore de sufixos

A implementação da árvore de sufixos utilizada foi a implementação de Kurtz para o software MUMMER 3.0 (15) (31), que utiliza as técnicas descritas em (13) para diminuir o uso de espaço da estrutura de dados. O algoritmo utilizado para a construção de árvores de sufixos é o de McCreight (11). O código foi compilado com a opção `HUGE` que prepara a árvore de sufixos para aceitar palavras maiores que 128 megabytes, ao custo de um espaço adicional, aumentando de cerca de 12 bytes por caractere para 15 bytes por caractere na construção de árvores de sufixos para palavras com $|\Sigma| = 4$. Foi preciso realizar algumas modificações pequenas no código para que a busca em profundidade funcionasse como esperado, pois a busca em profundidade como estava implementada não fazia a visita inicial à raiz da árvore.

A implementação é bastante eficiente na execução e no uso de espaço, mas por isso foi necessário acrescentar um pouco mais de informações à estrutura que é gerada durante a fase de pré-processamento do algoritmo de Landau e Vishkin. Além das estruturas típicas descritas na seção 3.1.3(tabelas I, L, A e de rótulos dos nós), foi necessário um arranjo com os ponteiros para os nós pais de cada nó e mais um arranjo com os rótulos dos nós pais, porque o resultado do cálculo $LCA(v, w)$ é o rótulo na pesquisa em profundidade do nó que é o LCA de v e w. A implementação utilizada também não expõe diretamente todos os nós da árvore, mas apenas as folhas. Essa característica faz com que o cálculo $LCA(v, w)$ em $O(1)$ implementado funcione apenas se tanto v quanto w forem *folhas* de $\mathcal{T}$. Como esse é exatamente o caso, não causou maiores impactos no algoritmo de Landau e Vishkin, mas para uma utilização diferente que precise calcular o LCA de nós arbitrários seria necessário uma implementação mais complexa e que usasse espaço adicional para mapear os nós internos da árvore. Assim, sendo n o comprimento de $T\#P\$$ e v o número de nós de $\mathcal{T}$, o espaço total para o pré-processamento foi de $20v + 4n + S_{\mathcal{T}}$, onde $S_{\mathcal{T}}$ é o espaço usado pela árvore de sufixos em si. Se assumirmos que $S_{\mathcal{T}} = 12n$ e $v = \frac{3}{2}n$, então o espaço usado no pré-processamento seria $46n$ bytes. Na prática o cálculo exato é difícil de determinar, pois a quantidade de nós da árvore depende do alfabeto utilizado e da estrutura da palavra indexada pela árvore de sufixos — em especial quando $|\Sigma|$ é grande, a árvore de sufixos tende a precisar de menos nós. O uso de espaço na implementação utilizada é de $14n$ a $15n$ para alfabetos pequenos (DNA, RNA) e $10n$ a $11n$ para alfabetos maiores (ASCII). Na seção 5.2 será apresentada a avaliação realizada para alguns valores de v.

5.1.2 Arranjos de sufixos

A escolha de implementação da construção arranjo de sufixos vai influenciar a velocidade total do algoritmo, mas não o uso de espaço, porque o espaço utilizado pelo arranjo *Pos* é fixo em $4n$ bytes como descrito na seção 4.1, e não é influenciado pelo alfabeto ou a estrutura da palavra. Após a construção do arranjo *lcp* e do índice reverso, o arranjo *Pos* não é mais necessário e pode ser descartado. A árvore cartesiana usa exatamente $8n$ bytes e mais $4n$ bytes para o arranjo *lcp*. O índice reverso — necessário para mapear um sufixo de $T\#P\$$ em *lcp* — usa $4n$ bytes. A estrutura do pré-processamento adiciona um fator de $20n$ bytes perfazendo um total de $36n$ bytes.

5.1.3 Usando algoritmos que melhoram o caso médio

Uma avaliação da literatura sobre arranjos de sufixos apresentou alguns resultados interessantes. Existem métodos de construção direta dos arranjos de sufixos com complexidade linear para o pior caso, como os algoritmos propostos por Ko e Aluru(19), Kärkkäinen e Sanders(20) e Kim et al.(21). Num estudo realizado por Puglisi, Smyth e Turpin(32) vários algoritmos de construção de arranjos de sufixos de complexidade linear e linear no caso médio foram avaliados e os autores verificaram que, em praticamente todos os casos, algoritmos cujo pior caso é quadrático mas que no caso médio são lineares têm desempenho bem melhor que os algoritmos lineares no pior caso. Esses algoritmos costumam se comportar mal em casos patológicos, em que o valor médio do arranjo *lcp* é alto relativo ao texto (ou seja o texto é composto por muitas repetições). Dentre esses algoritmos escolhemos o *Deep-Shallow Sort*, desenvolvido por Manzini e Ferragina(22).

O algoritmo *Deep-Shallow Sort* se caracteriza por dividir a ordenação dos sufixos de T em duas fases. Na primeira fase, chamada de fase rasa (*shallow*) ordena os sufixos de T com base nos seus prefixos de até L caracteres usando o algoritmo *multikey quicksort* de Bentley e Sedgewick(33). Nesse ponto temos todos os sufixos de T ordenados até o seu L-ésimo caractere. Na segunda fase, chamada de profunda (*deep*), utiliza-se uma combinação de dois algoritmos, *blind sorting*, apresentado por Ferragina e Grossi em (34) e *ternary quicksort* apresentado por Bentley e McIlroy em (35), de forma a usar o algoritmo mais eficiente para as palavras que estão sendo ordenadas em cada passo do algoritmo.

A implementação utilizada é a implementação em linguagem C de Manzini e Ferragina(22).

5.1.4 Diminuindo mais o uso de espaço

Analisando com cuidado a implementação da árvore cartesiana, percebemos que após o pré-processamento a mesma não é mais necessária, de forma que o uso de espaço cairia para $28n$ bytes, usados basicamente para o arranjo *lcp*, o índice reverso ($4n$ cada) e as estruturas de pré-processamento para a consulta LCA em tempo constante. Acreditamos ser possível fazer uma economia similar para versão baseada em árvores de sufixos para reduzir o espaço final para $30n$ a $36n$ bytes, mantendo o pico de uso de espaço da ordem de $41n$ a $51n$ bytes. O pico do uso de espaço para o pré-processamento de arranjos de sufixos continua sendo $36n$ bytes. A implementação baseada em árvores de sufixos aqui descrita não incluiu essa modificação.

5.2 Análise e Comparação de resultados

Para fazer a análise experimental, foram usados dados gerados aleatoriamente e dados reais retirados do Projeto Gutenberg (28), NCBI Genbank (29) e do *corpus* de Canterbury (30). O uso duas massas de dados com características diferentes quanto à distribuição dos caracteres permite uma análise de como o algoritmo tende a se comportar em situações diversas.

Observe que com relação aos valores percentuais apresentados, o valor de referência (100%) será sempre o valor do algoritmo de Landau e Vishkin original (baseado em árvores de sufixos).

5.2.1 Ambiente computacional

Os testes experimentais foram executados em um computador DELL PowerEdge 1800 com 2 GB de memória RAM e processador Intel Xeon de 3.0 GHz com 2MB de cache L2 e acesso à memória controlado por um FSB de 800MHz. O sistema operacional utilizado foi o Red Hat Enterprise Linux usando Linux 2.6.9-5, com compilador gcc versão 3.4.3 e biblioteca glibc versão 2.3.4-2. Além disso, para o gerador de palavras aleatórias usamos um programa python rodando em uma VM Python 2.3.4, usando o dispositivo provedor de entropia `/dev/random` e o algoritmo de geração de números pseudo aleatórios *Mersenne Twister*.

5.2.2 Dados aleatórios

Optamos por executar 5 séries de testes aleatórios para quatro tamanhos de alfabeto, a saber 2 (*binário*), 4 (*DNA*), 26 (*alfabeto*) e 93 (*caracteres ASCII93*). As séries são baseadas no tamanho do texto utilizado na pesquisa aproximada e o número máximo de erros permitidos. As séries utilizadas foram de:

- 1000 a 10.000 caracteres, em intervalos de 1000 caracteres, usando padrão de 100 caracteres.
- 10.000 a 100.000 caracteres, em intervalos de 10.000 caracteres, usando padrão de 100 caracteres.
- 100.000 a 1.000.000 caracteres, em intervalos de 100.000 caracteres, usando padrão de 1.000 caracteres.
- 1.000.000 a 10.000.000 caracteres, em intervalos de 1.000.000 caracteres, usando padrão de 1.000 caracteres.
- 11.000.000 a 20.000.000 caracteres, em intervalos de 1.000.000 caracteres, usando padrão de 10.000 caracteres.

Para cada tamanho de texto de cada série foram gerados 5 arquivos aleatórios de cada alfabeto, e o uso de memória e tempo de execução informados são a média dos dados recolhidos desses 5 arquivos. Como para palavras de tamanho pequeno as diferenças não são significativas, apresentamos nas tabelas e gráficos apenas os resultados para $n \geq$ 1.000.000. Nas tabelas 1, 2, 3 e 4 temos a comparação com ponto de vista de uso de espaço para o algoritmo que usa a construção de árvore de sufixos no seu pré-processamento e o algoritmo que usa arranjos de sufixos para seu pré-processamento.

A comparação de tempo foi dividida em duas tabelas para cada tamanho de alfabeto, uma tabela para o *tempo de execução*, e outra para o *tempo do processador*. *É importante fazer a distinção entre essas duas medidas para que os resultados sejam interpretados corretamente*:

- Por *tempo de execução* entendemos a medida de tempo total que o algoritmo levou para ser executado com sucesso no ambiente computacional. O tempo de execução pode ser afetado por outros programas executando no sistema, pelo uso de memória virtual, carga de processamento da máquina e eventos e originados pelo sistema operacional ou outros programas que interrompam a execução do programa.

- Por *tempo do processador* entendemos o tempo de uso do processador que o algoritmo usou no seu processamento. O tempo do processador é equivalente ao tempo que o algoritmo total usaria se fosse o único programa em execução no sistema e todos os dados utilizados coubessem na memória principal, e não sofre influência de eventos externos ao programa que interrompam a sua execução.

Nos experimentos realizados, o computador estava dedicado à execução de nossos experimentos, de forma que diferenças significativas entre tempo de processador e tempo de execução coletados dizem respeito ao uso de memória virtual.

As tabelas 6, 8, 10 e 12 apresentam as comparações do ponto de vista do tempo do processador, enquanto as tabelas 9, 7, 11 e 13 apresentam as comparações do ponto de vista do tempo de execução.

As tabelas 1, 2, 3 e 4 que descrevem o uso de espaço estão estruturadas da seguinte forma:

- Uma coluna descrevendo o tamanho do problema — N — para a execução do algoritmo, contado em milhares de caracteres
- Quatro colunas descrevendo o uso de espaço para a implementação baseada em arranjos de sufixos e quatro para a baseada em árvores de sufixos, nessa ordem, sendo que essas quatro colunas estão dispostas da seguinte forma:
  - Duas colunas descrevendo o espaço utilizado após o término da fase de pré-processamento, descrevendo a quantidade total de KBytes (KB) e a quantidade em bytes por caractere (Bpc).
  - Duas colunas descrevendo o uso de espaço total do algoritmo de Landau e Vishkin – incluindo o espaço utilizado pela fase de pré-processamento – descrevendo a quantidade total de KBytes (KB) e a quantidade em bytes por caractere (Bpc).
- Quatro colunas descrevendo a diferença no uso de espaço, sendo:
  - Duas colunas com a diferença do uso de espaço da implementação baseada em árvores de sufixos e o uso de espaço da implementação baseada em arranjos de sufixos, em KBytes e bytes por caractere, onde um valor positivo indica que o uso de espaço da versão baseada em árvores de sufixos é maior.

Tabela 1: Uso de espaço para $|\Sigma| = 93$, $k = 20$

	Arranjo de Sufixos				Árvore de Sufixos				Economia de Espaço			
	Pre-proc		Total		Pre-proc		Total		bytes	Bpc	Pre-Proc	Total
N (×1000)	KB	Bpc.	KB	Bpc	KB	Bpc	KB	Bpc				
1.000	27.371	28	86.943	89	38.872	40	98.444	101	11.501	12	29,59%	11,68%
2.000	54.715	28	173.858	89	78.402	40	197.545	101	23.687	12	30,21%	11,99%
3.000	82.059	28	260.772	89	112.889	39	291.602	100	30.830	11	27,31%	10,57%
4.000	109.402	28	347.686	89	144.432	37	382.715	98	35.029	9	24,25%	9,15%
5.000	136.746	28	434.600	89	174.889	36	472.742	97	38.143	8	21,81%	8,07%
6.000	164.090	28	521.514	89	205.161	35	562.585	96	41.072	7	20,02%	7,30%
7.000	191.434	28	608.428	89	235.634	34	652.628	95	44.200	6	18,76%	6,77%
8.000	218.777	28	695.342	89	266.429	34	742.993	95	47.651	6	17,89%	6,41%
9.000	246.121	28	782.256	89	297.581	34	833.716	95	51.460	6	17,29%	6,17%
10.000	273.465	28	869.170	89	329.169	34	924.874	95	55.704	6	16,92%	6,02%
11.000	301.055	28	956.330	89	361.381	34	1.016.656	95	60.326	6	16,69%	5,93%
12.000	328.399	28	1.043.244	89	393.703	34	1.108.548	95	65.304	6	16,59%	5,89%
13.000	355.742	28	1.130.158	89	426.370	34	1.200.786	95	70.628	6	16,56%	5,88%
14.000	383.086	28	1.217.072	89	459.403	34	1.293.390	95	76.317	6	16,61%	5,90%
15.000	410.430	28	1.303.986	89	492.793	34	1.386.350	95	82.363	6	16,71%	5,94%
16.000	437.774	28	1.390.900	89	526.522	34	1.479.649	95	88.749	6	16,86%	6,00%
17.000	465.117	28	1.477.815	89	560.489	34	1.573.187	95	95.372	6	17,02%	6,06%
18.000	492.461	28	1.564.729	89	594.882	34	1.667.149	95	102.421	6	17,22%	6,14%
19.000	519.805	28	1.651.643	89	629.554	34	1.761.392	95	109.749	6	17,43%	6,23%
20.000	547.149	28	1.738.557	89	664.506	34	1.855.915	95	117.358	6	17,66%	6,32%

