Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um índice baseado em árvores de sufixos comprimidas com baixo consumo de memória

Daniel Saad Nogueira Nunes

Brasília
2013

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um índice baseado em árvores de sufixos comprimidas com baixo consumo de memória

Daniel Saad Nogueira Nunes

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Orientador
Prof. Dr. Mauricio Ayala Rincón

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Programa de pós-graduação em Informática

Coordenador: Prof. Dr. Ricardo Pezzuol Jacobi

Banca examinadora composta por:

Prof. Dr. Mauricio Ayala Rincón (Orientador) — CIC/UnB
Prof. Dr.ª Maria Emília Machado Telles Walter — CIC/UnB
Prof. Dr. Guilherme Pimentel Telles — IC/Unicamp

CIP — Catalogação Internacional na Publicação

Nunes, Daniel Saad Nogueira.

Um índice baseado em árvores de sufixos comprimidas com baixo consumo de memória / Daniel Saad Nogueira Nunes. Brasília : UnB, 2013.
90 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2013.

1. árvores de sufixos comprimidas, 2. arranjos de sufixos comprimidos, 3. processamento de palavras.

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Um índice baseado em árvores de sufixos comprimidas com baixo consumo de memória

Daniel Saad Nogueira Nunes

Dissertação apresentada como requisito parcial
para conclusão do Mestrado em Informática

Prof. Dr. Mauricio Ayala Rincón (Orientador)
CIC/UnB

Prof. Dr.ª Maria Emília Machado Telles Walter
CIC/UnB

Prof. Dr. Guilherme Pimentel Telles
IC/Unicamp

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Programa de pós-graduação em Informática

Brasília, 17 de maio de 2013

Agradecimentos

Agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo apoio a pesquisa realizada durante o período de mestrado. Agradeço também ao Decanato de Pesquisa e Pós-graduação (DPP) da Universidade de Brasília pelo suporte à pesquisa e participação de eventos.

Resumo

Árvores de sufixos e arranjos de sufixos são índices bem conhecidos na literatura que organizam a informação combinatória de palavras e que possuem uma vasta gama de aplicações nas mais diversas áreas como processamento de palavras e análise de sequências biológicas. A principal desvantagem destes índices é a demanda excessiva de espaço na prática para entradas grandes. Recentemente, vários trabalhos vem explorando uma estrutura denominada árvore de sufixos comprimida, que oferece a mesma funcionalidade de uma árvore sufixos convencional e é baseada em arranjos de sufixos comprimidos, informação de maior prefixo comum e operações de navegação na árvore.

Neste trabalho uma implementação de uma árvore de sufixos comprimida baseada em consultas de *range-minimum-query* e *next/previous smaller queries* é apresentada. A implementação possui uma baixa memória de pico, requerendo pouco mais de espaço durante a sua construção em relação ao espaço de representação final da estrutura.

Experimentos mostram que este índice é útil para diversas aplicações visto que é possível efetuar operações complexas como travessia de *links* de sufixos e consultas de ancestral comum mais baixo até quando a quantidade disponível de memória é baixa, já que a estrutura comprimida cabe em memória principal mesmo em computadores mais modestos.

Palavras-chave: árvores de sufixos comprimidas, arranjos de sufixos comprimidos, processamento de palavras.

Abstract

Suffix trees and suffix arrays are well known indices which organize the combinatorial information of strings and which have a large amount of applications in areas such as string processing and molecular sequence analysis. The main drawback of these indices is that they demand a lot amount of space for large inputs. Recently, several works have been exploring a data structure called compressed suffix tree, which offers the same functionality of the suffix tree and is based on compressed suffix array, compressed longest common prefix information and navigational operations.

In this work, the implementation of a compressed suffix tree based on range-minimum-queries and next/previous smaller values queries is presented. The implementation has a low peak memory usage, requiring roughly more than the space needed to represent the index during the construction.

Experiments show that this index is useful for many applications since one can execute complex operations like suffix link traversals and longest common ancestor queries being of great interest when the amount of available memory is low, because the structure fits in main memory of ordinary computers.

Keywords: compressed suffix trees, compressed suffix arrays, String Processing.

Sumário

Capítulo 1

Introdução

Textos digitais, que podem ser vistos como sequências de símbolos sobre um mesmo alfabeto, são considerados algumas das formas mais elementares de representação de informação. Eles podem representar código fonte, sequências proteicas e seus metadados, arquivos de música, entre outros mais diversos tipos de informação, conforme ilustrado pelas Figuras 1.1, 1.2 e 1.3.

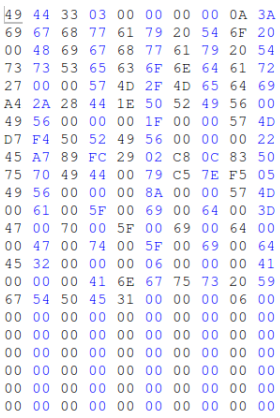


Figura 1.1: Arquivo mp3.

```c
int d2l(frame_t* frame){
    u4 vh,vl;
    s8 l;
    u4 resl,resh;
    u8 d = ZEROD;
    double db;
    vh = popOperand(frame);
    vl = popOperand(frame);
    d |= vh;
    d <<= 32;
    d |= vl;
    memmove(&db,&d,sizeof(u8));
    if(db > INT64_MAX){
        l = INT64_MAX;
    }
    else if(db < INT64_MIN){
        l = INT64_MIN;
    }
    else{
        l = (s8)db;
    }
    resl = l & L32;
    l >>= 32;
    resh = (l & L32);
    pushOperand(frame,resl);
    pushOperand(frame,resh);
    return(0);
}
```

Figura 1.2: Programa em C.

```
DR   HAMAP; MF_01330; Ycf2; 1; -.
DR   InterPro; IPR003593; ATPase_AAA+_core.
DR   InterPro; IPR003959; ATPase_AAA_core.
DR   InterPro; IPR008543; Uncharacterised_Ycf2.
DR   Pfam; PF00004; AAA; 1.
DR   Pfam; PF05695; DUF825; 2.
DR   SMART; SM00382; AAA; 1.
PE   3: Inferred from homology;
KW   ATP-binding; Chloroplast; Nucleotide-binding; Plastid.
FT   CHAIN         1   2280       Protein ycf2.
FT                                /FTId=PRO_0000242530.
FT   NP_BIND    1634   1641       ATP (Potential).
SQ   SEQUENCE   2280 AA;  268260 MW;  41048E53F6FAB295 CRC64;
     MKGHQFKSWI FELREILREI KNSHHFLDSW TQFNSVGSFI HIFFHQERFI
     LLSRNSQGST SNRYFTIKGV VLFVVAVLLY RINNRNMVER KNLYLTGLLP
     DTLEESFGSS NINRLIVSLL YLPKGKKISE SCFLDPKEST WVLPITKKCI
     WRNWIGKKRD SSCKISNETV AGIEISFKEK DIKYLEFLFV YYMDDPICKD
     PSKRRNIINL NSRQLFEILV KDWICYLMFA FREKIPIEVE GFFKQQGAGS
     SHLFSRKKWA ISLQNCAQFH MWQFRQDLFV SWGKNPPESD FLRNISRENW
     KDRFFSKVRN VSSNIQYDST RSSFVQVTDS SQLKGSSDQS RDHFDSISNE
     REIQQLKERS ILWDPSFLQT ERTELESDRF SKCLSGYSRL FTEREKEMKN
     LGNPTRSILS FFSDRWSELH LGSNPTERST RDQKLLKKEQ DVSFVPSRRS
     IITYLQNTVS IHPISSDPGC DMVPKDELDM HSSHKISFLN KNTFFDLFHL
```

Figura 1.3: Sequência proteica.

Pela sua versatilidade de representação de informação, a quantidade de textos vem crescendo em uma taxa exponencial, de acordo com Navarro e Mäkinen [NM07]. Dada a grande im-

portância dos textos, métodos de Processamento de Palavras são necessários para manipular, inferir e trabalhar com esse modelo de representação de informação.

1.1 Processamento de palavras

Processamento de palavras é uma área que possui uma vasta gama de aplicações em diversas outras áreas, como Biologia Computacional, Processamento de Sinais Digitais e Recuperação de Informação.

1.1.1 Recuperação de Informação

De acordo com Manning *et al.*, a área de Recuperação de Informação consiste em encontrar material (tipicamente documentos) de natureza não estruturada (geralmente textos) que satisfaz uma necessidade de informação dentre uma larga coleção de materiais (geralmente armazenada em computadores).

Atualmente, as coleções de materiais são extremamente extensas, como por exemplo a *Web*, e portanto mecanismos eficientes para Recuperação de Informação são necessários para que os documentos procurados possam ser encontrados de maneira eficiente.

O casamento de padrões é um desses mecanismos e consiste em verificar se um determinado padrão ocorre no texto em questão e, caso exista, reportar as suas ocorrências. Alguns dos tipos de casamento de padrões são o exato, o inexato e o de expressões regulares.

Casamento exato de padrões

O casamento exato de padrões tem como objetivo localizar ocorrências exatas de um determinado padrão em um texto, isto é, uma ocorrência do padrão no texto é válida se e somente se todos os símbolos do padrão casarem com a os símbolos do texto sem permitir erros.

Técnicas de casamento exato de padrões são bem conhecidas na literatura, conforme visto em [KMJP77], [BM77] e [CR03].