Tabela 2: Uso de espaço para $|\Sigma| = 26$, $k = 20$

	Arranjo de Sufixos				Árvore de Sufixos				Economia de Espaço			
	Pre-proc		Total		Pre-proc		Total		bytes	Bpc	Pre-Proc	Total
N (×1000)	KB	Bpc.	KB	Bpc	KB	Bpc	KB	Bpc				
1.000	27.371	28	86.943	89	40.829	42	100.402	103	13.458	14	32,96%	13,40%
2.000	54.715	28	173.858	89	77.724	40	196.867	101	23.009	12	29,60%	11,69%
3.000	82.059	28	260.772	89	112.830	38	291.543	99	30.771	10	27,27%	10,55%
4.000	109.402	28	347.686	89	148.997	38	387.280	99	39.594	10	26,57%	10,22%
5.000	136.746	28	434.600	89	186.654	38	484.508	99	49.908	10	26,74%	10,30%
6.000	164.090	28	521.514	89	225.621	38	583.045	99	61.531	10	27,27%	10,55%
7.000	191.434	28	608.428	89	265.680	39	682.674	100	74.246	11	27,95%	10,88%
8.000	218.777	28	695.342	89	306.579	39	783.144	100	87.802	11	28,64%	11,21%
9.000	246.121	28	782.256	89	348.167	40	884.302	101	102.046	12	29,31%	11,54%
10.000	273.465	28	869.170	89	390.300	40	986.006	101	116.836	12	29,93%	11,85%
11.000	301.055	28	956.330	89	433.174	40	1.088.449	101	132.119	12	30,50%	12,14%
12.000	328.399	28	1.043.244	89	475.847	41	1.190.693	102	147.449	13	30,99%	12,38%
13.000	355.742	28	1.130.158	89	518.654	41	1.293.070	102	162.911	13	31,41%	12,60%
14.000	383.086	28	1.217.072	89	561.515	41	1.395.501	102	178.429	13	31,78%	12,79%
15.000	410.430	28	1.303.986	89	604.317	41	1.497.873	102	193.887	13	32,08%	12,94%
16.000	437.774	28	1.390.900	89	647.032	41	1.600.159	102	209.258	13	32,34%	13,08%
17.000	465.117	28	1.477.815	89	689.596	42	1.702.293	102	224.479	14	32,55%	13,19%
18.000	492.461	28	1.564.729	89	731.948	42	1.804.216	103	239.487	14	32,72%	13,27%
19.000	519.805	28	1.651.643	89	774.101	42	1.905.939	103	254.296	14	32,85%	13,34%
20.000	547.149	28	1.738.557	89	816.031	42	2.007.439	103	268.882	14	32,95%	13,39%

– Duas colunas indicando a porcentagem que espaço economizado representa do espaço utilizado pela implementação baseada em árvores de sufixos para o pré-processamento e para o uso total de espaço do algoritmo

As figuras 10, 11, 12 e 13 apresentam de forma gráfica a diferença no uso de espaço no pré-processamento e total paras palavras usadas com mais de 1.000.000 de caracteres.

Analisando as tabelas e os gráficos de uso de espaço vemos que consistentemente a implementação do algoritmo modificado usou menos espaço. A diferença é maior para alfabetos pequenos, chegando a ser de 45% para a fase de pré-processamento quando $|\Sigma| = 4$. A economia de espaço para o algoritmo completo (pré-processamento e iteração) é menor, e percebe-se que na medida em que o valor do parâmetro k aumenta, o efeito dessa economia de espaço na execução completa do algoritmo fica menos significativa.

Tabela 3: Uso de espaço para $|\Sigma| = 4$, $k = 20$

	Arranjo de Sufixos				Árvore de Sufixos				Economia de Espaço			
	Pre-proc		Total		Pre-proc		Total		bytes	Bpc	Pre-Proc	Total
N (×1000)	KB	Bpc.	KB	Bpc	KB	Bpc	KB	Bpc				
1.000	27.371	28	86.943	89	50.381	52	109.953	112	23.010	24	45,67%	20,93%
2.000	54.715	28	173.858	89	100.623	51	219.765	112	45.908	23	45,62%	20,89%
3.000	82.059	28	260.772	89	151.093	52	329.806	113	69.034	24	45,69%	20,93%
4.000	109.402	28	347.686	89	201.362	52	439.646	113	91.960	24	45,67%	20,92%
5.000	136.746	28	434.600	89	251.497	51	549.351	112	114.751	23	45,63%	20,89%
6.000	164.090	28	521.514	89	301.668	51	659.092	112	137.578	23	45,61%	20,87%
7.000	191.434	28	608.428	89	351.953	51	768.947	112	160.519	23	45,61%	20,88%
8.000	218.777	28	695.342	89	402.339	51	878.904	112	183.562	23	45,62%	20,89%
9.000	246.121	28	782.256	89	452.816	52	988.951	113	206.695	24	45,65%	20,90%
10.000	273.465	28	869.170	89	503.337	52	1.099.042	113	229.872	24	45,67%	20,92%
11.000	301.055	28	956.330	89	554.249	52	1.209.524	112	253.194	24	45,68%	20,93%
12.000	328.399	28	1.043.244	89	604.684	52	1.319.530	113	276.285	24	45,69%	20,94%
13.000	355.742	28	1.130.158	89	655.028	52	1.429.444	113	299.286	24	45,69%	20,94%
14.000	383.086	28	1.217.072	89	705.353	52	1.539.339	113	322.267	24	45,69%	20,94%
15.000	410.430	28	1.303.986	89	755.579	52	1.649.135	113	345.149	24	45,68%	20,93%
16.000	437.774	28	1.390.900	89	805.808	52	1.758.935	113	368.034	24	45,67%	20,92%
17.000	465.117	28	1.477.815	89	855.938	52	1.868.636	112	390.821	24	45,66%	20,91%
18.000	492.461	28	1.564.729	89	906.086	52	1.978.353	112	413.625	24	45,65%	20,91%
19.000	519.805	28	1.651.643	89	956.233	52	2.088.071	112	436.429	24	45,64%	20,90%
20.000	547.149	28	1.738.557	89	1.006.329	51	2.197.737	112	459.180	23	45,63%	20,89%

Tabela 4: Uso de espaço para $|\Sigma| = 2$, $k = 20$

	Arranjo de Sufixos				Árvore de Sufixos				Economia de Espaço			
	Pre-proc		Total		Pre-proc		Total		bytes	Bpc	Pre-Proc	Total
N (×1000)	KB	Bpc.	KB	Bpc	KB	Bpc	KB	Bpc				
1.000	27.371	28	86.943	89	61.897	63	121.470	124	34.526	35	55,78%	28,42%
2.000	54.715	28	173.858	89	123.733	63	242.876	124	69.018	35	55,78%	28,42%
3.000	82.059	28	260.772	89	185.574	63	364.287	124	103.515	35	55,78%	28,42%
4.000	109.402	28	347.686	89	247.409	63	485.693	124	138.007	35	55,78%	28,41%
5.000	136.746	28	434.600	89	309.242	63	607.096	124	172.496	35	55,78%	28,41%
6.000	164.090	28	521.514	89	371.080	63	728.503	124	206.990	35	55,78%	28,41%
7.000	191.434	28	608.428	89	432.912	63	849.906	124	241.478	35	55,78%	28,41%
8.000	218.777	28	695.342	89	494.749	63	971.313	124	275.972	35	55,78%	28,41%
9.000	246.121	28	782.256	89	556.582	63	1.092.717	124	310.461	35	55,78%	28,41%
10.000	273.465	28	869.170	89	618.427	63	1.214.132	124	344.962	35	55,78%	28,41%
11.000	301.055	28	956.330	89	680.825	63	1.336.100	124	379.770	35	55,78%	28,42%
12.000	328.399	28	1.043.244	89	742.659	63	1.457.504	124	414.260	35	55,78%	28,42%
13.000	355.742	28	1.130.158	89	804.498	63	1.578.914	124	448.756	35	55,78%	28,42%
14.000	383.086	28	1.217.072	89	866.329	63	1.700.316	124	483.243	35	55,78%	28,42%
15.000	410.430	28	1.303.986	89	928.166	63	1.821.723	124	517.737	35	55,78%	28,42%
16.000	437.774	28	1.390.900	89	989.990	63	1.943.117	124	552.216	35	55,78%	28,42%
17.000	465.117	28	1.477.815	89	1.051.840	63	2.064.537	124	586.723	35	55,78%	28,42%
18.000	492.461	28	1.564.729	89	1.113.673	63	2.185.941	124	621.212	35	55,78%	28,42%
19.000	519.805	28	1.651.643	89	1.175.505	63	2.307.343	124	655.700	35	55,78%	28,42%
20.000	547.149	28	1.738.557	89	1.237.345	63	2.428.753	124	690.196	35	55,78%	28,42%

Tabela 5: Uso espaço nas árvores de sufixo

Σ	$\lvert\Sigma\rvert$	# *nós*	*Espaço* $\mathcal{T}$	*Espaço Pré-proc* $\mathcal{T}$	*Espaço Pré-proc SA*
Binário	2	$1.8n$	$19n$	$63n$	$28n$
DNA	4	$1.6n$	$15n$	$51n$	$28n$
Alfabeto	26	$1.3n$	$10n$	$40n$	$28n$
Caracteres ASCII93	93	$1.2n$	$9n$	$37n$	$28n$

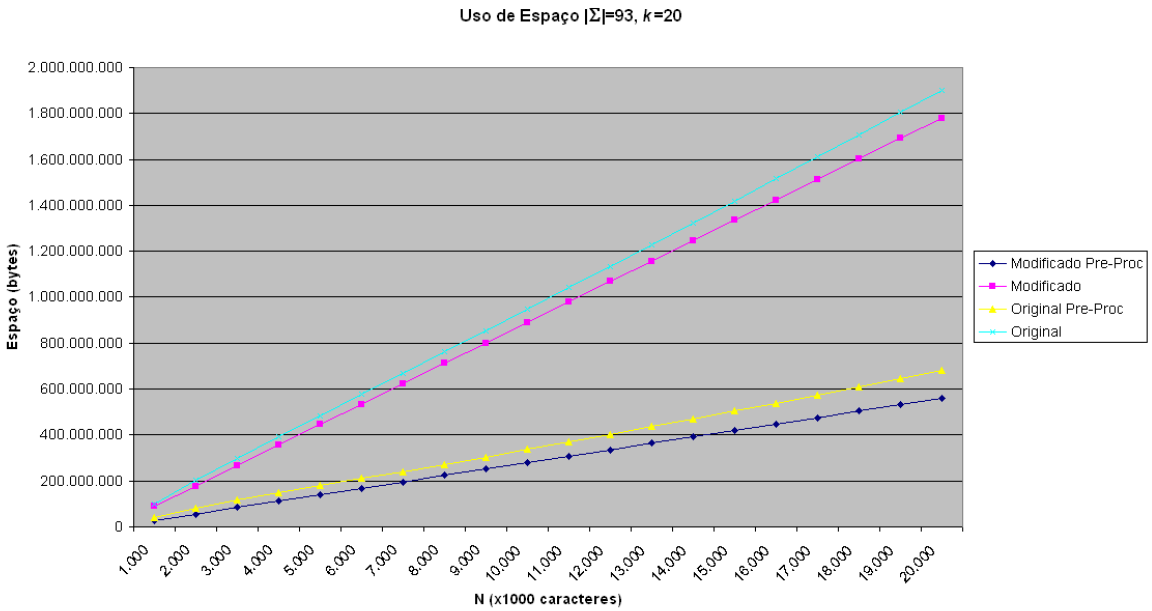


Figura 10: Gráfico de uso de espaço $|\Sigma| = 93$, $k = 20$

Observe que o uso de espaço da árvore de sufixos depende da estrutura da palavra e do alfabeto utilizado. Compilamos na tabela 5 a relação encontrada entre o alfabeto utilizado e a estrutura da árvore de sufixos construída, incluindo número total de nós e espaço utilizado por caractere para a árvore de sufixos e para a estrutura gerada pelo pré-processamento para cálculo do LCA. Os valores foram obtidos a partir dos valores médios para cada alfabeto. A coluna *Espaço Pré-proc SA* descreve o espaço utilizado pela versão baseada em arranjos de sufixos para comparação.

Como descrito na seção 5.1.1 a implementação de árvore de sufixos utilizada foi a de Kurtz usada no software MUMMER (15). Os valores apresentados na tabela 5 são os valores médios para o uso de espaço, aproximados para o número inteiro mais próximo. Para alfabetos pequenos, a árvore de sufixos usa mais espaço. É interessante comentar que para o caso do alfabeto grande (na tabela, os caracteres ASCII93), o uso de espaço é tal que, descartada a árvore de sufixos, o espaço final ficaria praticamente igual ao da versão baseada em arranjos de sufixos. Entretanto verificamos que para esses casos o

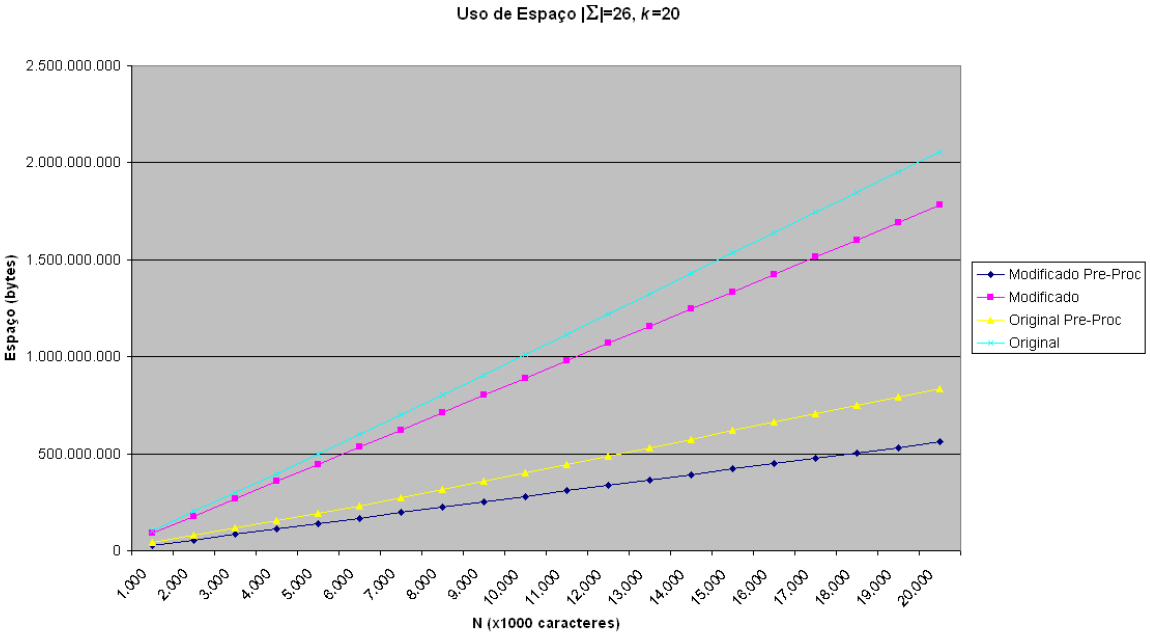


Figura 11: Gráfico de uso de espaço $|\Sigma| = 26$, $k = 20$

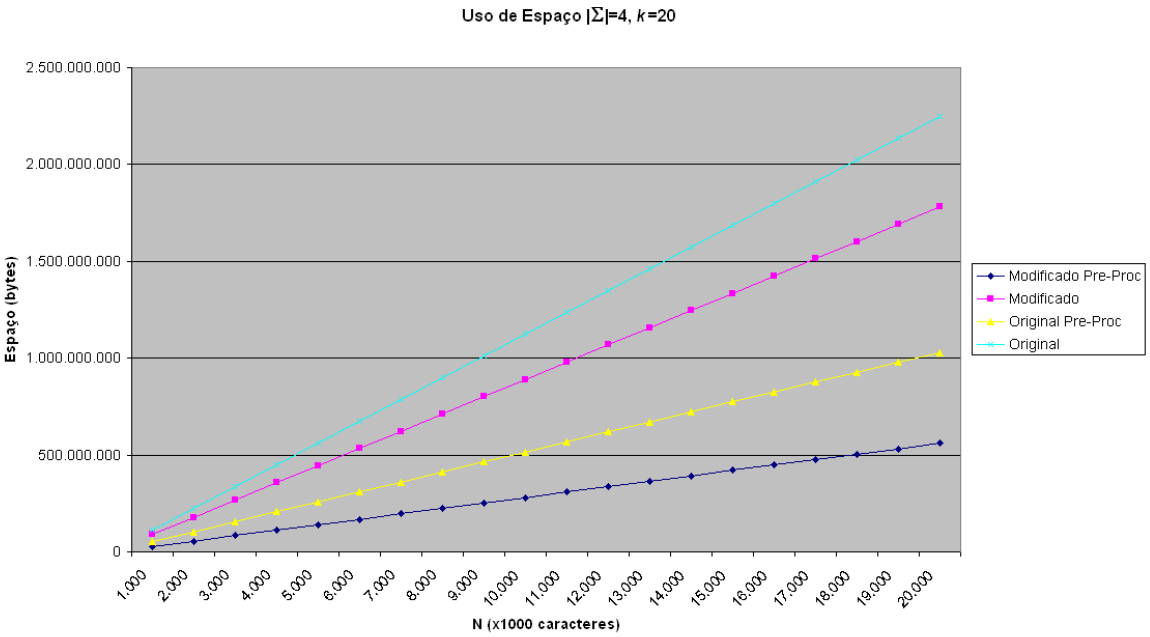


Figura 12: Gráfico de uso de espaço $|\Sigma| = 4$, $k = 20$

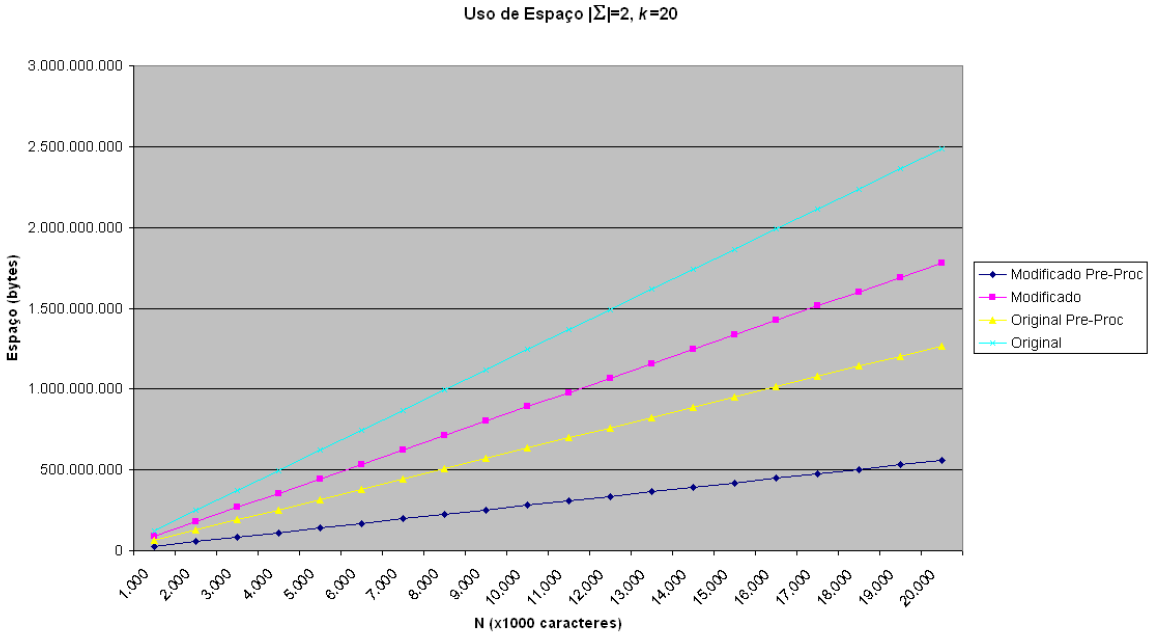


Figura 13: Gráfico de uso de espaço $|\Sigma| = 2$, $k = 20$

tempo de pré-processamento é bem maior na versão baseada em árvores de sufixos que na versão baseada em arranjos de sufixos, e para $k = 20$ fez com que o tempo de processador do algoritmo original fosse maior que o do modificado.

As tabelas 6, 7, 8, 9, 10, 11, 12 e 13 que descrevem o comportamento do algoritmo com relação a tempo de execução e de processador estão estruturadas da seguinte forma:

- Uma coluna descrevendo o tamanho do problema (N) para a execução do algoritmo.
- Duas colunas descrevendo o tempo para a implementação baseada em arranjos de sufixos e duas para a baseada em árvores de sufixos, nessa ordem, sendo uma coluna para a fase de pré-processamento e uma coluna para a execução total do algoritmo.
- Duas colunas com a diferença percentual do tempo da implementação baseada em árvores de sufixos e da implementação baseada em arranjos de sufixos, para o pré-processamento e o algoritmo completo, sendo que um valor positivo em uma linha indica que o tempo de execução para a versão baseada em árvores de sufixos foi maior.

Analisando as tabelas com os resultados de tempo de execução podemos fazer duas observações interessantes:

Tabela 6: Tempo do Processador para $|\Sigma| = 93$, $k = 20$

	Arranjo de Sufixos		Árvore de sufixos		Diferença	
N (×1000)	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**
1.000	0,606	3,440	4,460	7,246	86%	53%
2.000	1,380	7,132	13,410	19,022	90%	63%
3.000	2,178	10,858	23,066	31,494	91%	66%
4.000	3,010	14,612	32,540	43,790	91%	67%
5.000	3,820	18,394	41,890	55,976	91%	67%
6.000	4,602	22,086	51,134	68,034	91%	68%
7.000	5,410	25,824	60,560	80,248	91%	68%
8.000	6,298	29,714	70,030	92,652	91%	68%
9.000	7,082	33,398	80,020	105,352	91%	68%
10.000	7,928	38,318	89,730	117,806	91%	67%
11.000	10,453	44,127	99,773	131,660	90%	66%
12.000	9,620	45,523	109,787	144,953	91%	69%
13.000	10,567	50,343	119,863	157,993	91%	68%
14.000	11,403	53,230	131,540	172,010	91%	69%
15.000	12,183	56,917	140,597	184,013	91%	69%
16.000	13,080	61,817	151,527	199,650	91%	69%
17.000	14,010	68,957	162,197	217,063	91%	68%
18.000	14,887	75,177	176,477	233,283	92%	68%
19.000	15,653	80,423	184,423	246,800	92%	67%
20.000	16,680	84,923	195,453	260,883	91%	67%

Tabela 7: Tempo de Execução para $|\Sigma| = 93$, $k = 20$

	Arranjo de Sufixos		Árvore de sufixos		Diferença	
N (×1000)	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**
1.000	0,608	3,442	4,465	7,252	86%	53%
2.000	1,385	7,137	13,412	19,028	90%	62%
3.000	2,183	10,863	23,070	31,499	91%	66%
4.000	3,014	14,616	32,543	43,793	91%	67%
5.000	3,826	18,400	41,896	55,980	91%	67%
6.000	4,604	22,088	51,139	68,040	91%	68%
7.000	5,412	25,826	60,563	80,251	91%	68%
8.000	6,301	29,717	70,032	92,656	91%	68%
9.000	7,084	33,400	80,026	105,354	91%	68%
10.000	7,931	38,321	89,733	117,808	91%	67%
11.000	10,456	44,337	99,785	131,679	90%	66%
12.000	9,628	45,531	109,801	144,968	91%	69%
13.000	10,569	50,352	119,876	158,005	91%	68%
14.000	11,406	53,233	131,553	172,024	91%	69%
15.000	12,181	56,922	140,610	184,027	91%	69%
16.000	13,087	62,257	151,534	205,667	91%	70%
17.000	14,008	93,450	162,391	271,528	91%	66%
18.000	14,892	178,667	176,489	335,989	92%	47%
19.000	15,661	262,197	184,640	414,900	92%	37%
20.000	16,681	424,762	195,468	545,217	91%	22%

Tabela 8: Tempo do Processador para $|\Sigma| = 26$, $k = 20$

	Arranjo de Sufixos		Árvore de sufixos		Diferença	
N (×1000)	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**
1.000	0,636	4,120	2,432	5,678	74%	27%
2.000	1,410	8,570	5,948	12,468	76%	31%
3.000	2,230	13,086	9,542	19,356	77%	32%
4.000	3,120	17,706	13,252	26,402	76%	33%
5.000	3,994	22,274	17,244	33,754	77%	34%
6.000	4,824	26,884	21,292	41,148	77%	35%
7.000	5,740	31,732	25,582	49,062	78%	35%
8.000	6,714	36,484	30,056	56,838	78%	36%
9.000	7,574	41,218	34,610	64,914	78%	37%
10.000	8,512	45,790	39,298	72,932	78%	37%
11.000	11,163	53,497	44,110	82,143	75%	35%
12.000	10,403	55,610	48,980	90,517	79%	39%
13.000	11,497	61,163	53,927	99,307	79%	38%
14.000	12,417	65,897	59,013	107,397	79%	39%
15.000	13,293	70,723	64,090	117,043	79%	40%
16.000	14,277	76,160	69,337	133,700	79%	43%
17.000	15,370	83,627	74,687	146,910	79%	43%
18.000	16,387	105,817	79,923	160,007	79%	34%
19.000	17,190	105,137	85,327	166,317	80%	37%
20.000	18,387	115,130	91,060	184,693	80%	38%

Tabela 9: Tempo de Execução para $|\Sigma| = 26$, $k = 20$

	Arranjo de Sufixos		Árvore de sufixos		Diferença	
N (×1000)	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**
1.000	0,638	4,122	2,435	5,681	74%	27%
2.000	1,414	8,574	5,952	12,472	76%	31%
3.000	2,233	13,089	9,545	19,359	77%	32%
4.000	3,124	17,710	13,256	26,406	76%	33%
5.000	3,998	22,278	17,247	33,757	77%	34%
6.000	4,826	26,886	21,295	41,151	77%	35%
7.000	5,742	31,734	25,584	49,064	78%	35%
8.000	6,717	36,487	30,060	56,842	78%	36%
9.000	7,577	41,221	34,612	64,916	78%	37%
10.000	8,514	45,792	39,301	72,935	78%	37%
11.000	11,164	53,503	44,118	82,156	75%	35%
12.000	10,407	55,616	48,987	90,530	79%	39%
13.000	11,506	61,172	53,936	99,315	79%	38%
14.000	12,421	65,900	59,019	107,409	79%	39%
15.000	13,296	70,728	64,094	119,189	79%	41%
16.000	14,495	76,388	69,343	184,856	79%	59%
17.000	15,381	101,361	74,697	310,134	79%	67%
18.000	16,390	270,983	79,931	409,047	79%	34%
19.000	17,196	641,710	85,333	728,444	80%	12%
20.000	18,388	1.031,485	91,071	1.219,468	80%	15%

Tabela 10: Tempo do Processador para $|\Sigma| = 4$, $k = 20$

	Arranjo de Sufixos		Árvore de sufixos		Diferença	
N (×1000)	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**
1.000	**0,652**	**9,756**	**1,538**	**8,638**	**58%**	**-13%**
2.000	**1,452**	**20,344**	**3,418**	**17,874**	**58%**	**-14%**
3.000	**2,324**	**30,928**	**5,398**	**27,032**	**57%**	**-14%**
4.000	**3,274**	**41,912**	**7,418**	**36,430**	**56%**	**-15%**
5.000	**4,200**	**52,774**	**9,536**	**46,318**	**56%**	**-14%**
6.000	**5,102**	**63,934**	**11,512**	**55,592**	**56%**	**-15%**
7.000	**6,038**	**74,856**	**13,612**	**65,850**	**56%**	**-14%**
8.000	**7,022**	**86,926**	**15,748**	**75,970**	**55%**	**-14%**
9.000	**7,898**	**98,840**	**17,906**	**84,752**	**56%**	**-17%**
10.000	**8,834**	**108,832**	**20,094**	**94,704**	**56%**	**-15%**
11.000	**11,480**	**118,933**	**22,270**	**101,780**	**48%**	**-17%**
12.000	**10,787**	**124,067**	**24,513**	**110,613**	**56%**	**-12%**
13.000	**11,843**	**135,643**	**26,707**	**120,673**	**56%**	**-12%**
14.000	**12,810**	**149,580**	**29,030**	**130,683**	**56%**	**-14%**
15.000	**13,693**	**157,630**	**31,240**	**144,953**	**56%**	**-9%**
16.000	**14,650**	**170,177**	**33,537**	**160,777**	**56%**	**-6%**
17.000	**15,813**	**184,343**	**35,897**	**175,693**	**56%**	**-5%**
18.000	**16,790**	**208,327**	**38,360**	**187,250**	**56%**	**-11%**
19.000	**17,603**	**221,323**	**40,573**	**215,793**	**57%**	**-3%**
20.000	18,770	304,000	43,010	309,043	56%	2%