A figura 1.4 nos mostra a ferramenta grep para casamento exato do padrão $P = religion$.

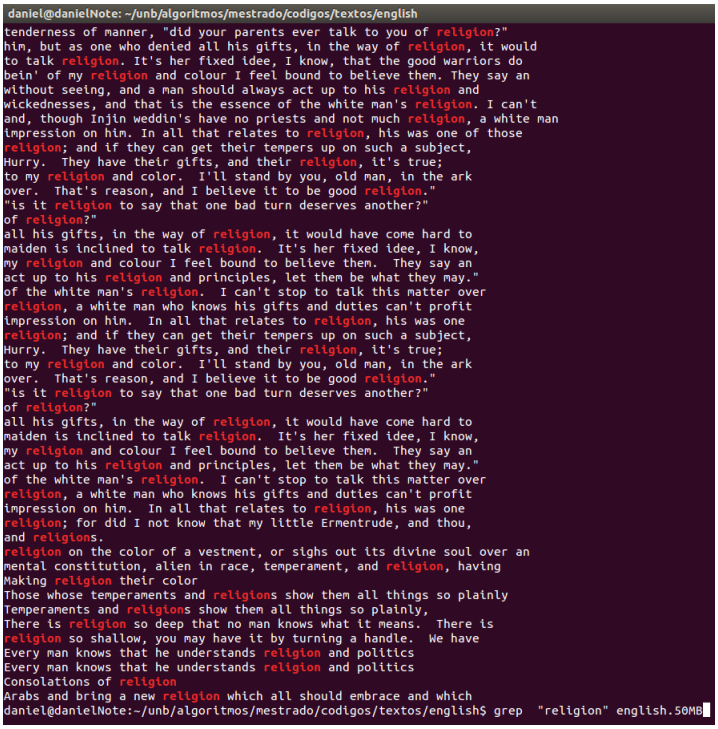


Figura 1.4: Ferramenta Grep efetuando casamento exato do padrão $religion$.

Casamento aproximado de padrões

Muitas vezes o casamento exato de padrões não é suficiente para que os materiais que contêm a informação requisitada sejam recuperados, visto que cada vez mais as coleções têm se tornado mais heterogêneas e suscetíveis a inserção de erros.

O casamento aproximado (ou inexato) assemelha-se ao o casamento exato de padrões, no entanto, o primeiro permite erros, ou seja, operações como inserção e deleção são permitidas para que o padrão case com o texto. Frequentemente não-casamentos de símbolos são permitidos também.

Um exemplo de casamento aproximado entre o texto $T = acagt$ e o padrão $P = agc$ é ilustrado pela Figura 1.5. O símbolo "-" representa a operação de inserção, símbolos em vermelho correspondem à não-casamentos e símbolos azuis correspondem ao casamento sem erros.

a c a g t
a - - g c

Figura 1.5: Casamento aproximado do padrão $P = agc$ contra o texto $T = acagt$.

Casamento de expressões regulares

As expressões regulares são um formalismo capaz de expressar todas as palavras de uma determinada linguagem formal (um conjunto de palavras) de maneira simples e sucinta, fatores muito desejáveis pela Recuperação de Informação.

Alguns exemplos de expressão regular e seus conjuntos associados seguem abaixo:

- aa^*ca: o conjunto de todas as palavras que começam com o símbolo "a", possuem zero ou mais símbolos "a" e terminam com "ca".
- $(aa|ac)ta^*g$: o conjunto de todas as palavras que começam com "aa" ou com "ac" e são seguidas de "t", zero ou mais símbolos "a" e terminam com "g".
- a^+b^*: o conjunto de todas as palavras que começam com uma ou mais ocorrências de "a" e são seguidas por zero ou mais ocorrências de "b".

O trabalho de Thompson propõe um algoritmo para casamento de expressões regulares [Tho68].

1.1.2 Biologia Computacional

A estrutura primária de uma molécula de ácido desoxirribonucleico (ADN) pode ser encarada como um texto sobre o alfabeto quaternário $\{a, c, g, t\}$. Proteínas podem ser representadas analogamente com um alfabeto de 20 símbolos.

A partir destas premissas é notável que o processamento de palavras tenha um papel essencial para gerar resultados biologicamente significantes, conforme ilustrado em [Gus97] e [J. 97].

A presença de métodos computacionais é extremamente relevante para tratar problemas biológicos dada a gigantesca produção de dados das tecnologias de sequenciamento de alto desempenho. Técnicas que tratam erros são essenciais devido aos problemas inerentes às tecnologias de sequenciamento.

Mapeamento de fragmentos

Os dados produzidos pelos sequenciadores de alto desempenho representam fragmentos de ADN (*reads*) que muitas vezes necessitam ser mapeadas em um genoma de referência para uma análise genômica (ou transcritômica) mais acurada. O mapeamento é ilustrado pela Figura 1.6.

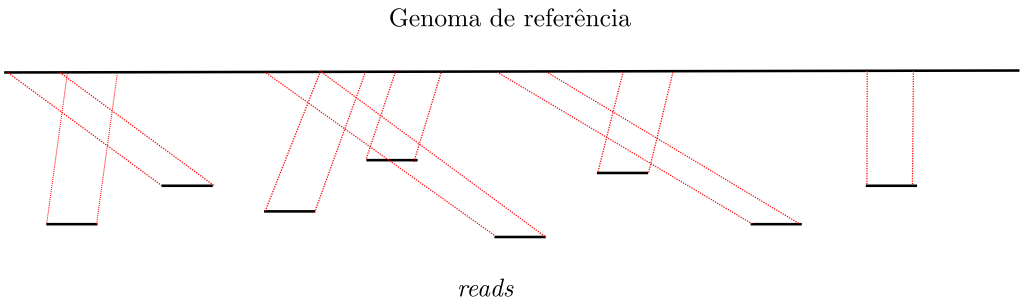


Figura 1.6: Mapeamento de fragmentos em um genoma de referência.

Diversas implementações de mapeadores de fragmentos estão disponíveis, como o Bowtie [LTPS09], BWA [LD09] e o Segemehl [HOK+09].

Montagem de fragmentos

Quando um genoma ainda não conhecido deve ser obtido, são utilizados procedimentos de montagem. Tais procedimentos consistem em detectar sobreposições entre os fragmentos pro-

duzidos pelo sequenciador para concatenar esses fragmentos de modo a construir o genoma de interesse, como ilustrado pela Figura 1.7.

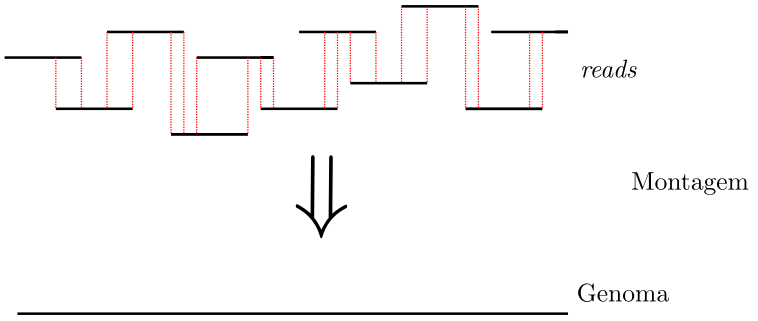


Figura 1.7: Montagem de fragmentos.

Alinhamento de sequências biológicas

Para determinar a similaridade de sequências recorre-se à solução do problema de alinhamento, que procura alinhar as sequências biológicas considerando casamentos, inserções, deleções e não-casamentos de modo a alcançar uma pontuação máxima, que quantifica a similaridade entre as sequências. Frequentemente casamentos estão associados com uma pontuação positiva e erros com uma pontuação negativa.

Um alinhamento de sequências com uma alta pontuação pode indicar que os genes associados às sequências tem uma função parecida. Consequentemente, alinhamentos múltiplos podem ser úteis para construção de uma árvore filogenética, que expressa a distância evolutiva entre diferentes organismos. Um exemplo de alinhamento múltiplo de sequências proteicas é ilustrado pela Figura 1.8.

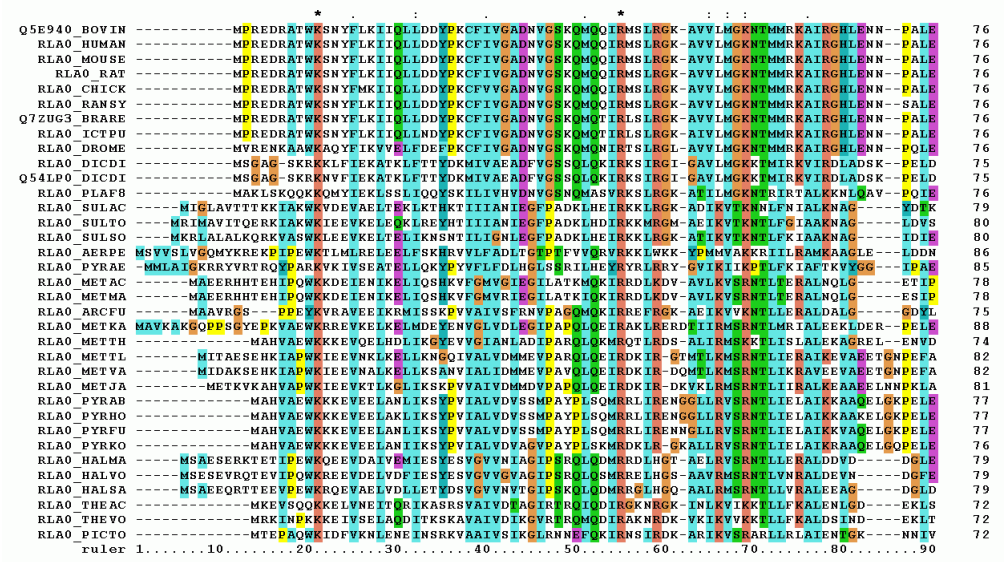


Figura 1.8: Alinhamento de proteínas pelo software ClustalW [TGH+02].

1.1.3 Processamento de Sinais Digitais

O Processamento de Palavras é de grande ajuda na área de Processamento de Sinais Digitais, de acordo com Navarro e Mäkinen [NM07]. Como uma gigantesca quantidade de arquivos multimídia tais como imagens, vídeos e sons encontram-se disponíveis com facilidade atualmente, são necessários mecanismos de busca sobre tais arquivos, que podem ser vistos como um texto sobre o alfabeto binário. Geralmente, estes arquivos passam por codificações com perda, necessitando então de mecanismos capazes de atuar sobre erros inerentes ao processo de compressão. Um destes mecanismos é o casamento aproximado de padrões.

Para compactar o sinal original impedindo que qualquer tipo de informação seja perdida, o Processamento de Palavras pode ser utilizado. Ao explorar propriedades do sinal como redundância e frequências de certos padrões, o sinal original pode ser representado em uma quantidade menor de espaço sem a perda de qualquer parte da informação. Tal processo é chamado de compressão sem perdas. Exemplos de mecanismos relacionados à compressão sem perda são a transformada Burrows-Wheeler a transformada Move-to-front e o Run-Length-Encoding, descritas brevemente em seguida.

Transformada Burrows-Wheeler

A transformada Burrows-Wheeler reorganiza o texto original de modo que símbolos iguais tendam a ficar em posições contíguas. A transformada é reversível, isto é, é possível recuperar o texto original.

A transformada consiste em ordenar uma matriz conceitual, contendo os sufixos do texto, e tomar como resultado a última coluna dessa matriz. A Figura 1.9 mostra a transformada Burrows-Wheeler sobre o texto $T = alabar\_a\_la\_alabarda\$$. A última coluna da matriz conceitual representa o resultado da transformada, que é dada por $BWT(T) = araadl\_ll\$\_bbaar\_aaaa$.

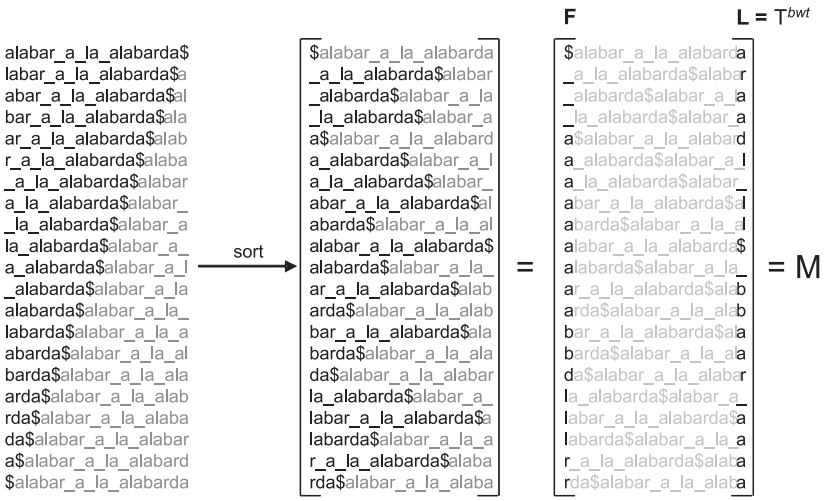


Figura 1.9: Transformada Burrows-Wheeler [NM07].

É importante ressaltar que a transformada Burrows-Wheeler apenas organiza a informação, não a comprime. No entanto, como os símbolos tendem a estar agrupados, é comum usar a

transformada Move-to-front, seguida do Run-Lenght encoding, para comprimir a informação,como segue.

Transformada Move-to-front

A transformada Move-to-front consiste em trocar um símbolo pela sua posição numa pilha de símbolos recentemente usados. Desta forma, cada vez que um símbolo é lido da entrada, a transformada fornece como saída a posição desse símbolo na pilha e coloca o este mesmo símbolo no topo da pilha (o topo da pilha representa a posição 0). Inicialmente a pilha contém os símbolos em ordem lexicográfica.

Para ilustrar a transformada tome $\Sigma = \{a, b \ldots, z\}$ e $W = mississippi$. A aplicação da transformada nos dá a saída $(12, 9, 18, 0, 1, 1, 0, 1, 16, 0, 1)$, conforme ilustrado pela Tabela 1.1.

Tabela 1.1: Transformada *move-to-front* passo a passo.

Símbolo a ser codificado	Pilha	Saída
m	$abcdefghijklmnopqrstuvwxyz$	12
i	$mabcdefghijklnopqrstuvwxyz$	9
s	$imabcdefghjklnopqrstuvwxyz$	18
s	$simabcdefghjklnopqrtuvwxyz$	0
i	$simabcdefghjklnopqrtuvwxyz$	1
s	$ismabcdefghjklnopqrtuvwxyz$	1
s	$simabcdefghjklnopqrtuvwxyz$	0
i	$simabcdefghjklnopqrtuvwxyz$	1
p	$ismabcdefghjklnopqrtuvwxyz$	16
p	$pismabcdefghjklnoqrtuvwxyz$	0
i	$pismabcdefghjklnoqrtuvwxyz$	1

O método é reversível e o processo de decodificação é análogo ao de codificação, a pilha contém os símbolos em ordem lexicográfica, e para a entrada i, recuperamos a saída que corresponde ao i-ésimo símbolo na pilha.

Run-length encoding

O run-length encoding consiste em codificar agrupamentos do mesmo símbolo de maneira mais compacta. Tomando como exemplos o texto $T = \underbrace{aaaaa}_{5}\underbrace{bbbb}_{4}abab$, poderíamos representar como $T^{rle} = \mathbf{5}a\mathbf{4}babab$.

Existem diversas variações dessa técnica, mas a essência continua sendo comprimir a informação redundante a partir dos agrupamentos de símbolos iguais.

Tomando como exemplo o trabalho de Ferragina e Manzini [FM05], a estrutura descrita pelos autores usava a transformada Burrows-Wheeler, para agrupar símbolos iguais, seguida da aplicação da transformada move-to-front para transformar esses agrupamentos de símbolos iguais em blocos de zeros para que o run-length encoding fosse aplicado a fim de compactar esses blocos.

1.2 Estruturas de Dados

Diversas estruturas de dados, também denominadas de índices, são úteis em Processamento de Palavras. Tais estruturas de dados procuram organizar a informação de tal forma que consultas sobre o texto original sejam feitas de maneira eficiente. Duas estruturas importantes são as árvores de sufixos e os arranjos de sufixos.

A árvore de sufixos é uma estrutura em forma de árvore de modo que representa a informação de todos os sufixos de maneira compacta, utilizando apenas $O(n)$ palavras de memória. Em comparação, uma *trie* leva no pior caso gasta $O(n^2)$ para a representação dos sufixos.

Esta estrutura é extremamente flexível, podendo ser utilizada em uma diversa gama de aplicações, como visto em [Gus97].

Apesar de ser versátil, a árvore de sufixos consome muito espaço na prática. Tomando como exemplo o genoma humano, que possui $\approx 3 \cdot 10^9$ símbolos, a árvore de sufixos possui um fator de $15\times$ de espaço em relação ao tamanho do genoma segundo a melhor implementação conhecida [Kur99]. Isto significa que, a estrutura requer $45GB$ de memória para a indexação do genoma humano.

Como alternativa mais econômica das árvores de sufixos, os arranjos de sufixos foram propostos por Udi Manber e Gene Myers [MM90].

Os arranjos de sufixos representam arranjos de inteiros contendo a posição de cada sufixo do texto em ordem lexicográfica.

Em relação à arvore de sufixos, o arranjo de sufixos mostra-se uma alternativa mais econômica. Na prática, o fator envolvido de espaço é de $4\times$ sobre o tamanho da entrada, muito menor que o fator demandado pelas árvores de sufixos. No entanto, para criar a estrutura para o genoma humano, seria necessário pelo menos $12GB$ de memória para armazenamento do arranjo de sufixos, o que representa bastante espaço para máquinas mais modestas.

Conclui-se então que, mesmo com estruturas bem conhecidas, o consumo de espaço é excessivo na prática, inviabilizando o seu uso quando há pouca memória disponível.

Estas estruturas de dados serão abordadas detalhadamente no Capítulo 2.

1.3 Motivação

Como visto anteriormente, existem índices eficientes para processamento de palavras, como as árvores de sufixos e os arranjos de sufixos.

As árvores de sufixos, embora estruturas extremamente flexíveis que podem ser usadas em uma vasta gama de aplicações, possuem uma deficiência quanto ao uso de espaço. Em fatores práticos, a constante envolvida chega a um fator de $15\times$ sobre o tamanho da entrada. Logo, seu uso para entradas grandes, é inviável.