Tabela 11: Tempo de Execução para $|\Sigma| = 4$, $k = 20$

	Arranjo de Sufixos		Árvore de sufixos		Diferença	
N (×1000)	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**	**Pre-Proc**	**Total**
1.000	0,655	9,759	11,296	18,396	94%	47%
2.000	1,454	20,346	23,764	38,220	94%	47%
3.000	2,327	30,931	36,328	57,962	94%	47%
4.000	3,276	41,914	49,334	78,346	93%	47%
5.000	4,204	52,778	62,312	99,094	93%	47%
6.000	5,104	63,936	75,450	119,530	93%	47%
7.000	6,040	74,858	88,470	140,708	93%	47%
8.000	7,025	86,929	102,676	162,898	93%	47%
9.000	7,900	98,842	116,749	183,595	93%	46%
10.000	8,837	108,835	128,928	203,538	93%	47%
11.000	**11,487**	**118,948**	**22,279**	**101,799**	**48%**	**-17%**
12.000	**10,786**	**124,074**	**24,527**	**110,624**	**56%**	**-12%**
13.000	**11,849**	**135,655**	**26,714**	**120,686**	**56%**	**-12%**
14.000	**12,813**	**149,591**	**29,041**	**131,119**	**56%**	**-14%**
15.000	13,701	157,644	31,244	201,778	56%	22%
16.000	14,658	170,194	33,537	345,019	56%	51%
17.000	15,819	200,034	35,899	570,844	56%	65%
18.000	16,797	611,849	38,364	894,459	56%	32%
19.000	17,607	887,734	40,576	1.928,291	57%	54%
20.000	18,772	5.455,711	43,017	11.437,399	56%	52%

Tabela 12: Tempo do Processador para $|\Sigma| = 2$, $k = 20$

	Arranjo de Sufixos		Árvore de sufixos		Diferença	
N (×1000)	Pre-Proc	Total	Pre-Proc	Total	Pre-Proc	Total
1.000	**0,605**	**14,629**	**1,572**	**12,337**	**62%**	**-19%**
2.000	**1,372**	**30,982**	**3,352**	**25,538**	**59%**	**-21%**
3.000	**2,176**	**47,998**	**5,192**	**38,682**	**58%**	**-24%**
4.000	**3,064**	**64,056**	**7,040**	**52,084**	**56%**	**-23%**
5.000	**3,918**	**80,202**	**8,932**	**66,112**	**56%**	**-21%**
6.000	**4,728**	**98,122**	**10,884**	**79,478**	**57%**	**-23%**
7.000	**5,608**	**115,474**	**12,772**	**94,630**	**56%**	**-22%**
8.000	**6,510**	**132,102**	**14,752**	**107,136**	**56%**	**-23%**
9.000	**7,314**	**151,738**	**16,752**	**123,754**	**56%**	**-23%**
10.000	**8,200**	**168,560**	**18,764**	**139,736**	**56%**	**-21%**
11.000	**10,747**	**194,140**	**20,877**	**155,527**	**49%**	**-25%**
12.000	**9,953**	**202,100**	**22,963**	**165,077**	**57%**	**-22%**
13.000	**10,917**	**219,023**	**25,140**	**181,083**	**57%**	**-21%**
14.000	**11,823**	**239,107**	**27,220**	**201,173**	**57%**	**-19%**
15.000	**12,630**	**256,097**	**29,247**	**223,907**	**57%**	**-14%**
16.000	**13,487**	**275,207**	**31,450**	**244,030**	**57%**	**-13%**
17.000	**14,550**	**299,187**	**33,657**	**260,900**	**57%**	**-15%**
18.000	**15,477**	**333,623**	**35,933**	**291,047**	**57%**	**-15%**
19.000	16,237	357,973	38,090	367,090	57%	2%
20.000	**17,313**	**452,457**	**40,257**	**434,663**	**57%**	**-4%**

Tabela 13: Tempo de Execução para $|\Sigma| = 2$, $k = 20$

	Arranjo de Sufixos		Árvore de sufixos		Diferença	
N (×1000)	Pre-Proc	Total	Pre-Proc	Total	Pre-Proc	Total
1.000	**0,612**	**14,637**	**1,577**	**12,341**	**61%**	**-19%**
2.000	**1,376**	**30,984**	**3,353**	**25,543**	**59%**	**-21%**
3.000	**2,181**	**48,003**	**5,192**	**38,682**	**58%**	**-24%**
4.000	**3,063**	**64,061**	**7,042**	**52,085**	**57%**	**-23%**
5.000	**3,917**	**80,207**	**8,932**	**66,238**	**56%**	**-21%**
6.000	**4,736**	**98,134**	**10,890**	**79,617**	**57%**	**-23%**
7.000	**5,615**	**115,491**	**12,780**	**94,640**	**56%**	**-22%**
8.000	**6,514**	**132,115**	**14,755**	**107,146**	**56%**	**-23%**
9.000	**7,316**	**151,747**	**16,755**	**123,763**	**56%**	**-23%**
10.000	**8,200**	**168,573**	**18,765**	**139,744**	**56%**	**-21%**
11.000	**10,748**	**194,160**	**20,882**	**155,547**	**49%**	**-25%**
12.000	**9,955**	**202,315**	**22,972**	**165,096**	**57%**	**-23%**
13.000	**10,919**	**219,044**	**25,146**	**181,125**	**57%**	**-21%**
14.000	11,826	239,129	27,222	334,518	57%	29%
15.000	12,630	256,298	29,250	347,950	57%	26%
16.000	13,496	275,230	31,454	579,657	57%	53%
17.000	14,550	308,850	33,662	852,980	57%	64%
18.000	15,482	644,768	35,937	1.531,532	57%	58%
19.000	16,238	1.024,395	38,095	7.525,399	57%	86%
20.000	17,314	5.822,213	40,268	13.809,782	57%	58%

(*i*) A fase de iteração da versão baseada em árvores de sufixos é em geral mais rápida que a versão baseada em arranjos de sufixos.

(*ii*) Quando $|\Sigma|$ é grande o tempo de pré-processamento é muito menor na versão baseada em arranjos de sufixos, e pode influenciar fortemente no tempo total de execução.

Nossa análise indica que a observação (*i*) acima é causada pela necessidade de fazer consultas ao índice reverso (que mapeia os sufixos P_j e T_k na sua posição no arranjo Pos), para podermos identificar os nós da árvore cartesiana cujo LCA será consultado. Esta uma operação não é necessária na versão baseada em árvores de sufixos, e, além disso, como as posições de Pos que serão consultadas não são facilmente previstas a partir de j e k, e provavelmente causam erros na consulta à memória cache e uma busca da memória RAM.

A observação (*ii*) acima é conseqüência direta da implementação de árvore de sufixos utilizada e da freqüência de distribuição aleatória dos caracteres, pois o tipo de árvore de sufixos construída é a variante ILLI descrita em (13), que usa uma lista ligada para guardar as arestas dos filhos de cada nó. Como $|\Sigma|$ é grande, as buscas nas arestas para identificar se já existe uma aresta cujo rótulo começa com determinado caractere geram uma busca linear $O(|\Sigma|)$. Para resolver isso, seria necessário usar a variante IHTI (13) baseada em tabelas de *hash*, mas a implementação que obtivemos suporta apenas a construção de árvores do tipo ILLI. De qualquer forma, o nosso foco é a economia de espaço e segundo Kurtz(13) a versão IHTI usa mais espaço que a versão ILLI. Por exemplo, para o genoma da bactéria *E. coli* a versão ILLI usa 12,56 bytes por caractere, enquanto a versão IHTI usa 17,14 bytes por caractere.

Percebemos que aumentar a quantidade de diferenças permitidas aumenta o custo da fase de iteração, o que diminui a vantagem que o algoritmo modificado poderia ter por uma fase de pré-processamento mais rápida. Assim, na medida em que k aumente, a tendência é que o custo da fase de iteração domine a execução total do algoritmo, minimizando a vantagem que o algoritmo modificado tem na fase de pré-processamento.

Com respeito ao uso de espaço, o algoritmo modificado usa menos espaço em todos os casos. Na versão com espaço $\theta(kn)$ do algoritmo, se aumentamos o valor de k também aumentamos o uso de espaço da fase de iteração e isso diminui a vantagem de uso de espaço do algoritmo modificado, pois para k grande o espaço utilizado na fase de iteração será bem maior que o utilizado na fase de pré-processamento. A versão com espaço $\theta(n)$