Arranjos de sufixos surgiram como uma alternativa espaço-econômica para árvores de sufixos. Em termos práticos, a constante envolvida é de apenas $4\times$ sobre o tamanho da entrada, uma redução substancial em relação às árvores de sufixos, mas que ainda demanda uma quantia razoavelmente grande de memória para entradas de dimensões gigantescas, como a representação da estrutura primária do genoma humano.

Estruturas bem conhecidas como as descritas acima não são viáveis para entradas grandes. Consequentemente, o uso de espaço precisa ser amenizado com a adoção de estruturas de dados comprimidas, que se baseiam em propriedades específicas de estruturas de dados comuns para alcançar a compressibilidade desejada.

Exemplos de estruturas de dados que utilizam a compressão são os arranjos de sufixos comprimidos, o índice-FM [FM05], uma variação da primeira estrutura, e as árvores de sufixos comprimidas, foco deste trabalho.

1.4 Objetivos

Os objetivos deste trabalho são:

- Implementar um índice espaço-eficiente baseado em árvores de sufixos comprimidas.
- Implementar consultas não triviais sobre a árvore de sufixos comprimida, como consultas de ancestral comum mais baixo e travessia de *links* de sufixo.
- Codificar o índice elaborado em uma linguagem de programação que permita um alto desempenho.
- Validar o índice e compará-lo com outros índices existentes na literatura levando em consideração recursos como espaço e tempo.

1.5 Metodologia

Para alcançar os objetivos, realizou-se uma extensa revisão bibliográfica sobre os índices comprimidos encontrados na literatura para que uma implementação satisfatória fosse realizada.

A implementação, por sua vez, foi baseada no paradigma de orientação à objetos com a linguagem C++. A linguagem foi escolhida por questões de desempenho e flexibilidade.

A validação do índice foi observada ao comparar os resultados obtidos com implementações cujas correções já haviam sido demonstradas. Uma implementação é a `libdivsufsort` [Mor07], que baseia-se em arranjos de sufixos não-comprimidos. A outra implementação utilizada, baseia-se na árvore de sufixos comprimida do grupo SuDS da universidade de Helsinque [V. 13].

1.6 Contribuição do trabalho

A principal contribuição do trabalho é uma implementação de árvore de sufixos comprimida que é espaço-eficiente e ao mesmo tempo possui uma memória de pico relativamente baixa, isto é, durante a construção da estrutura, é preciso um pouco mais de espaço do que o necessário para representação final da própria estrutura.

A implementação mostrou-se de grande valia prática quando o ambiente dispõe de pouca memória, além de ser competitiva em termos de espaço em relação a outro índice comprimido presente na literatura e à uma implementação que não apresentava compressão.

1.7 Organização do documento

Durante a escrita da dissertação os capítulos foram dispostos em uma determinada ordem de modo que refletisse o desenvolvimento do índice e do trabalho.

Referencial Teórico: Traz conceitos importantes sobre as estruturas básicas: árvores, arranjos de sufixos e arranjos de sufixos enriquecidos.

Índices Comprimidos: Aborda conceitos importantes de compressão e estruturas de dados comprimidas em geral, como arranjos de sufixos comprimidos, índice-FM e árvores de sufixos comprimidas.

Soluções propostas: Descreve os algoritmos, métodos desenvolvidos neste trabalho.

Experimentos e comparações: São avaliados o tempo de construção , o custo de operação básica e o espaço consumido pela estrutura em relação à outras implementações.

Considerações finais: Discute brevemente os pontos principais do trabalho e indica linhas de pesquisas futuras.

Capítulo 2

Referencial teórico

Conforme visto anteriormente, a área de Processamento de Palavras possui uma vasta gama de aplicações com os mais variados tipos de problemas. Estruturas de dados eficientes são necessárias para que os métodos de processamento de palavras sejam viáveis na prática. Tais estruturas de dados são comumente chamadas de **índices**. Elas reúnem informação sobre o texto para que consultas sobre este sejam feitas de maneira eficiente quanto em tempo e espaço.

Antes de introduzir conceitos sobre os índices, são necessárias algumas noções importantes para entendimento deste trabalho.

2.1 Definições e notações

Definição 2.1 (Alfabeto)
Um alfabeto Σ é um conjunto finito de símbolos $\Sigma = \{\alpha_0, \alpha_1, \ldots, \alpha_{\sigma-1}\}$ com $\sigma = |\Sigma|$. Este conjunto possui uma ordem total $\alpha_0 < \alpha_1 < \ldots < \alpha_{\sigma-1}$ chamada de ordem lexicográfica.

Definição 2.2 (Palavra)
Uma palavra W sobre o alfabeto Σ é constituída somente por símbolos de Σ. O conjunto de todas as palavras de comprimento finito sobre o alfabeto Σ é denotado por Σ^*. Em especial $\epsilon \in \Sigma^*$ é a palavra vazia.

Definição 2.3 (Comprimento de palavra)
O comprimento de uma palavra W é denotado por $|W|$ e corresponde à quantidade de símbolos que compõem W. Em especial $|\epsilon| = 0$.

Notação 2.1 (Símbolo individual)
O i-ésimo símbolo de uma palavra W, $|W| = n > 0$ é representado por $W[i]$, com $0 \leq i < n$.

Notação 2.2 (Frequência)
$|W|_c$ representa a frequência do símbolo c na palavra W associada.

Definição 2.4 (Subpalavra)
Uma subpalavra de W, representada por $W[i, j]$ compreende todos os símbolos $W[i], W[i + 1], \ldots, W[j]$, com $0 \leq i \leq j < |W|$. Caso contrário $W[i, j] = \epsilon$.

Notação 2.3 (Sufixo)
Seja W uma palavra com $|W| = n$. O sufixo $W[i, n - 1]$ é denotado por W_i.

Notação 2.4 (Concatenação)
A concatenação de duas palavras R e S é denotada por RS.

Notação 2.5 (Texto e padrão)
Ao longo do documento, duas palavras em especial serão referenciadas com frequência, T e P, onde T representa um determinado texto e P um determinado padrão. Por convenção, $|T| = n$ e $|P| = m$. Além disso, o último símbolo de T é \$, um símbolo que marca o final do texto e ocorre somente na posição $T[n-1]$. Além disso, \$ é lexicograficamente menor que todos os outros símbolos.

2.2 Árvores de sufixos

A árvore de sufixos é um índice bastante conhecido na literatura. É uma estrutura de dados que representa todos os sufixos de uma palavra em espaço $O(n \log n)$ bits, ou $O(n)$ palavras de computador e podem ser construídas em tempo $O(n)$.

Inicialmente, as árvores de sufixos foram propostas por Weiner em 1973 [Wei73]. Sua construção levava tempo $O(n)$ se $\sigma = O(1)$. Uma vez que o trabalho de 1973 não se mostrava tão claro, o trabalho de McCreight em 1976 forneceu um outro algoritmo para a construção desta estrutura de modo a explicitar as propriedades combinatórias da estrutura de um modo menos complexo [McC76]. Na década de 90, o trabalho de Ukkonen forneceu o primeiro algoritmo *online* para construção de árvores de sufixos [Ukk95]. Este algoritmo é altamente intuitivo e de mais fácil compreensão que os trabalhos anteriores. O trabalho de Farach em 1997 introduz o primeiro algoritmo de construção de árvores de sufixos que tem um tempo de construção de $O(n)$ sem restrição sobre um alfabeto inteiro, isto é, não era necessário que $\sigma \in O(1)$ [Far97].

Definição 2.5 (Árvore de sufixos)
Uma árvore de sufixos sobre T é uma árvore orientada com raiz com exatamente $|T| = n$ folhas. Cada nó interno, excetuando a raiz, tem pelo menos dois filhos e cada aresta é rotulada com uma subpalavra não vazia de T. Cada aresta que parte de um nó interno começa com um símbolo diferente. A concatenação dos símbolos das arestas da raiz até a folha $0 \leq i < n$ soletra o sufixo $T[i, n-1]$.

A Definição 2.5 usa como base a nomenclatura adotada em [Gus97].

Nota 2.1 (Nós internos e prefixos)
Segue imediatamente da Definição 2.5 que existe um nó interno que é um ancestral comum de k folhas da árvore, se, e somente se, essas k folhas compartilham um prefixo comum e maximal que corresponde à subpalavra formada pela concatenação dos símbolos das arestas que saem da raiz e vão até o nó interno. Se o prefixo comum não fosse maximal, seria possível estendê-lo ao caminhar pelas arestas, mas como cada nó interno tem pelo menos 2 arestas que começam com símbolo diferente isto é impossível.

A Figura 2.1 ilustra uma árvore de sufixos para o texto $T = acaaacatat\$$. Para alcançar a cota de $O(n \log n)$ bits de espaço, as subpalavras que rotulam as arestas são representadas por inteiros (i, j) correspondendo à posição de início da subpalavra e à posição de fim, respectivamente. Na mesma Figura, são colocados símbolos e sufixos em vez de inteiros por motivo didático.

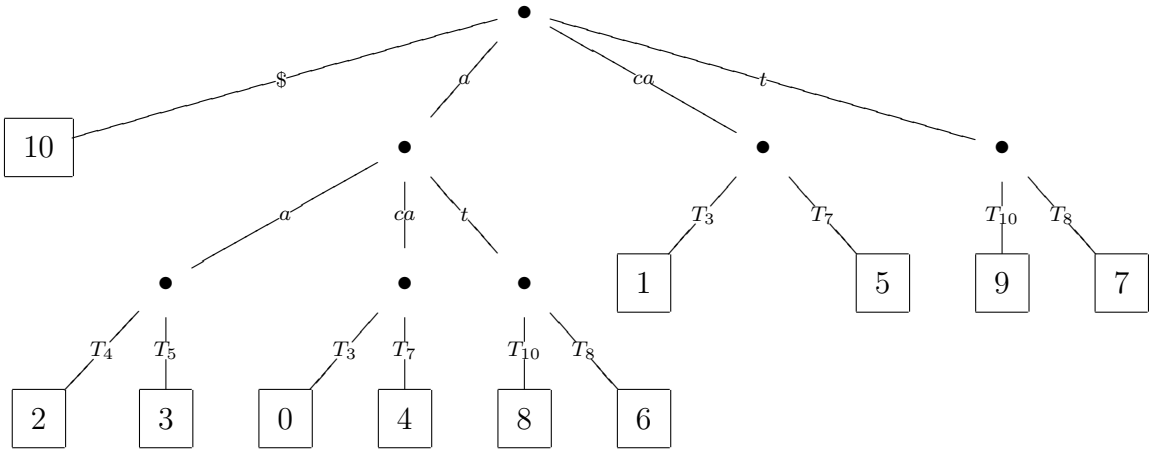


Figura 2.1: Árvore de sufixos para $T = acaaacatat\$$.

O problema de casamento exato de padrões pode ser resolvido facilmente ao utilizar uma árvore de sufixos. Basta construir a árvore para T e buscar P na árvore ao percorrer o caminho que soletra P a partir da raiz. Caso um caminho da árvore tenha casado com o padrão, então todas as folhas abaixo desse caminho correspondem a posições as quais o padrão ocorre no texto. Por exemplo, tomando a figura 2.1 e o padrão $P = aca$, padrão casaria com os rótulos da aresta até chegar no nó interno que da origem às folhas rotuladas com 0 e 4. Portanto, o padrão P ocorre nas posições 0 e 4 do texto.

Duas operações em árvores de sufixos são bastante comuns. A operação de atravessar um *link* de sufixo (*suffix link*), e a operação de ancestral comum mais baixo (lowest common ancestor ou LCA).

Definição 2.6 (*Link de sufixos*)
Seja um nó u de uma árvore de sufixos cuja concatenação do rótulo das arestas da raiz até este nó soletre a palavra C. O link *de sufixo pode ser visto como uma aresta que liga o nó u a um nó v da seguinte maneira:*

- *Se $C = \epsilon$, o nó u é a própria raiz que possui um* link *de sufixo para si mesma.*
- *Caso u for uma folha rotulada com a posição $i < n - 1$, então v é a folha rotulada com a posição $i + 1$. Se u for uma folha rotulada como $i = n - 1$, então u não possui* link *de sufixo.*
- *Se $C = aB$, então o nó u é um nó interno que não é a raiz e se liga ao nó v de maneira que a concatenação dos rótulos das arestas da raiz a v soletra a palavra B.*

O teorema exposto em [McC76] garante a existência do nó v no terceiro item da Definição 2.6. A árvore com os *links* de sufixos para o texto $T = acaaacatat\$$ é representada pela Figura 2.2.

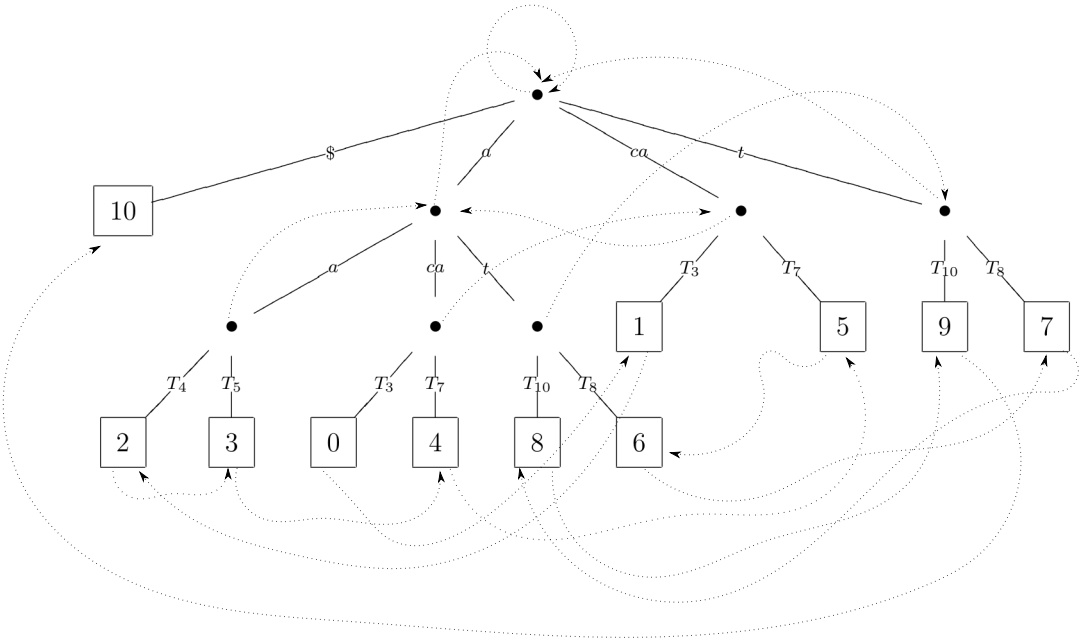


Figura 2.2: Árvore de sufixos para o texto $T = acaaacatat\$$ e seus *links* de sufixos.

Definição 2.7 (Ancestral comum mais baixo (LCA))
O ancestral comum mais baixo entre dois nós u e v em uma árvore de sufixos é único e corresponde ao ancestral de u e v mais distante da raiz.

Por exemplo, segundo a Definição 2.7 e tomando como base a Figura 2.1, o ancestral comum mais baixo das folhas rotuladas com 2 e 4 é o nó interno que é alcançado a partir da raiz pela aresta que soletra o símbolo a.

2.2.1 Aplicações

De acordo com Apostolico [Apo85] e Gusfield [Gus97], árvores de sufixos possuem uma vasta gama de aplicações, dentre algumas delas estão:

- Casamento exato de padrões.
- Maior subpalavra comum de duas palavras.
- Localização de repetições maximais.
- Localização de palíndromos maximais.
- Casamento aproximado de padrões.
- Codificação Lempel-Ziv.
- Computação das estatísticas de casamento.
- Construção de arranjos de sufixos.

2.2.2 Pontos negativos

Apesar da alta versatilidade das árvores de sufixos, estas possuem alguns pontos negativos.

As árvores de sufixos possuem uma pobre localidade de referência, isto é, dados referenciados na árvore (como nós, arestas, folhas, ...) podem estar em páginas distintas de memória e portanto, muitas vezes as páginas necessárias não se encontram em memória *cache*, o que atrapalha no desempenho, visto que estas páginas tem que ser carregadas para a memória *cache* devido ao *cache miss*, como visto em [AKO02].

Outro ponto fraco é que o tempo de construção e de consulta é pior, em termos práticos, se comparado a outras estruturas de dados como os arranjos de sufixos. Principalmente devido ao mal uso de *cache*, conforme visto em [MM90] e [AKO02].

A rigidez da estrutura de dados quanto à atualização também se mostra um ponto negativo. Para atualizar uma árvore de sufixos ao inserir um novo símbolo no texto, leva-se tempo $O(n)$ no pior caso, assintoticamente o mesmo que o tempo de construção da estrutura [ARC03].

Além disso, o principal ponto negativo é que o espaço gasto para representação da estrutura é demasiado para entradas grandes. Por exemplo, segundo a implementação melhor espaço-eficiente conhecida, a construção da estrutura para o genoma humano, que contém $\approx 3 \cdot 10^9$ símbolos, consumiria $45GB$ de memória principal, ou seja, existe um fator de $15\times$ associado à estrutura, de acordo com Kurtz [Kur99]. Consequentemente, o uso desse índice para entradas grandes é inviável. Estruturas mais espaço-econômicas, como os arranjos de sufixos, são mais apropriadas para a indexação de textos maiores. Para possíveis consultas mais complexas, como a consulta de *links* de sufixo e de ancestral comum mais baixo, necessita-se de espaço extra para que estas sejam realizadas em tempo eficiente [Gus97].

2.3 Arranjos de sufixos

Os arranjos de sufixos inicialmente foram descritos por Udi Manber e Gene Myers em [MM90] como uma alternativa mais econômica em espaço em relação às árvores de sufixos.

A ideia da estrutura é construir um arranjo contendo os sufixos de uma palavra de modo que estes sufixos estejam ordenados lexicograficamente. Uma definição mais formal desta estrutura é apresentada a seguir.

Notação 2.6 (Ordem lexicográfica)
Denote por $<$ a relação de ordem lexicográfica induzida sobre a ordem total que compreende os símbolos de Σ, isto é, se uma palavra S é lexicograficamente menor que outra palavra R, então $S < R$.

Similarmente, denotamos por $S \leq R$ se S é lexicograficamente menor ou igual a R.

Analogamente, $S \leq_n R$ denota que o prefixo de tamanho n de S é lexicograficamente menor ou igual ao prefixo de tamanho n de R. As relações $=_n$, $<_n$, $>_n$ e $\geq_n$, são definidas analogamente.