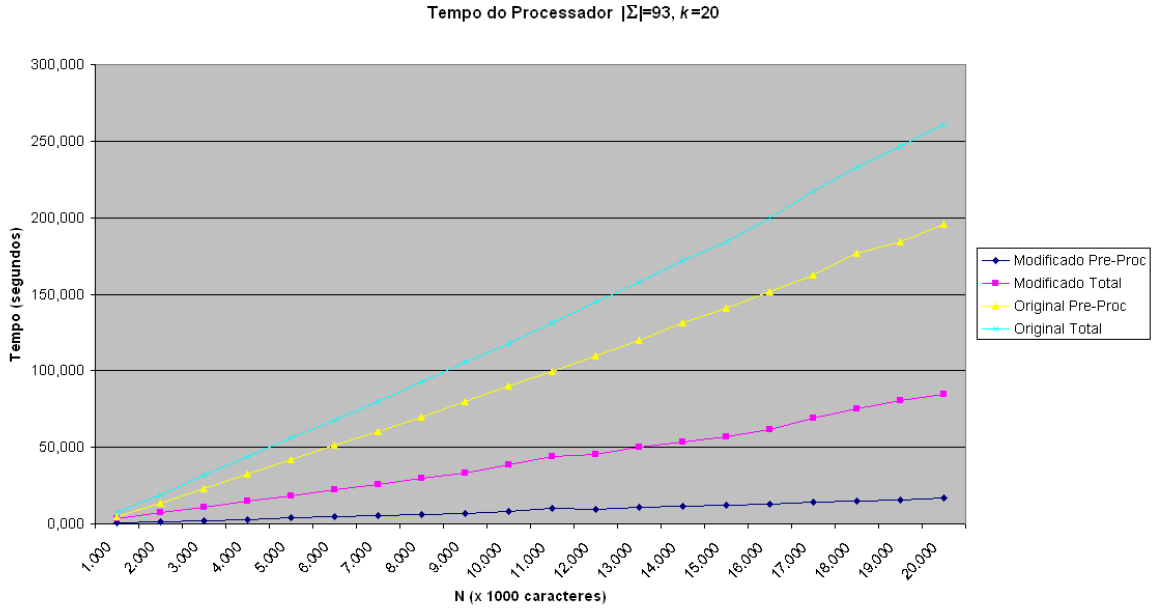


Figura 14: Gráfico de uso de processador $|\Sigma| = 93$, $k = 20$

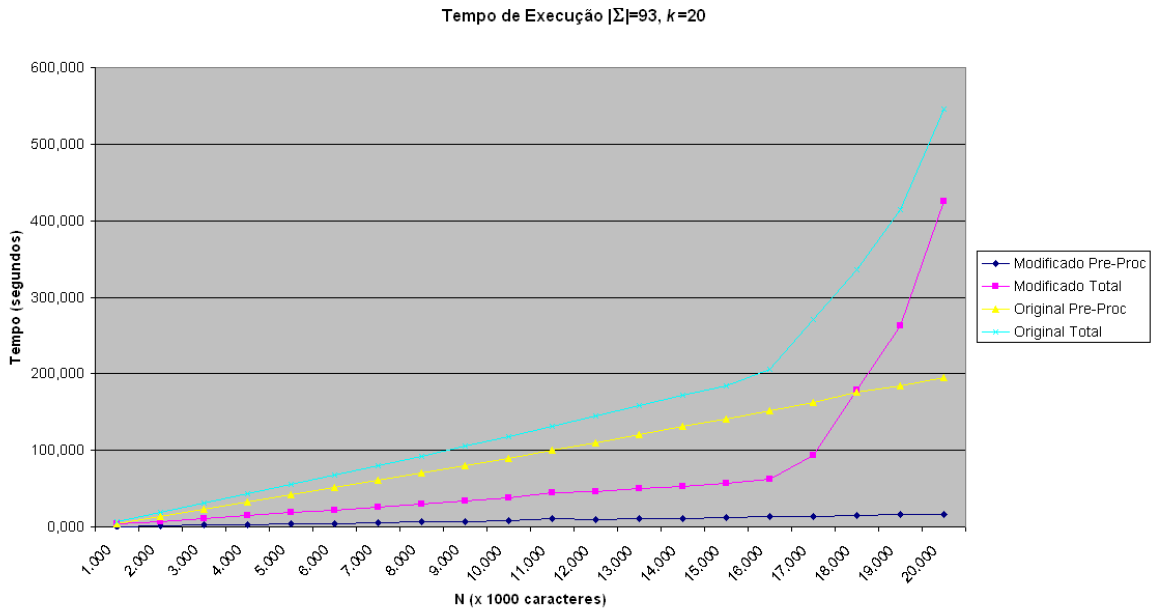


Figura 15: Gráfico de tempo de execução $|\Sigma| = 93$, $k = 20$

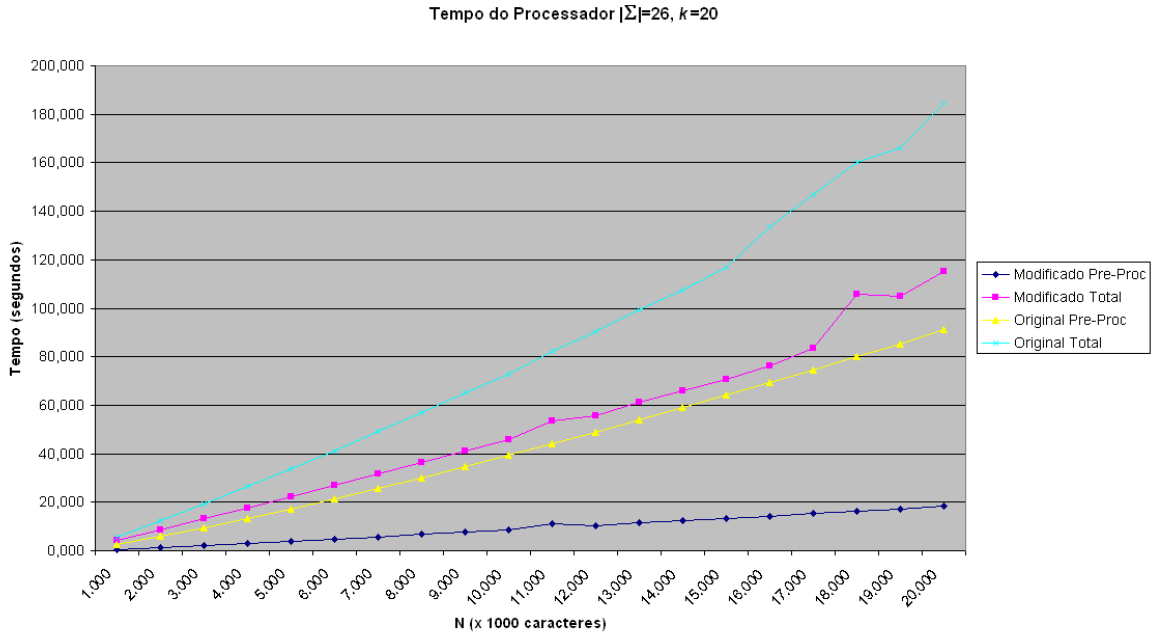


Figura 16: Gráfico de uso de processador $|\Sigma| = 26$, $k = 20$

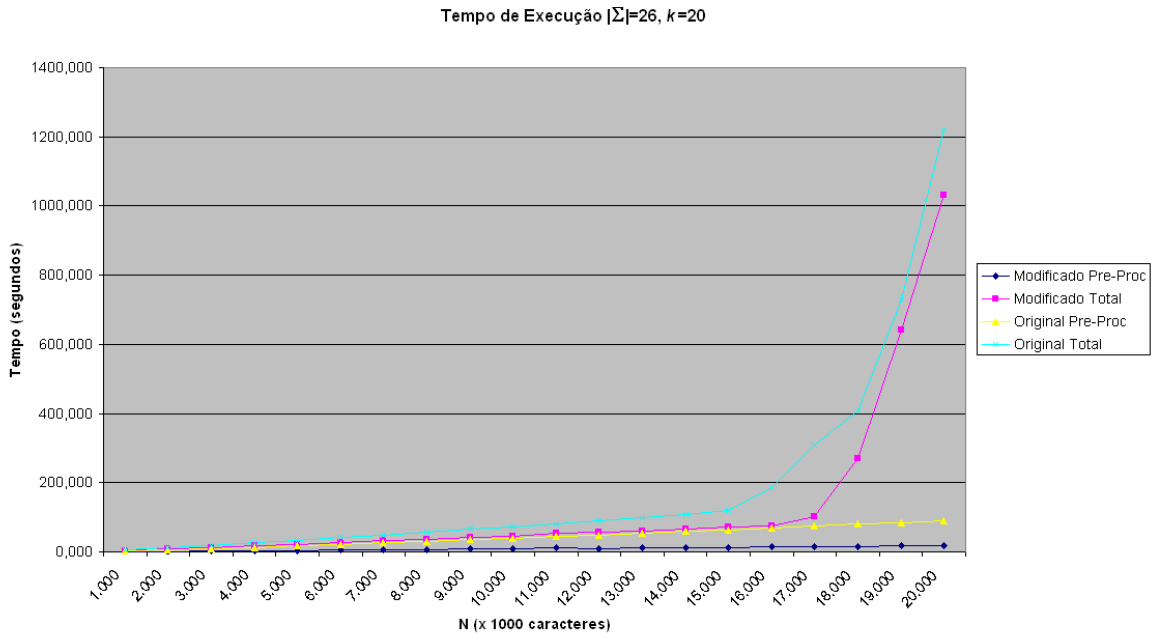


Figura 17: Gráfico de tempo de execução $|\Sigma| = 26$, $k = 20$

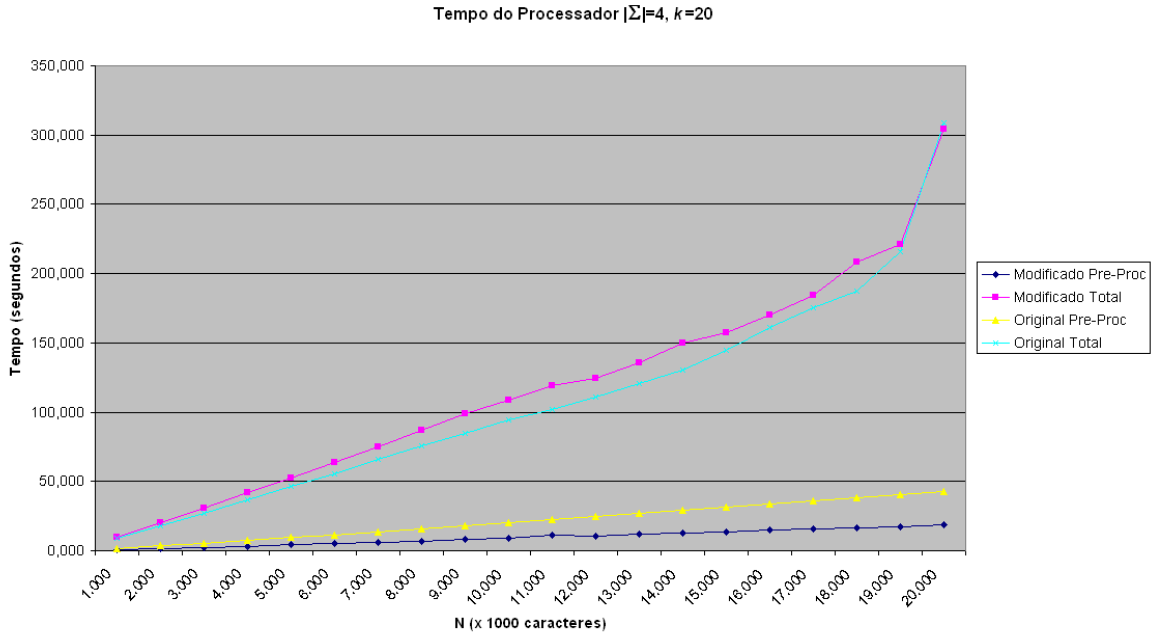


Figura 18: Gráfico de uso de processador $|\Sigma| = 4$, $k = 20$

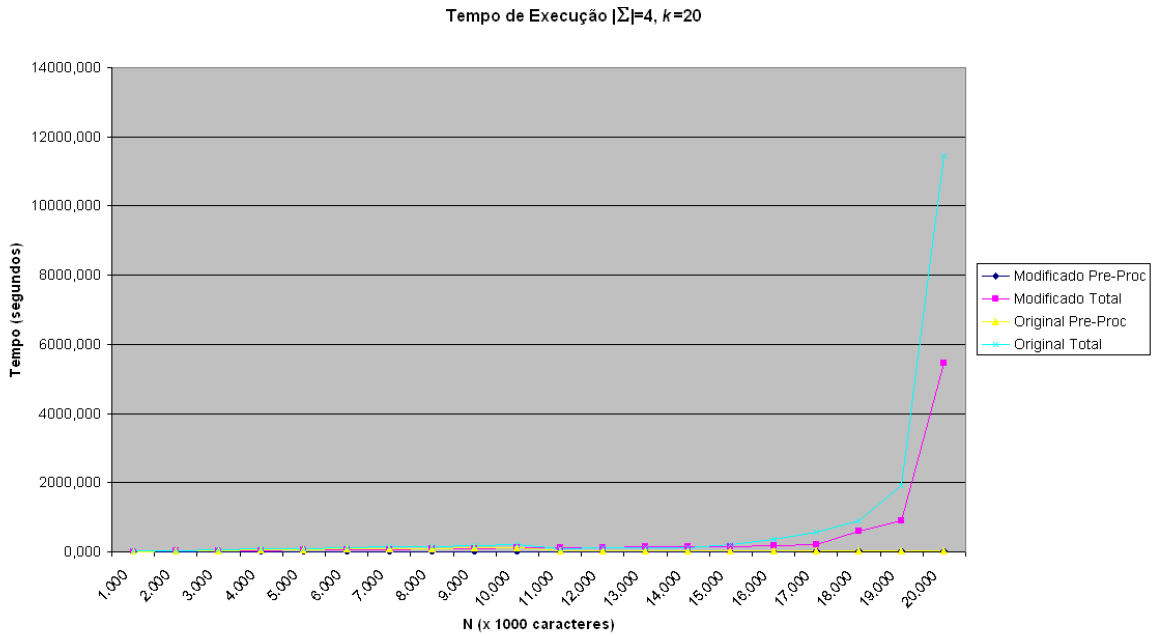


Figura 19: Gráfico de tempo de execução $|\Sigma| = 4$, $k = 20$

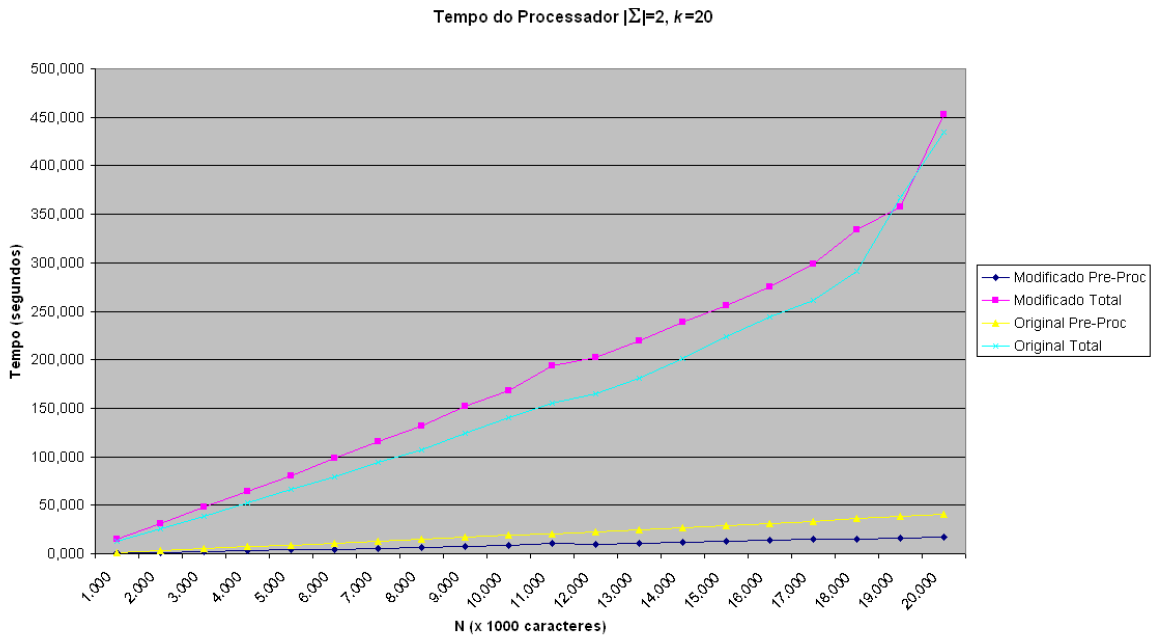


Figura 20: Gráfico de uso de processador $|\Sigma| = 2$, $k = 20$

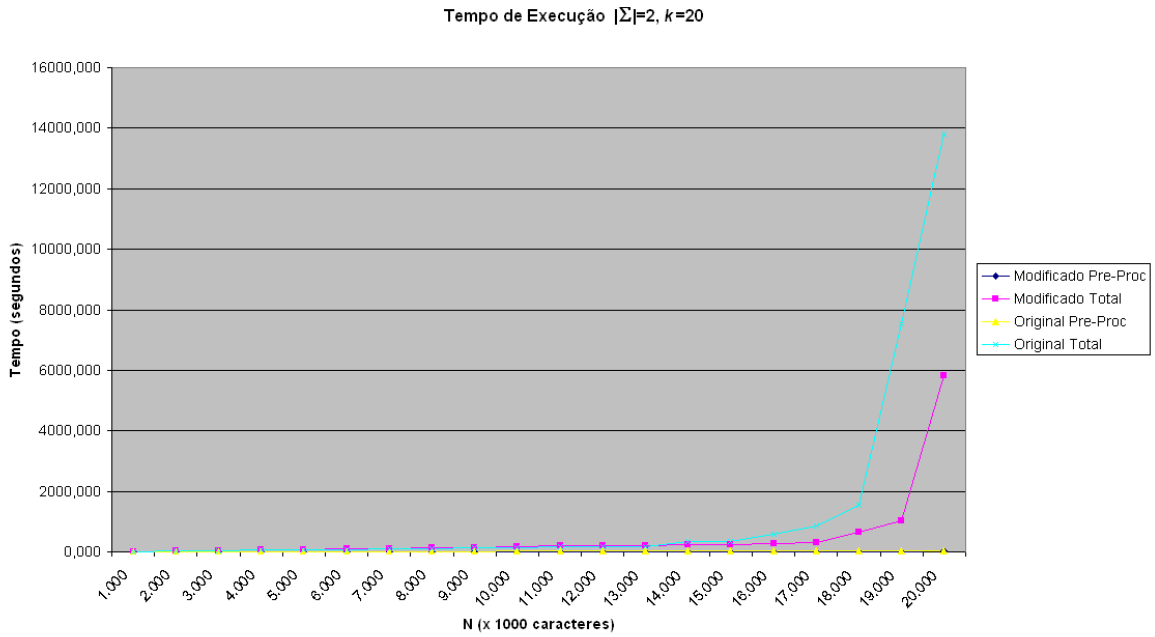


Figura 21: Gráfico de tempo de execução $|\Sigma| = 2$, $k = 20$

do algoritmo modificado sempre usará menos espaço independente de k, mas necessitará de uma fase adicional para construir os alinhamentos das palavras.

5.2.3 Dados reais

Para a avaliação do comportamento do algoritmo com dados reais, usamos seqüências do Genbank e textos do projeto Gutenberg, acessíveis pela internet. Dividimos a avaliação em duas partes.

Na primeira parte avaliamos o comportamento das variações do algoritmo com árvores e arranjos de sufixos, utilizando dados reais retirados do Genbank e do projeto Gutenberg. São palavras que cabem completamente na memória principal do computador utilizado nos experimentos e servem somente para validar as observações de uso de espaço e tempo feitas sobre os experimentos com dados aleatórios.

Na segunda parte avaliamos o uso do algoritmo com problemas de dimensões maiores, que precisam ser tratados pela variante que usa espaço $\theta(n)$ por usarem massas de dados muito grandes e exigirem valores grandes para k.

Como dados reais utilizamos as seguinte seqüências do Genbank:

- *Halo.* – *Halobacterium* NRC-1 plasmídeo pNRC100, seqüência completa.
- *E. rumin.* – *Ehrlichia ruminantium* str. Welgevonden, genoma completo.
- *Influenza C*, vírus, segmento 5, seqüência completa (usado como padrão).
- *Coliphage phiX174*, vírus, genoma completa (usado como padrão).