Definição 2.8 (Arranjo de sufixos)
O arranjo de sufixos $\mathcal{A}_T[0, n-1]$ de T (denotado simplesmente por $\mathcal{A}$ quando o texto referenciado for T) é um arranjo de inteiros que representa posições de início de sufixos de modo que esses sufixos estejam lexicograficamente ordenados. Ou seja, $\mathcal{A}[k] = i$ indica que o sufixo que começa na posição i do texto, ocupa a posição k na ordenação lexicográfica dos sufixos.

Assim, $T_{\mathcal{A}[0]} < T_{\mathcal{A}[1]} < T_{\mathcal{A}[2]} < \ldots < T_{\mathcal{A}[n-1]}$.

A Figura 2.3 exemplifica um arranjo de sufixos $\mathcal{A}$ para a palavra $T = acactag\$$. Conforme visto na definição: $T_{\mathcal{A}[0]} = \$ < T_{\mathcal{A}[1]} = aaacatat\$ < T_{\mathcal{A}[2]} = aacatat\$ < T_{\mathcal{A}[3]} = acaaacatat\$ < T_{\mathcal{A}[4]} < \ldots < T_{\mathcal{A}[10]} = tat\$$.

i	$T_{\mathcal{A}[i]}$	$\mathcal{A}[i]$
0	\$	10
1	$aaacatat\$$	2
2	$aacatat\$$	3
3	$acaaacatat\$$	0
4	$acatat\$$	4
5	$at\$$	8
6	$atat\$$	6
7	$caaacatat\$$	1
8	$catat\$$	5
9	$t\$$	9
10	$tat\$$	7

Figura 2.3: Arranjo de sufixos $\mathcal{A}$ para a palavra $acaaacatat\$$ e os sufixos correspondentes, indicados por $T_{\mathcal{A}[i]}$.

Nota 2.2
Pela convenção adotada, $T_{\mathcal{A}[0]} = \$$ para qualquer texto, uma vez que $\$$ é menor que qualquer outro símbolo do alfabeto Σ e não ocorre em outra posição de T, a não ser em $T[n-1]$. Consequentemente, $\mathcal{A}[0] = n - 1$.

Para ilustrar a utilidade desta estrutura de dados, tome o problema do casamento exato de padrão, onde o objetivo é detectar as ocorrências do padrão P em um texto T. Denote L_P e R_P pelo seguinte:

$$L_P = \min\{k | P \leq_m T_{\mathcal{A}[k]} \vee k = n\} \tag{2.1}$$
$$R_P = \max\{k | T_{\mathcal{A}[k]} \leq_m P \vee k = -1\} \tag{2.2}$$

De acordo com a definição de L_P e R_P, as ocorrências de P em T estão no intervalo $[L_P, R_P]$, caso $L_P \leq R_P$. Caso contrário, P não ocorre em T. Portanto, para responder onde existem as ocorrências de P em T basta achar tais índices L_P e R_P no arranjo de sufixos e reportar $\mathcal{A}[i]$ com $L_P \leq i \leq R_P$. Reportar as ocorrências dados L_P e R_P leva tempo $O(R_P - L_P + 1)$.

Como a estrutura é ordenada lexicograficamente, a busca binária pode ser efetuada para encontrar esses índices, conforme ilustrado pelo Algoritmo 1.

Algoritmo 1 Busca em Arranjo de Sufixos para computar os índices L_P e R_P.

Input A,T,P
Output (L_P, R_P)

```
function BUSCABINARIA(A, T, P)
    L_P ← BUSCAL_P(A, T, P, 0, n − 1)
    R_P ← BUSCA R_P(A, T, P, L_P, n − 1)
    return (L_P, R_P)
end function

function BUSCAL_P(A, T, P, L, R)
    if P ≤_m T_A[L] then return L
    else if P >_m T_A[R] then return R + 1
    end if
    while R − L > 1 do
        M ← ⌊(L+R)/2⌋
        if P ≤_m T_A[M] then
            R ← M
        else
            L ← M
        end if
    end while
    return R
end function

function BUSCAR_P(A, T, P, L, R)
    if P < T_A[L] then return L − 1
    else if P ≥_m T_A[R] then return R
    end if
    while R − L > 1 do
        M ← ⌊(L+R)/2⌋
        if T_A[M] ≤_m P then
            L ← M
        else
            R ← M
        end if
    end while
    return L
end function
```

A busca binária ilustrada anteriormente é efetuada em tempo $O(m \log n)$, visto que no pior caso, em cada iteração da busca binária, compara-se $O(m)$ caracteres para decidir em qual partição a busca deve prosseguir. Como existem no máximo $O(\log n)$ iterações na busca binária, o tempo está justificado.

Tomando como exemplo o arranjo da figura 2.3, e tomando o padrão como $P = aca$, é obtido que $L_P = 3$ e $R_P = 4$, portanto as ocorrências se encontram no intervalo $[3, 4]$ do arranjo, o que corresponde às posições 0 e 2 do texto original, visto que $\mathcal{A}[3] = 0$ e $\mathcal{A}[4] = 4$.

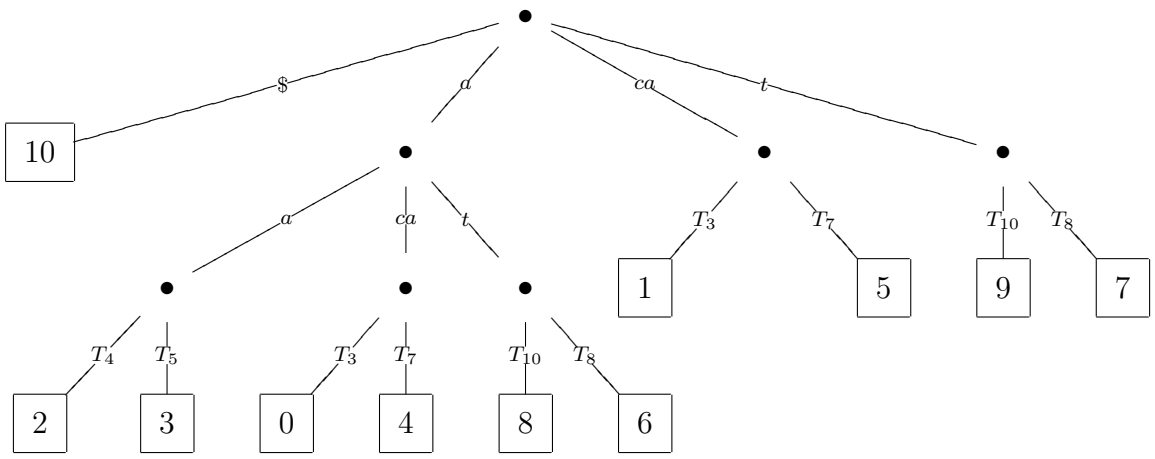
2.3.1 Espaço e tempo

A construção de arranjos de sufixos pode ser efetuada em tempo $O(n)$, como visto em [KSB06], e como se trata de um arranjo de inteiros, a estrutura pode ser representada utilizando $O(n \log n)$ bits. Na prática, gasta-se 4 bytes para cada símbolo da entrada, isto é, existe um fator de $4\times$ associado à estrutura de dados, conforme descrito em [MM90],[AKO02] e [AKO04].

A constante envolvida em arranjos de sufixos é consideravelmente menor do que a presente em árvores de sufixos, isto é, é um fator de $4\times$ contra um fator de $15\times$. Em comparação, na indexação do genoma humano, $12GB$ de memória seria gasta com a primeira estrutura enquanto $45GB$ seria usado na segunda. Uma redução substancial.

Como a estrutura é representada na forma de um arranjo, os dados estão em posições contíguas de memória. Por isto, a estrutura possui uma boa localidade de referência, isto é, os dados acessados encontram-se frequentemente em áreas próximas da memória, possibilitando assim um melhor uso da memória *cache*, como descrito em [AKO02]. Por conta desta característica, algoritmos para construção direta de arranjos de sufixos são mais rápidos do que algoritmos para construção de árvores de sufixo.

Note que os arranjos de sufixos correspondem apenas às folhas das árvores de sufixos associadas, como exemplificado pela Figura 2.4.



i	$T_{\mathcal{A}[i]}$	$\mathcal{A}[i]$
0	\$	10
1	$aaacatat$\$	2
2	$aacatat$\$	3
3	$acaaacatat$\$	0
4	$acatat$\$	4
5	at\$	8
6	$atat$\$	6
7	$caaacatat$\$	1
8	$catat$\$	5
9	t\$	9
10	tat\$	7

Figura 2.4: Arranjos de sufixos contém a informação sobre a ordem lexicográfica das folhas das árvores de sufixos.

Contudo, arranjos de sufixos podem ser enriquecidos com informação adicional, como a informação de LCP (Longest Common Prefix) que informa o tamanho do maior prefixo compartilhado por sufixos correspondentes às posições de uma entrada e da seguinte do arranjo. Arranjos de sufixos acompanhados desta informação extra são chamados de arranjos de sufixos enriquecidos.

2.3.2 Arranjos de sufixos enriquecidos

A informação de LCP é essencial para permitir consultas mais elaboradas utilizando arranjos de sufixos. Formalmente, define-se a informação da seguinte maneira:

Definição 2.9 (LCP)

$$LCP(i) = \left\{ \begin{array}{ll} \max\{k | T_{A[i]} =_k T_{A[i+1]}\}, & 0 \leq i < n-1 \\ 0, & i = n-1 \end{array} \right\}$$

A Tabela 2.1 ilustra a informação de LCP , para o texto $T = acaaacatat\$$. A informação de LCP, assim como o arranjo de sufixos, pode ser armazenada em um arranjo de inteiros.

Tabela 2.1: Arranjo de sufixos enriquecido para $T = acaaacatat\$$

i	$T_{\mathcal{A}[i]}$	$\mathcal{A}[i]$	$LCP[i]$
0	$\$$	10	0
1	$aaacatat\$$	2	2
2	$aacatat\$$	3	1
3	$acaaacatat\$$	0	3
4	$acatat\$$	4	1
5	$at\$$	8	2
6	$atat\$$	6	0
7	$caaacatat\$$	1	2
8	$catat\$$	5	0
9	$t\$$	9	1
10	$tat\$$	7	0

Para representar a tabela de LCP, no pior caso, gasta-se tanto espaço quanto para representar o arranjo A. Na prática, o custo associado é de 4 bytes por símbolo de entrada para representar a tabela de LCP, no pior caso. Isto é, para representar ambos $\mathcal{A}$ e LCP , gasta-se 8 bytes por símbolo de T, ou seja, existe um fator de $8\times$ para representar essas duas informações, como visto em [MM90] e [AKO02].

Uma vez introduzido o conceito de LCP, uma técnica é necessária para construir essa informação. De maneira ingênua, ao realizar comparações explícitas entre sufixos, seria necessário tempo $O(n^2)$ no pior caso, visto que existem n sufixos e cada um tem tamanho $O(n)$. Entretanto, é possível construir a tabela de LCP em tempo $O(n)$, mas antes algumas propriedades precisam ser introduzidas.

2.3.3 Propriedades sobre LCP

Uma definição estendida de LCP indica o maior prefixo comum dos sufixos contidos no intervalo $[i, j]$ do arranjo, isto é:

Definição 2.10 (LCP sobre intervalo)

$$LCP(i,j) = \left\{ \begin{array}{l} \max\{k | T_{\mathcal{A}[i]} =_k T_{\mathcal{A}[i+1]} =_k \ldots =_k T_{\mathcal{A}[j]}\}, \quad 0 \leq i < j \leq n-1 \\ LCP(i), \quad i = j \end{array} \right\}$$

Das Definições 2.9 e 2.10, nota-se que $LCP(i, j)$ corresponde ao valor mínimo de $LCP(k)$ encontrado para um k no intervalo $[i, j-1]$.

Teorema 2.1

$$LCP(i,j) = \min_{i \leq k \leq j-1} \{LCP(k)\}, \quad i < j.$$

Demonstração
O caso em que $i = j$ vem da Definição 2.10.

Antes de demonstrar, é necessário ressaltar que $=_k$ é uma relação de equivalência, ou seja, é uma relação que tem propriedade reflexiva, simétrica e transitiva.

A demonstração para $i < j$ é por indução. Para o caso base, se $j = i + 1$, é observado que $LCP(i, j) = LCP(i) = \min\{LCP(i)\}$.

Suponha que $LCP(i, j-1) = \min_{i \leq l \leq j-2} \{LCP(l)\} = m$, para $i < l < j-1$. Logo os sufixos concentrados no intervalo $[i, j-1]$ compartilham um prefixo de tamanho m.

Caso $LCP(j-1) \geq m$ então $LCP(i, j) = m$, pois $LCP(i, l) = m$ para $i < l < j-1$, isto é,$LCP(i, j)$ não pode ser de tamanho maior que m. Se $LCP(j-1) < m$, então $LCP(i, j) = LCP(j-1)$, pois o último sufixo $T_{\mathcal{A}[j]}$ compartilha um prefixo de tamanho apenas $LCP(j-1)$ com o penúltimo sufixo $T_{\mathcal{A}[j-1]}$ e os demais, por transitividade, não podem compartilhar mais que $LCP(j-1)$ símbolos com o último sufixo. Em ambos os casos, $LCP(i,j) = \min_{i \leq k \leq j-1} \{LCP(k)\}$.

□

2.3.4 Construindo LCP em tempo $O(n)$

Um método para construção de LCP em tempo $O(n)$ pode ser encontrado no trabalho de Kasai *et al.* [KLA+01]. Ele poupa comparações desnecessárias utilizando informações anteriores baseadas nas propriedades da informação de LCP .

Este método precisa de uma tabela extra que corresponde ao inverso dos arranjos de sufixos, isto é:

Definição 2.11
O inverso dos arranjos de sufixos, denotado por $\bar{\mathcal{A}}$ possui a seguinte propriedade:

$$\bar{\mathcal{A}}[\mathcal{A}[i]] = i, \quad 0 \leq i \leq n-1$$

Segundo a Definição 2.11 , enquanto $\mathcal{A}[i]$ informa o sufixo $T_{\mathcal{A}[i]}$ que se encontra na posição i na ordem lexicográfica em relação ao restante dos sufixos, $\bar{\mathcal{A}}[i]$ informa a posição lexicográfica que o sufixo T_i do texto possui em relação aos demais sufixos do texto.

Alguns resultados são importantes para assegurar a correção do método.

Lema 2.1 (Posição relativa de sufixos sem o primeiro símbolo)
Se $LCP\,[i] > 1$*, então:*

$$\bar{\mathcal{A}}[\mathcal{A}[i]+1] < \bar{\mathcal{A}}[\mathcal{A}[i+1]+1]$$

Intuitivamente, o Lema 2.1 afirma que se há dois sufixos adjacentes no arranjo que compartilham um prefixo de tamanho pelo menos 2, ao desconsiderar o primeiro símbolo de ambos os sufixos, eles tem que possuir a mesma ordem relativa.

Demonstração
Tome $R = T_{\mathcal{A}[i]}$ *e* $S = T_{\mathcal{A}[i+1]}$*. Obrigatoriamente,* R *compartilha um prefixo de tamanho pelo menos* 2 *com* S*, pois* $LCP\,[i] > 1$*. Logo,* R *e* S *são da forma* $R = abR'$ *e* $S = abS'$ *respectivamente. Seja* $p = \bar{\mathcal{A}}[\mathcal{A}[i]+1]$ *e* $q = \bar{\mathcal{A}}[\mathcal{A}[i+1]+1]$

De acordo com $\mathcal{A}$ *e* $\bar{\mathcal{A}}$*,* $T_{\mathcal{A}[p]}$ *tem a forma* bR' *e* $T_{\mathcal{A}[q]}$ *tem a forma* bS'*. Consequentemente* $p < q$*, já que* $R' < S'$*.*
□

Lema 2.2
Se $LCP\,[i] > 1$ *então* $\max\{k|(T_{\bar{\mathcal{A}}[\mathcal{A}[i]+1]}) =_k T_{\bar{\mathcal{A}}[\mathcal{A}[i+1]+1]}\} = LCP\,[i] - 1$.

O Lema 2.2 afirma que se dois sufixos compartilham mais de um símbolo em comum, ao desconsiderar o primeiro símbolo de ambos, os sufixos restantes compartilham a quantidade que compartilhavam menos uma unidade.

Demonstração
Tome R*,* S*,* p *e* q *com a mesma semântica da demonstração anterior. Se* $LCP\,[i] > 1$*, então a afirmação* $LCP\left[\bar{\mathcal{A}}[i+1]\right] = LCP\,[i] - 1$ *vale, pois* $T_{\mathcal{A}[p]}$ *e* $T_{\mathcal{A}[q]}$ *tem as forma* bR' *e* bS' *respectivamente, e portanto, compartilham os mesmos caracteres com exceção do primeiro símbolo retirado de ambos* R *e* S*.*
□

Teorema 2.2
Sejam $LCP\,[i] > 1$ *,* $p = \bar{\mathcal{A}}[A[i]+1]$ *e* $q = \bar{\mathcal{A}}[\mathcal{A}[i+1]+1]$*. Então* $LCP\,[p] \geq LCP\,[i] - 1$.

O Teorema 2.2 expressa que se um sufixo S e o seu sufixo adjacente no arranjo compartilham pelo menos 2 símbolos, então o mesmo sufixo ignorando o primeiro símbolo, S', tem que compartilhar ao menos a mesma quantidade menos uma unidade com o sufixo adjacente correspondente no arranjo

Demonstração
Pelo Lema 2.1, $p < q$*. Note que* q *não é necessariamente* $p+1$*. Pelo Lema 2.2,* $LCP\,[p] \geq LCP\,[i] - 1$*, visto que* $LCP(p,q) = LCP(i,i+1) - 1$ *e* $LCP(p,p+1) \geq LCP(p,q)$*, já que* $p+1 \leq q$ *e pela própria definição de* $LCP(p,p+1)$ *e* $LCP(p,q)$*.*
□

Em resumo, o Teorema 2.2 informa que podemos economizar $LCP\,[i] - 1$ comparações ao considerarmos o sufixo sem o primeiro símbolo e o seu adjacente no arranjo.

Graficamente, o Teorema 2 é representado pela Figura 2.5. Considerando a mesma figura, ao considerar a informação de $LCP\,[i]$, conclui-se que $LCP\,[p] = LCP\,[i] - 1$ mais o número de símbolos que casam entre W e V.

i	$T_{\mathcal{A}[i]}$
0	
$\vdots$	
i	$abSW$
$i+1$	$abSU$
$\vdots$	
p	bSW
$p+1$	bSV
$\vdots$	
q	bSU
$\vdots$	
$n-1$	

Figura 2.5: Representação gráfica do Teorema 2.2.