Além disso, usamos os seguintes textos retirados do projeto Gutenberg e do *corpus* de Canterbury:

- *Fifteen* – Fifteen Thousand Useful Phrases, por Greenville Kleiser (*EBook #18362* do Projeto Gutenberg).
- *Moby Dick* – Moby Dick, por Herman Melville (*Etext #2701* do Projeto Gutenberg)
- *Shakespeare* – Compilação de obras de William Shakespeare (*Etexts #2253, #1103, #1107, #1113, #1114, #1127, #1794, #1129, #1135, #1522 e #1524* do Projeto Gutenberg)

Tabela 14: Uso de espaço para dados reais

			Arranjo de Sufixos				Árvore de Sufixos				Economia de Espaço			
			PP		Total		PP		Total		MB	Bpc	PP	Total
	|Σ|	N (KB)	MB	Bpc.	MB	Bpc	MB	Bpc	MB	Bpc				
Halo.	4	191	5	29	15	79	10	53	19	104	5	25	46,38%	23,68%
E. rumin.	4	1.516	42	28	117	79	78	53	154	104	36	25	46,70%	23,72%
E. coli	4	4.639	127	28	358	79	236	52	467	103	109	24	46,13%	23,29%
Biblia	62	4.047	111	28	312	79	185	47	387	98	75	19	40,32%	19,32%
Fifteen	85	589	16	28	45	79	27	46	56	97	10	18	39,33%	18,69%
Moby Dick	90	1.256	34	28	97	79	56	46	119	97	22	18	38,88%	18,40%
Shakespeare	93	1.875	51	28	145	79	84	46	178	97	33	18	39,10%	18,54%
world192	93	2.473	68	28	191	79	112	46	235	97	44	18	39,36%	18,70%

Tabela 15: Tempo de execução para dados reais

			Arranjo de Sufixos		Árvore de sufixos		Diferença	
	|Σ|	N (KB)	Pre-Proc	Total	Pre-Proc	Total	Pre-Proc	Total
Halo.	4	191	0,080	0,753	0,185	0,790	57%	4,7%
E. rumin.	4	1.516	1,030	7,820	2,438	7,810	58%	-0,1%
E. coli	4	4.639	3,875	25,605	8,595	25,340	55%	-1,0%
Biblia	62	4.047	3,070	13,000	6,847	15,503	55%	16,1%
Fifteen	85	589	0,310	1,480	0,713	1,823	57%	18,8%
Moby Dick	90	1.256	0,830	3,370	2,167	4,550	62%	25,9%
Shakespeare	93	1.875	1,310	5,137	3,227	6,767	59%	24,1%
world192	93	2.473	1,790	7,277	3,677	8,570	51%	15,1%

- *Bíblia* – Bíblia do Rei Jaime (do arquivo *large* do *corpus* de Canterbury).
- *E.coli* – Genoma da E. coli (do arquivo *large* do *corpus* de Canterbury).
- *world192 – CIA World Factbook* de 1992 (do arquivo *large* do *corpus* de Canterbury).

Nas tabela 14 temos a comparação do uso de espaço para o algoritmo que usa a construção de árvore de sufixos no seu pré-processamento e o algoritmo que usa arranjos de sufixos para seu pré-processamento, e na tabela 15 temos a comparação do tempo de execução. Como em todos os casos as informações cabiam completamente na memória principal, não houve diferença significativa entre tempo de execução e tempo de processador e apresentamos apenas o tempo de processador. Essas tabelas têm estrutura similar a das tabelas de resultados para dados aleatórios (apenas acrescentamos uma identificação da seqüência e o tamanho de Σ para cada experimento).

Verificamos que o comportamento é similar ao dos dados gerados de forma aleatória, e as mesmas considerações se aplicam. Em todos os casos, o pré-processamento é mais rápido no algoritmo modificado, e a fase de iteração do algoritmo original é mais rápida. Para alfabetos pequenos (DNA), a vantagem no pré-processamento do algoritmo modificado é pequena o suficiente para não se converter em vantagem no tempo total de execução do algoritmo, mas para os casos de alfabetos grandes, a diferença foi suficiente para que o desempenho total do algoritmo modificado fosse superior. Em todos os casos, o uso de espaço da versão baseada em arranjos de sufixos foi menor, sendo que quando $|\Sigma|$ é

grande, a diferença de uso de espaço é menor e o tempo de execução é maior.

Observe que mesmo para alfabetos grandes a diferença no uso de espaço continua significativa, e maior que no caso dos dados aleatórios. Isso ocorre porque tanto textos de literatura quanto seqüências biológicas possuem uma certa estrutura, e a freqüência de cada caractere (ou de *codons*) não é uniforme, o que gera uma quantidade maior de nós na árvore de sufixos que no caso aleatório. Isso aumenta o espaço utilizado tanto para a própria árvore de sufixos como para a estrutura usada para o cálculo do LCA.

Sabemos que no computador utilizado para a análise experimental rapidamente chegamos ao ponto em que não é possível processar de forma realista palavras grandes, como as palavras que formariam todas as regiões codificadoras de um cromossomo. Isso acontece em parte porque qualquer resultado significativo exigiria um valor alto para k.

Ora, para realizar buscas *exatas* nessas palavras o caminho usual seria realizar o algoritmo de programação dinâmica em tempo linear, que marcaria todas as posições em T onde encontramos uma ocorrência de P compatíveis com o critério de busca, e então posteriormente construiríamos os respectivos alinhamentos usando tempo $\theta(m^2)$ e espaço $\theta(m)$, sendo que m é muito menor que n.

Vamos usar a mesma abordagem para estender o limite das palavras que podemos processar independente da quantidade de diferenças admitidas. Dessa forma o valor de k somente interferirá no tempo de execução do algoritmo, mas não no uso de espaço que será $\theta(n)$.

Para esse esperimento escolhemos como padrão regiões de *cDNA* do ser humano retirado do NCBI Genbank de comprimentos da ordem de 8, 9, e 11 mil caracteres (*H. sapiens adenomatosis polyposis coli*, *H. sapiens dystrophin* e *H. sapiens talin 1*).

Para texto usamos:

- *Drosophila melanogaster*, cDNA *release 5* de 17 de Abril de 2006, eucromatina e heterocroma:
  - Braço 2R (21 milhões de caracteres).
  - Braço 2L (23 milhões de caracteres).
  - Braço 3L (25 milhões de caracteres).
  - Braço 3R (28 milhões de caracteres).
  - Braço U Extra, (29 milhões de caracteres).

Tabela 16: Dados reais - uso de espaço para pesquisa em DNA

		Arranjo de Sufixos (MB)		Árvore de Sufixos (MB)		Economia de Espaço		
M (KB)	*N* (MB)	Pre-Proc	Total	Pre-Proc	Total	MB	Bpc	%
8	*21*	*592*	*1.438*	*1.102*	*1.948*	*510*	*24*	*26%*
9	*21*	*592*	*1.438*	*1.102*	*1.948*	*510*	*24*	*26%*
9	*21*	*592*	*1.438*	*1.102*	*1.948*	*510*	*24*	*26%*
11	*21*	*592*	*1.438*	*1.102*	*1.948*	*510*	*24*	*26%*
8	*23*	*645*	*1.565*	*1.200*	*2.121*	*556*	*24*	*26%*
9	*23*	*645*	*1.565*	*1.200*	*2.121*	*556*	*24*	*26%*
9	*23*	*645*	*1.565*	*1.200*	*2.121*	*556*	*24*	*26%*
11	*23*	*645*	*1.565*	*1.200*	*2.121*	*556*	*24*	*26%*
8	*25*	*687*	*1.669*	*1.280*	*2.262*	*593*	*24*	*26%*
9	*25*	*687*	*1.669*	*1.280*	*2.262*	*593*	*24*	*26%*
9	*25*	*687*	*1.669*	*1.280*	*2.262*	*593*	*24*	*26%*
11	*25*	*688*	*1.669*	*1.280*	*2.262*	*593*	*24*	*26%*
8	*28*	*782*	*1.898*	*1.451*	*2.567*	*669*	*24*	*26%*
9	*28*	*782*	*1.898*	*1.451*	*2.567*	*669*	*24*	*26%*
9	*28*	*782*	*1.898*	*1.451*	*2.567*	*669*	*24*	*26%*
11	*28*	*782*	*1.898*	*1.451*	*2.567*	*669*	*24*	*26%*
8	*29*	*812*	*1.973*	*1.637*	*2.797*	*825*	*28*	*29%*
9	*29*	*812*	*1.973*	*1.637*	*2.797*	*825*	*28*	*29%*
9	*29*	*812*	*1.973*	*1.637*	*2.797*	*825*	*28*	*29%*
11	*29*	*812*	*1.973*	*1.637*	*2.797*	*825*	*28*	*29%*
8	*35*	*968*	*2.350*	*1.818*	-	-	-	-
9	*35*	*968*	*2.350*	*1.818*	-	-	-	-
9	*35*	*968*	*2.350*	*1.818*	-	-	-	-
11	*35*	*968*	*2.350*	*1.818*	-	-	-	-

- *Homo sapiens*, cDNA do cromossomo 22 (34,5 milhões de caracteres)

Observe que a avaliação feita aqui é de caráter estritamente *algorítmico*, não tendo sido realizada buscando um significação biológica real. O alinhamento de seqüências biológicas se beneficia mais de um algoritmo de *maximização* de uma medida de similaridade, que leve em conta fatores como a probabilidade de mutação de uma base para outra, enquanto o algoritmo que estamos avaliando é um algoritmo de *minimização* cujo critério é a distância de edição.

Na tabela 16 apresentamos os resultados com relação ao uso de espaço e nas tabelas 17 e 18 os resultados com relação ao tempo de execução para $k = 50$ e $k = 100$, respectivamente (como usamos a variante com espaço linear do algoritmo, não há diferença no uso de espaço para diferentes valores de k).

Na figura 22 apresentamos um gráfico comparativo do uso de espaço dos algoritmos, baseado na tabela 16. Nas figuras 23 e 24 apresentamos de forma gráfica a comparação do tempo de processador para cada algoritmo, para $k = 50$ e $k = 100$, respectivamente. Nas figuras 25 e 26 apresentamos de forma gráfica a comparação do tempo de execução para cada algoritmo, para $k = 50$ e $k = 100$, respectivamente.

Note que o texto de 34,5 milhões de caracteres (cromossomo 22 do *H. sapiens*) não pôde ser processado pelo algoritmo original. O espaço adicional de 845 MB usado no pré-processamento do algoritmo foi decisivo para que a memória total utilizada superasse a capacidade de memória virtual do computador utilizado. Para o experimento de pesquisa

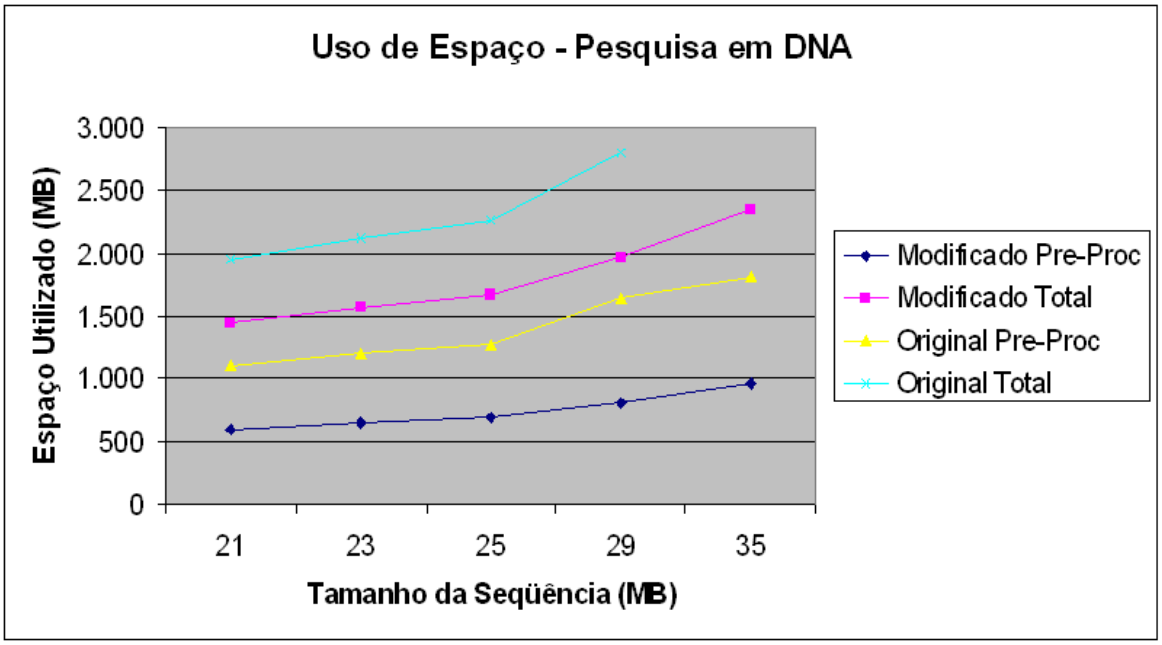


Figura 22: Gráfico de uso de espaço para pesquisa em DNA

Tabela 17: Dados reais – tempo de execução para pesquisa em DNA — $k = 50$

		Arranjo de Sufixos (seg)			Árvore de Sufixos (seg)			%	
M (KB)	N (MB)	PP CPU	Total	Total CPU	PP CPU	Total	Total CPU	Total	CPU
8	21	19,7350	428	428,4000	45,1400	360	359,9100	-19,0%	-19,0%
9	21	19,6350	508	507,5450	45,1100	408	408,3300	-24,3%	-24,3%
9	21	19,6300	506	505,8350	45,0100	408	407,7800	-24,0%	-24,0%
11	21	19,7000	476	476,3100	45,0050	387	386,9400	-23,1%	-23,1%
8	23	21,2300	458	458,2500	49,0550	428	389,8900	-7,2%	-17,5%
9	23	21,1550	557	556,7300	49,2300	485	446,5800	-14,7%	-24,7%
9	23	21,1850	555	554,8300	49,1600	484	445,9900	-14,7%	-24,4%
11	23	21,3900	525	525,0500	49,0400	466	425,2600	-12,6%	-23,5%
8	**25**	**23,0900**	**504**	**504,4050**	**52,8950**	**509**	**421,6750**	**0,9%**	**-19,6%**
9	25	23,0250	610	610,0750	52,7050	563	488,6500	-8,4%	-24,8%
9	25	23,0800	608	607,1250	52,7850	576	481,9650	-5,5%	-26,0%
11	25	23,3350	561	561,0250	52,8250	539	455,2300	-4,1%	-23,2%
8	**28**	**26,3950**	**598**	**590,2850**	**61,1900**	**648**	**482,2800**	**7,7%**	**-22,4%**
9	**28**	**26,4900**	**732**	**696,1950**	**61,2850**	**767**	**558,8350**	**4,5%**	**-24,6%**
9	**28**	**26,4550**	**701**	**693,6900**	**61,2950**	**814**	**565,2350**	**13,9%**	**-22,7%**
11	**28**	**26,4250**	**659**	**649,6550**	**61,1850**	**797**	**539,9100**	**17,3%**	**-20,3%**
8	**29**	**27,2950**	**597**	**541,7650**	**47,1750**	**613**	**454,3500**	**2,6%**	**-19,2%**
9	**29**	**27,6300**	**723**	**641,4150**	**47,2800**	**907**	**523,7300**	**20,3%**	**-22,5%**
9	29	27,5250	750	652,6300	47,0350	719	538,5300	-4,2%	-21,2%
11	29	28,0350	673	605,9550	47,3250	663	499,4850	-1,4%	-21,3%
8	**35**	**32,5450**	**1.894**	**798,6850**	**72,7300**	-	-	-	-
9	**35**	**32,1750**	**1.772**	**895,3400**	**72,7750**	-	-	-	-
9	**35**	**31,9550**	**1.680**	**875,1150**	**72,6750**	-	-	-	-
11	**35**	**31,9500**	**1.983**	**824,2750**	**72,9100**	-	-	-	-

Tabela 18: Dados reais – tempo de execução para pesquisa em DNA — $k = 100$

M (KB)	N (MB)	Arranjo de Sufixos (seg) PP CPU	Total	Total CPU	Árvore de Sufixos (seg) PP CPU	Total	Total CPU	% Total	CPU
8	*21*	*19,6600*	*913*	*912,5000*	*45,2250*	*717*	*716,7350*	*-27,3%*	*-27,3%*
9	*21*	*19,6300*	*1.008*	*1.008,0200*	*44,9550*	*782*	*782,0000*	*-28,9%*	*-28,9%*
9	*21*	*19,6450*	*1.007*	*1.007,1350*	*45,0800*	*778*	*778,0500*	*-29,4%*	*-29,4%*
11	*21*	*19,6300*	*960*	*960,3950*	*45,0050*	*750*	*750,1250*	*-28,0%*	*-28,0%*
8	*23*	*21,2800*	*991*	*991,1750*	*49,0400*	*826*	*776,9900*	*-19,9%*	*-27,6%*
9	*23*	*21,1850*	*1.103*	*1.102,2950*	*49,2050*	*913*	*852,5700*	*-20,8%*	*-29,3%*
9	*23*	*21,2000*	*1.104*	*1.104,0250*	*49,1900*	*905*	*853,3050*	*-22,0%*	*-29,4%*
11	*23*	*21,3900*	*1.068*	*1.068,0450*	*49,1450*	*876*	*825,0950*	*-21,9%*	*-29,4%*
8	*25*	*23,1100*	*1.075*	*1.074,7000*	*52,7550*	*1.018*	*839,6100*	*-5,6%*	*-28,0%*
9	*25*	*23,0750*	*1.201*	*1.201,3150*	*52,7200*	*1.082*	*922,3450*	*-11,1%*	*-30,2%*
9	*25*	*23,0500*	*1.201*	*1.200,2350*	*52,7400*	*1.097*	*920,8450*	*-9,5%*	*-30,3%*
11	*25*	*23,2900*	*1.134*	*1.134,1700*	*52,9200*	*1.068*	*891,3500*	*-6,2%*	*-27,2%*
8	*28*	*26,4150*	*1.275*	*1.261,4500*	*61,3100*	*1.209*	*978,6850*	*-5,4%*	*-28,9%*
9	*28*	*26,4800*	*1.415*	*1.398,1900*	*61,2600*	*1.299*	*1.073,7400*	*-9,0%*	*-30,2%*
9	*28*	*26,3900*	*1.398*	*1.379,8300*	*61,2850*	*1.262*	*1.057,6400*	*-10,7%*	*-30,5%*
11	*28*	*26,4150*	*1.316*	*1.310,7350*	*61,2300*	*1.277*	*1.031,5550*	*-3,1%*	*-27,1%*
8	*29*	*27,3050*	*1.185*	*1.119,4400*	*47,2400*	*1.446*	*887,8850*	*18,1%*	*-26,1%*
9	*29*	*27,5500*	*1.404*	*1.260,3400*	*47,0800*	*1.199*	*988,0650*	*-17,0%*	*-27,6%*
9	**29**	**27,5450**	**1.319**	**1.256,0500**	**47,2800**	**1.325**	**988,6050**	**0,4%**	**-27,1%**
11	*29*	*28,0600*	*1.272*	*1.201,1700*	*47,2200*	*1.241*	*952,5350*	*-2,5%*	*-26,1%*
8	**35**	**32,6500**	**2.512**	**1.588,9400**	**72,8300**	-	-	-	-
9	**35**	**31,9050**	**2.516**	**1.691,7050**	**73,1050**	-	-	-	-
9	**35**	**31,8950**	**2.696**	**1.705,4500**	**72,7450**	-	-	-	-
11	**35**	**31,9300**	**2.568**	**1.629,1200**	**72,7450**	-	-	-	-

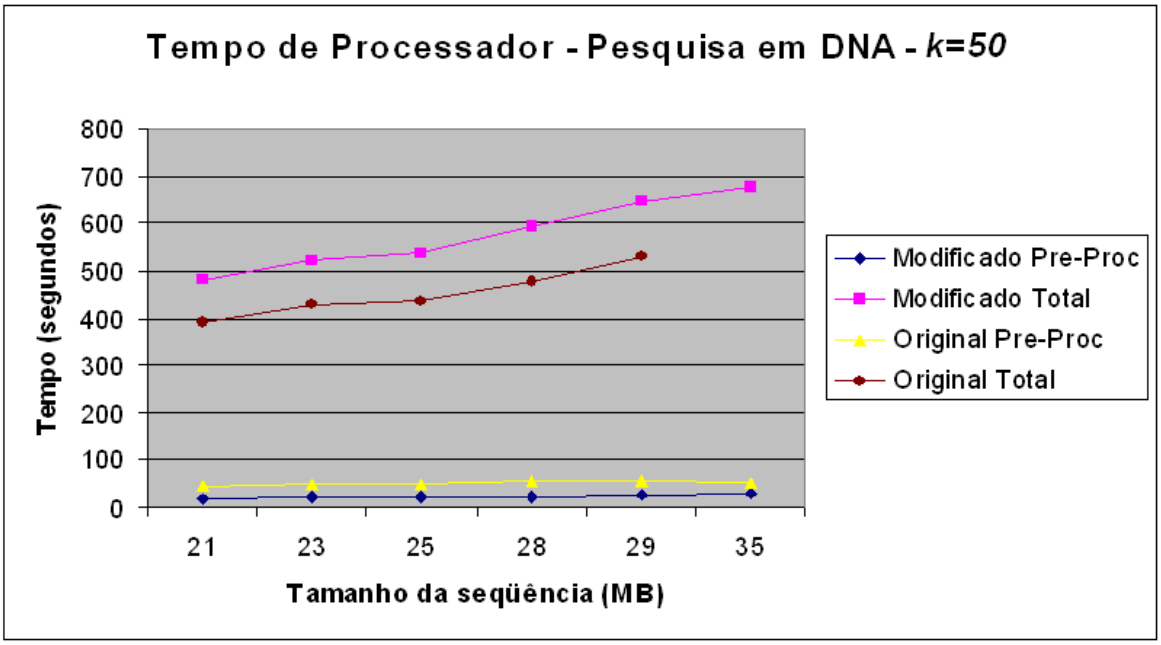


Figura 23: Gráfico de tempo de processador para pesquisa em DNA – $k = 50$

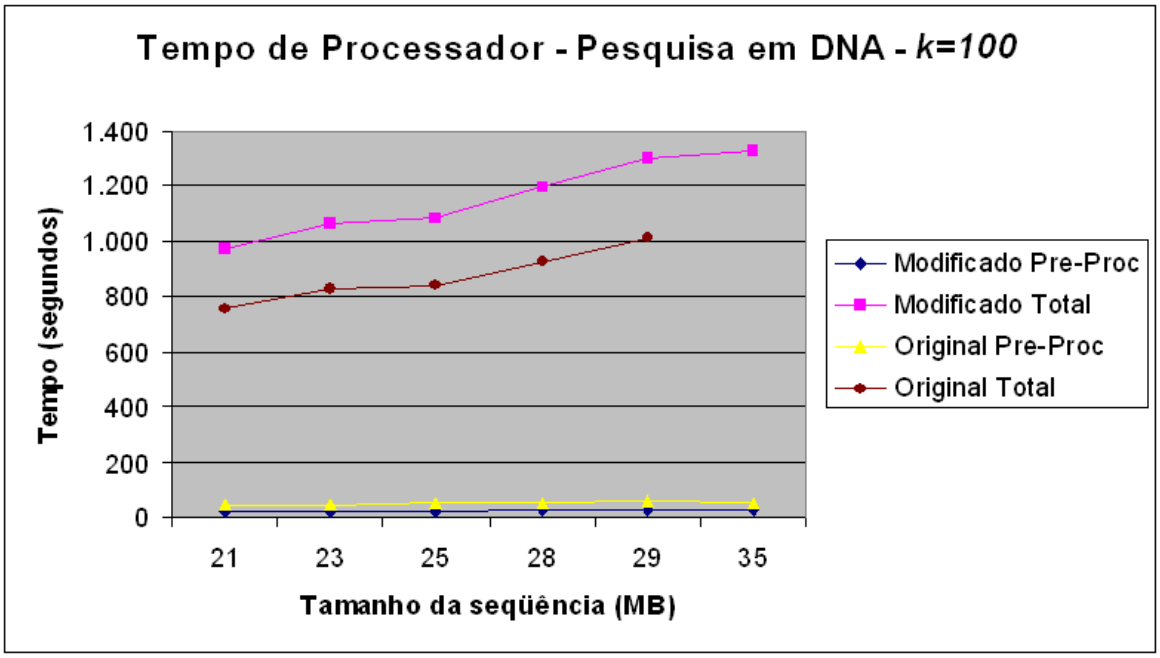


Figura 24: Gráfico de tempo de processador para pesquisa em DNA – $k = 100$

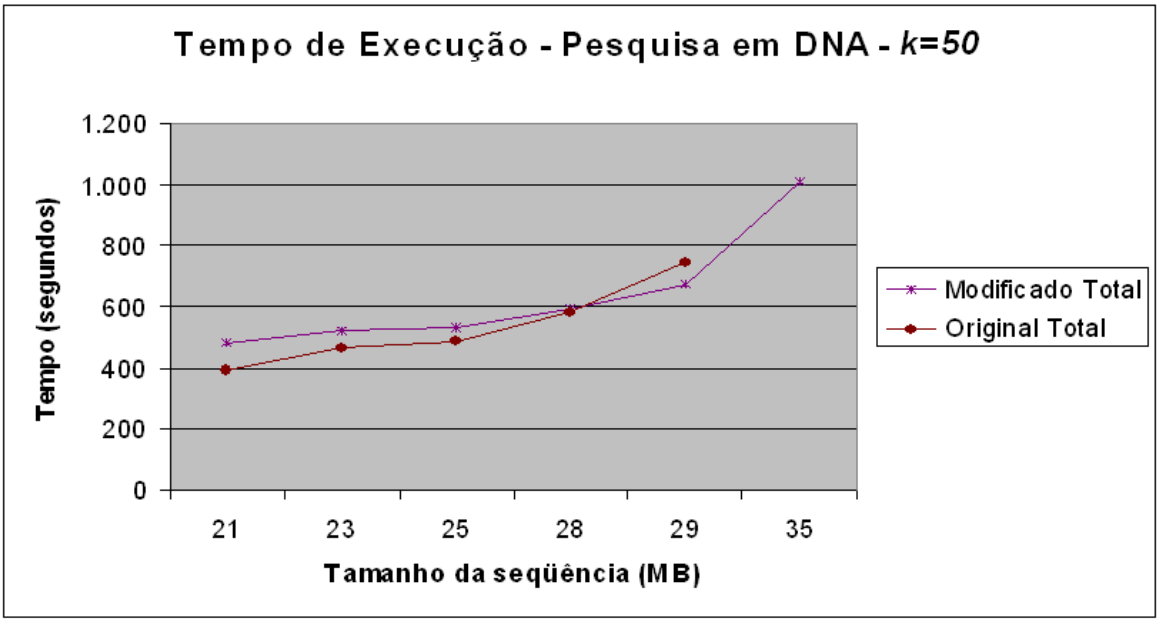


Figura 25: Gráfico de tempo de execução para pesquisa em DNA – $k = 50$

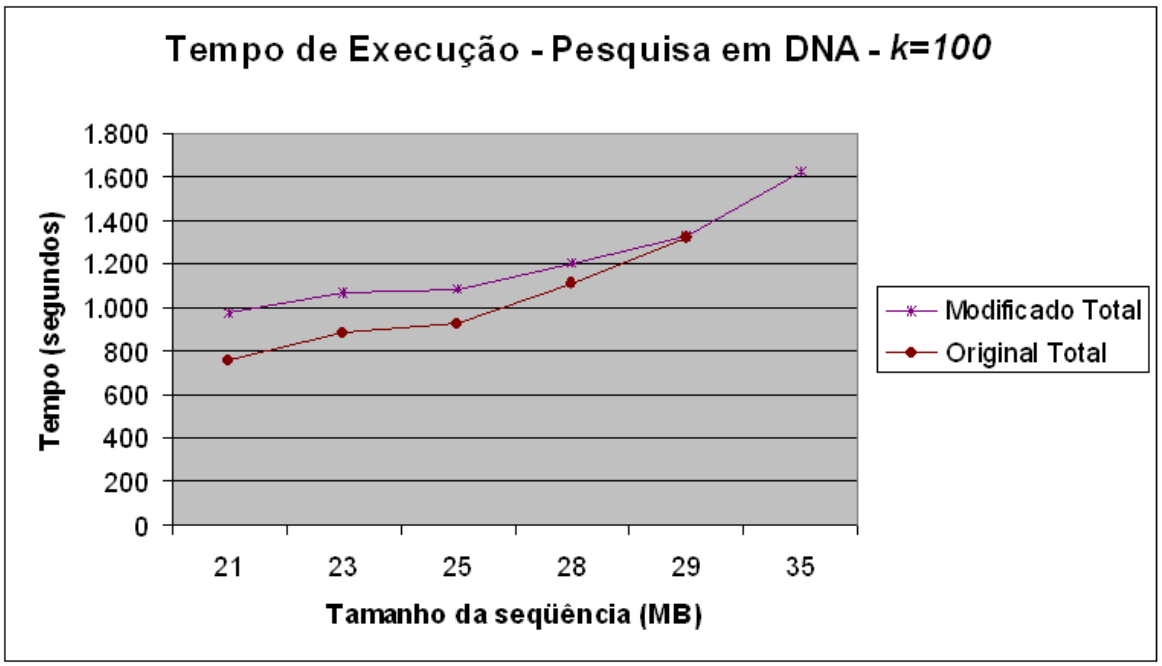


Figura 26: Gráfico de tempo de execução para pesquisa em DNA – $k = 100$

em DNA a economia média de espaço foi de 27%.

Com respeito ao tempo de execução, ao eliminarmos a tabela para manter a informação de construção dos alinhamentos (mantendo somente as posições iniciais e finais) ambas as versões do algoritmo se comportaram melhor na presença da memória virtual. Já comentamos na seção 5.2.2 que a fase de iteração baseada em árvore de sufixos é mais rápida. Essa diferença fica mais expressiva na medida em que aumentamos k, como podemos ver nas coluna que descrevem o tempo de processador utilizado (CPU). Apesar disso, quando a versão baseada em árvore de sufixos começa a usar a memória virtual a diferença de tempo real de execução passa a ser menor que a diferença do tempo de processador. Quando ambas as implementações estão usando memória virtual, as diferenças são mínimas e em vários casos a implementação baseada em arranjos de sufixos foi mais rápida.

5.2.4 Avaliação dos resultados

A partir dos experimentos relatados nas seções 5.2.2 e 5.2.3 podemos fazer algumas observações comparando o algoritmo de Landau e Vishkin original e o algoritmo modificado.

- A fase de iteração do algoritmo original é mais rápida que a do algoritmo modificado.

- O pré-processamento do algoritmo modificado é mais rápido que o do original.
- Para alfabetos grandes, a diferença no tempo de pré-processamento é ainda maior, embora a economia de espaço seja reduzida.
- O algoritmo modificado usa menos espaço que o algoritmo original. Em especial, ao usarmos a versão com espaço linear, a diferença é máxima.

Em todos os casos o algoritmo modificado usou menos espaço que o algoritmo original, e pudemos apresentar dados reais que puderam ser processados com o algoritmo modificado que não puderam ser processados com o algoritmo original no ambiente computacional utilizado para os experimentos.

Em geral, se k for grande e toda a informação usada pelo algoritmo couber na memória principal do computador, a versão original do algoritmo terá desempenho melhor que a versão modificada. Em contrapartida, quando o uso de espaço passar do limite da memória principal do computador e começar a fazer uso de memória virtual, a vantagem de processamento do algoritmo original tende a diminuir, até efetivamente desaparecer quando houver palavras que não podem ser processadas pelo algoritmo original e que puderem ser processadas pelo algoritmo modificado.

6 Conclusão e caminhos futuros

Estudamos o algoritmo de Landau e Vishkin e identificamos uma oportunidade de melhorar o seu uso de espaço ao substituir as árvores de sufixos utilizadas no algoritmo original por arranjos de sufixos, que são estruturas de dados mais compactas. Para isso foi necessário um estudo que nos permitiu chegar a um cálculo em tempo constante do menor valor em um intervalo de um arranjo (também conhecido como *RMQ — range minimum query*), e verificar que esse cálculo seria suficiente para que a nossa proposta fosse viável.

Desenvolvemos nossa proposta e apresentamos neste trabalho um algoritmo para o problema de pesquisa aproximada de padrões em palavras baseado no algoritmo de Landau e Vishkin(4) de complexidades $\theta(kn)$ para execução e $\theta(kn)$ ou $\theta(n)$ para uso de espaço. O algoritmo de Landau e Vishkin usa uma consulta em tempo constante do LCA de pares de folhas de uma árvore de sufixos para calcular saltos em tempo constante ao longo das diagonais da tabela de programação dinâmica.

Em primeiro lugar verificamos que a informação essencial para esses saltos é o comprimento do maior prefixo comum de um sufixo do texto e um sufixo do padrão. Seguindo as idéias expostas por Abouelhoda, Kurtz e Ohlebusch(25), pudemos verificar que esse comprimento seria igual ao valor mínimo em um intervalo do arranjo *lcp* construído a partir do arranjo de sufixos. A chave para realizarmos esse cálculo em tempo constante foram as árvores cartesianas (26), após serem processadas para responder consultas de LCA em tempo constante.

O nosso objetivo foi diminuir o uso de espaço do algoritmo de Landau e Vishkin. Apesar de aumentarmos a quantidade de passos na fase de pré-processamento, e introduzirmos uma quantidade maior de estruturas de dados, verificamos que a versão modificada do algoritmo usava menos espaço que a original, mantendo a complexidade linear para a fase de pré-processamento e $\theta(kn)$ para a fase de iteração.

Em (5) apresentamos uma previsão teórica de economia de espaço da ordem de $4n$

bytes — que neste trabalho modificamos para $12n$ bytes — sobre uma implementação de árvores de sufixos que utilizasse $12n$ bytes. Na prática essa medida não é exata porque o uso exato de espaço de uma árvore de sufixos depende do alfabeto utilizado e da sua estrutura (freqüência de cada caractere, quantidade e tamanho de repetições). Nos experimentos realizados, o uso de espaço da árvore de sufixos ficou entre $10n$ e $19N$, de acordo com o alfabeto utilizado, e a economia média de espaço no pré-processamento variou de $8n$ a $35n$ (para caracteres ASCII que podem ser impressos e alfabeto binário, respectivamente).

Fizemos uma avaliação experimental com dados gerados aleatóriamente para vários tamanhos e alfabetos e para dados reais, disponíveis publicamente e retirados dos projetos NCBI Genbank, Projeto Gutenberg e *Corpus* de Canterbury.

A avaliação experimental foi satisfatória, pois comprovou o ganho de espaço esperado. O uso de alfabetos maiores diminui o espaço utilizado pela árvore de sufixos do algoritmo original e diminui a economia obtida, ao passo que alfabetos pequenos (como o alfabeto binário e DNA) aumentam o uso de espaço da árvore de sufixos e fazem com que a economia de espaço do algoritmo modificado seja mais expressiva.

Em especial para alfabetos de 4 símbolos (seqüências de DNA) o ganho foi em média 27% no cálculo com espaço linear para dados reais, e em média 20,9% para palavras aleatórias no algoritmo com espaço $\theta(kn)$ quando $k = 20$. A economia média para alfabetos de tamanho 4 foi de $24n$ bytes. Além disso o algoritmo original não foi capaz de processar uma seqüência baseada no cromossomo 22 do *H. sapiens* porque precisou de mais espaço que a memória virtual do computador utilizado foi capaz de prover, enquanto a mesma seqüência pôde ser processada pelo algoritmo modificado.

Quanto ao tempo de execução, o algoritmo modificado é mais lento que o original na fase de iteração e mais rápido na fase de pré-processamento. Isso faz com que o algoritmo modificado seja competitivo principalmente quando k é pequeno e $|\Sigma|$ é grande. Nos demais casos ainda assim a diferença de uso do processador foi de 22% e 28% para $k = 50$ e $k = 100$ no experimento com seqüências de DNA. Apesar disso a diferença do tempo real de execução diminuiu na medida em que o algoritmo original usava mais espaço e começou a demandar memória virtual do computador utilizado.

Do ponto de vista teórico, entendemos que o estudo desse algoritmo e a sua implementação são de grande utilidade para dominar as ferramentas complexas que ele utiliza, como o processamento de árvores para consultas de LCA em tempo constante. Além disso, ao implementar uma técnica nova para alguns dos componentes utilizados por esse

algoritmo foi possível testar essa técnica e comparar com a técnica original, e entender as vantagens e desvantagens que cada técnica apresenta ao ser utilizada num problema real.

Entendemos que a maior contribuição deste trabalho é justamente a implementação e avaliação da técnica para cálculo do LCE de dois sufixos de uma palavra usando arranjos de sufixos ao invés de árvores de sufixos, e uma ferramenta para avaliar empiricamente outras técnicas para o cálculo do LCE de dois sufixos de uma palavra que possam vir a serem desenvolvidos.

Recentemente foi publicado na conferência *Combinatorial Pattern Matching 2006* um artigo de Fischer e Heun(36) que descreve um algoritmo para calcular o LCE de dois sufixos de uma palavra em tempo constante, após pré-processamento linear que não necessita da construção de uma árvore cartesiana e do cálculo de LCA. O primeiro trabalho futuro que enxergamos seria atualizar a nossa versão do algoritmo de Landau e Vishkin para utilizar essa técnica e comparar o uso de espaço e o tempo de execução com a versão original e com a que desenvolvemos.

Outro caminho futuro que visualizamos seria tentar adaptar o algoritmo de Landau e Vishkin para a utilização em alinhamento de seqüências biológicas, transformando-o num algoritmo de maximização *exato* baseado em similaridade, que utilize matrizes de pontuação como é possível fazer com os algoritmos FASTA, BLAST e de programação dinâmica.

Se pudermos estabeler a aplicabilidade do algoritmo de Landau e Vishkin para dados biológicos, a sua utilização de forma paralela seria uma conseqüência imediata. O ano de 2006 foi marcado pela proliferação de processadores com múltiplos núcleos (*multi-core*) disponíveis já em preços acessíveis para computadores pessoais de mesas e portáteis. Ao quebrar o texto sendo pesquisado em palavras com alguma sobreposição é possível realizar o pré-processamento e a iteração de cada uma dessas partes em paralelo, e a comunicação entre cada uma dessas execuções seria mínima, o que resultaria em um *speed-up* expressivo. No caso distribuído, a memória adicional presente em cada nó do sistema seria melhor aproveitada pelo nosso algoritmo, permitindo aumentar o tamanho da massa de dados processada por cada componente do sistema. Uma aplicação possível para isso seria a pesquisa de uma seqüência em bases de dados de proteínas como o *Swissprot*.

Referências

1 NEEDLEMAN, S. B.; WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, v. 48, n. 3, p. 443–453, March 1970. ISSN 0022-2836.

2 SMITH, T. F.; WATERMAN, M. S. Identification of common molecular subsequences. *J Mol Biol*, v. 147, n. 1, p. 195–197, March 1981. ISSN 0022-2836.

3 HIRSCHBERG, D. A linear space algorithm for computing the maximal common subsequences. *Communications of the ACM*, v. 18, p. 341–343, 1975.

4 LANDAU, G.; VISHKIN, U. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In: *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM Press, 1986. p. 220–230. ISBN 0-89791-193-8.

5 MIRANDA, R. de C.; AYALA-RINCÓN, M. A Modification of the Landau-Vishkin Algorithm Computing Longest Common Extensions via Suffix Arrays (resumo estendido – publicada como artigo completo no XXXI CLEI). In: *Brazilian Symposium on Bioinformatics*. São Leopoldo, RS, Brasil: Springer Verlag, 2005. (Lecture Notes in Bioinformatics, v. 3594), p. 210–213.

6 KNUTH, D. E. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89683-4.

7 DURBIN, R. et al. *Biological Sequence Analysis*. Cambridge, UK: Cambridge University Press, 1998.

8 PEVZNER, P. A. *Computational Molecular Biology: An Algorithmic Approach*. Cambridge, MA, USA: The MIT Press, 2000. ISBN 0262161974.

9 GUSFIELD, D. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. New York, NY, USA: Cambridge University Press, 1997.

10 WEINER, P. Linear pattern matching algorithm. In: *14th Annual Symposium on Switching and Automata Theory*. [S.l.: s.n.], 1973. p. 1–11.

11 MCCREIGHT, E. M. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association for Computing Machinery*, v. 23, n. 2, p. 262–272, abr. 1976.

12 UKKONEN, E. On-line Construction of Suffix-Trees. *Algorithmica*, v. 14, p. 249–260, 1995.

13 KURTZ, S. Reducing the space requirement of suffix trees. *Softw., Pract. Exper.*, v. 29, n. 13, p. 1149–1171, 1999.

14 KURTZ, S.; GIEGERICH, R. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, p. 331–353, 1997.

15 MUMMER Home Page. Disponível em: <http://mummer.sourceforge.net/>. Acesso em: 11/2006.

16 KURTZ, S. et al. Reputer: The manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, v. 29, n. 22, p. 4633–4642, 2001.

17 SEDGEWICK, R. *Algorithms in Java, 3rd Edition, parts 1–4*. USA: Addison-Wesley, 2003.

18 MANBER, U.; MYERS, G. *Suffix arrays: A new method for on-line string searches*. [S.l.], 1989.

19 KO, P.; ALURU, S. Space-Efficient Linear Time Construction of Suffix Arrays. *Journal of Discrete Algorithms*, v. 3, n. 2-4, p. 143–156, 2005.

20 KäRKKäINEN, J.; SANDERS, P. Simpler linear work suffix array construction. In: *Int. Colloquium on Automata, Languages and Programming*. [S.l.]: Springer Verlag, 2003. (Lecture Notes in Computer Science, v. 2719), p. 943–955.

21 KIM, D. K. et al. Linear-time construction of suffix arrays. In: *14th Annual Symposium on Combinatorial Pattern Matching*. [S.l.]: Springer Verlag, 2003. (Lecture Notes in Computer Science, v. 2676), p. 186–199.

22 MANZINI, G.; FERRAGINA, P. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, v. 23, n. 40, p. 33–50, 2004.

23 FARACH, M. Optimal suffix tree construction with large alphabets. In: . [S.l.: s.n.], 1997. p. 137–143.

24 KASAI, T. et al. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: *12th Annual Symposium on Combinatorial Pattern Matching*. [S.l.]: Springer Verlag, 2001. (Lecture Notes in Computer Science, v. 2089), p. 181–192.

25 ABOUELHODA, M.; KURTZ, S.; OHLEBUSCH, E. The enhanced suffix array and its applications to genome analysis. In: *Workshop on Algorithms in Bioinformatics*. [S.l.]: Springer Verlag, 2002. (Lecture Notes in Computer Science, v. 2452).

26 GABOW, H. N.; BENTLEY, J. L.; TARJAN, R. E. Scaling and related techniques for geometry problems. In: 16th *ACM STOC*. [S.l.: s.n.], 1984. p. 135–143.

27 BENDER, M.; FARACH-COLTON, M. The LCA Problem Revisited. In: *LATIN 2000*. London, UK: Springer Verlag, 2000. (Lecture Notes in Computer Science, v. 1776), p. 88–94.

28 PROJECT Gutenberg. Disponível em: <http://www.gutenberg.org/>. Acesso em: 11/2006.

29 NCBI Genbank. Disponível em: <http://ncbi.nlm.nih.gov/Genbank/index.html>. Acesso em: 11/2006.

30 CANTERBURY Corpus. Disponível em: <http://corpus.canterbury.ac.nz/>. Acesso em: 11/2006.

31 KURTZ, K. et al. Versatile and open software for comparing large genomes. *Genome Biology*, v. 5, p. R12, 2004.

32 PUGLISI, S. J.; SMYTH, W. F.; TURPIN, A. The Performance of Linear Time Suffix Sorting Algorithms. In: *DCC '05: Proceedings of the Data Compression Conference*. Washington, DC, USA: IEEE Computer Society, 2005. p. 358–367.

33 BENTLEY, J.; SEDGEWICK, R. Fast Algorithms for Sorting and Searching Strings. In: *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997. p. 360–369.

34 FERRAGINA, P.; GROSSI, R. The string b-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, v. 46, n. 2, p. 236–280, 1999.

35 BENTLEY, J.; MCILROY, M. D. Engineering a sort function. *Software, Practice and Experience*, v. 23, n. 11, p. 1249–1265, 1993.

36 FISCHER, J.; HEUN, V. Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE. In: *Combinatorial Pattern Matching*. [S.l.]: Springer Verlag, 2006. (Lecture Notes in Computer Science, v. 4009), p. 36–48.