A partir destes resultados, deriva-se o Algoritmo 2, que computa LCP em ordem crescente de tamanho de sufixo.

Algoritmo 2 Computação de LCP de acordo com o método de Kasai *et al.* [KLA+01].

```
Input  A, Ā, T
Output LCP
  function COMPARA(i, j, h)
      return max{k|T_{i+h} =_k T_{j+h}}
  end function

  function COMPUTALCP
      prefix ← 0
      for i ← 0 to n − 1 do    // Computa LCP [Ā[i]]
          if Ā[i] < n − 1 then
              adjacent ← A[Ā[i] + 1]
              prefix ← prefix + COMPARA(i, adjacent, h)
              LCP [Ā[i]] ← prefix
              if prefix > 1 then
                  prefix − −
              end if
          else
              LCP [Ā[i]] ← 0
          end if
      end for
  end function
```

O Algoritmo 2 leva tempo $O(n)$ pois são feitas no máximo $2n$ comparações, já que o laço interno não executa mais vezes que o laço externo, visto que há economia de comparações devido ao mecanismo de memória do método, diferentemente do que ocorre no método força-bruta, que leva tempo $O(n^2)$.

A informação de LCP possui uma propriedade essencial. Pois de maneira implícita, codifica a topologia da árvore de sufixos associada. Portanto, a partir desta informação, é possível inferir a topologia da árvore de sufixos associada.

Com a informação de LCP e conhecimento de como navegar nessa topologia descrita por essa informação, o arranjo de sufixos ganha o mesmo poder de representação de uma árvore de sufixos e deixa de representar somente as folhas na ordem lexicográfica, conforme descrito em [AKO04]. Nessa seguinte parte, será descrito como o LCP pode ser usado para representar a topologia da árvore.

2.3.5 Emulando árvores de sufixos com arranjos de sufixos enriquecidos

Um conceito extremamente importante é o de ℓ-intervalo sobre a informação de LCP , como veremos, cada intervalo deste tipo tem correspondência de um para um com os nós internos da árvore de sufixos associada.

Definição 2.12 (ℓ-intervalo)
Um ℓ-intervalo é um intervalo $[i, j]$ sobre LCP que tem as seguintes propriedades:

1. $i = 0 \vee LCP[i-1] < \ell,\ 0 < i \leq n-1$.

2. $LCP[k] \geq \ell,\ i \leq k < j$.

3. $LCP[k] = \ell$, *para* ***algum*** $i \leq k < j$.

4. $j = n - 1 \vee LCP[j] < \ell,\ 0 < j \leq n - 1$.

Um intervalo $[i, j]$ *que é um* ℓ*-intervalo é denotado por* ℓ*-*$[i, j]$.
Um ℓ*-índice é uma posição* $i \leq k \leq j$ *de* ℓ*-*$[i, j]$ *tal que* $LCP\,[k] = \ell$.

A Definição 2.12 considera que um ℓ-intervalo é um $[i, j]$ tal que todos os sufixos $T_{\mathcal{A}[i]}, T_{\mathcal{A}[i+1]}, \ldots, T_{\mathcal{A}[j]}$ compartilham um prefixo de tamanho **exatamente** ℓ, pois, de acordo com a Definição 2.10, $LCP(i, j) = \ell$. Adicionalmente, também é capturada a noção de maximalidade, isto é, o intervalo $[i, j]$ não pode ser estendido para a esquerda e nem para a direita, portanto, o intervalo é maximal e o maior intervalo que compartilha o prefixo de tamanho ℓ.

Como conclusão, ao tomar a Nota 2.1, é observado que um ℓ-intervalo corresponde exatamente à um nó interno da árvore de sufixos associada, pois o intervalo corresponde à um conjunto de folhas $\{\mathcal{A}[i], \mathcal{A}[i + 1], \ldots, \mathcal{A}[j]\}$ que compartilham um prefixo maximal de tamanho ℓ.

É importante notar que podem existir intervalos com a mesma propriedade contidos em um m-intervalo, $m > \ell$. Contudo, mais uma vez, a propriedade de maximalidade é visada. Mais formalmente, isto corresponde à definição de intervalo filho.

Definição 2.13 (Intervalo filho)
Seja m*-*$[k, l]$ *um intervalo contido em* ℓ*-*$[i, j]$*, isto é,* $i \leq k \leq l \leq j$ *com* $m > \ell$*. Se não existe algum outro* k*-intervalo além de* ℓ*-*$[i, j]$ *que contenha* m*-*$[k, l]$*, dizemos que* m*-*$[k, l]$ *é um intervalo filho de* ℓ*-*$[i, j]$.

Um intervalo do tipo $[i, i]$ *com* $0 \leq i \leq n - 1$ *também é considerado um intervalo filho, apesar de não ser um* ℓ*-intervalo, desde que não exista outro intervalo filho que englobe este intervalo.*

Note que obrigatoriamente não é possível que $k = i$ e $l = j$ simultaneamente de acordo com a Definição 2.12.

A Definição 2.13 informa que, se existe um intervalo filho, a árvore de sufixos associada terá um nó interno, que corresponde a m-$[k, l]$ e que tem como ancestral imediato outro nó interno, correspondente ao intervalo pai ℓ-$[i, j]$.

Para exemplificar as Definições 2.12 e 2.13, tome a Figura 2.6.

i	$T_{\mathcal{A}[i]}$	$\mathcal{A}[i]$	$LCP[i]$
0	\$	10	0
1	$aaacatat\$$	2	2
2	$aacatat\$$	3	1
3	$acaaacatat\$$	0	3
4	$acatat\$$	4	1
5	$at\$$	8	2
6	$atat\$$	6	0
7	$caaacatat\$$	1	2
8	$catat\$$	5	0
9	$t\$$	9	1
10	$tat\$$	7	0

- $[1, 6]$ é um 1-intervalo.
- $[1, 2]$ é um 2-intervalo.
- $[1, 3]$ **não** é um 1-intervalo, pois não é maximal.
- $[7, 8]$ é um 2-intervalo.
- $[3, 4]$ é um 3-intervalo **filho** de $[1, 6]$.
- Cada intervalo ℓ tem correspondência de 1 para 1 com um nó interno da árvore de sufixos.
- Os intervalos filhos são separados por ℓ-índices.

Figura 2.6: ℓ-intervalos e intervalhos filhos.

A Figura 2.7 apresenta a relação existente entre ℓ-intervalos e nós internos de uma árvore de sufixos. O arranjo de sufixos corresponde às folhas desta árvore conceitual.

i	$T_{\mathcal{A}[i]}$	$\mathcal{A}[i]$	$LCP[i]$
0	\$	10	0
1	$aaacatat\$$	2	2
2	$aacatat\$$	3	1
3	$acaaacatat\$$	0	3
4	$acatat\$$	4	1
5	$at\$$	8	2
6	$atat\$$	6	0
7	$caaacatat\$$	1	2
8	$catat\$$	5	0
9	$t\$$	9	1
10	$tat\$$	7	0

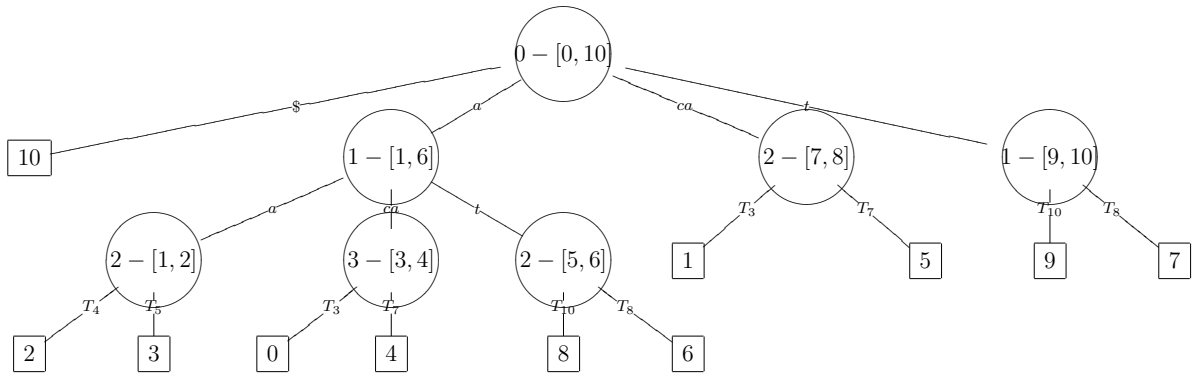


Figura 2.7: Arranjo de sufixos com LCP e a árvore de sufixos conceitual, codificada implicitamente pelos ℓ-intervalos de LCP.

Nota 2.3
Apesar de formalmente não serem ℓ-intervalos, os intervalos $[i, i]$ são considerados intervalos-filhos para lidar com os algoritmos de maneira mais intuitiva e representam as folhas da árvore, isto é, $\mathcal{A}[i]$.

Com a obtenção dos ℓ-intervalos e seus intervalos filhos, replica-se uma árvore de sufixos **conceitual**, que contém a mesma informação da árvore de sufixos associada, conforme representada pela Figura 2.8.

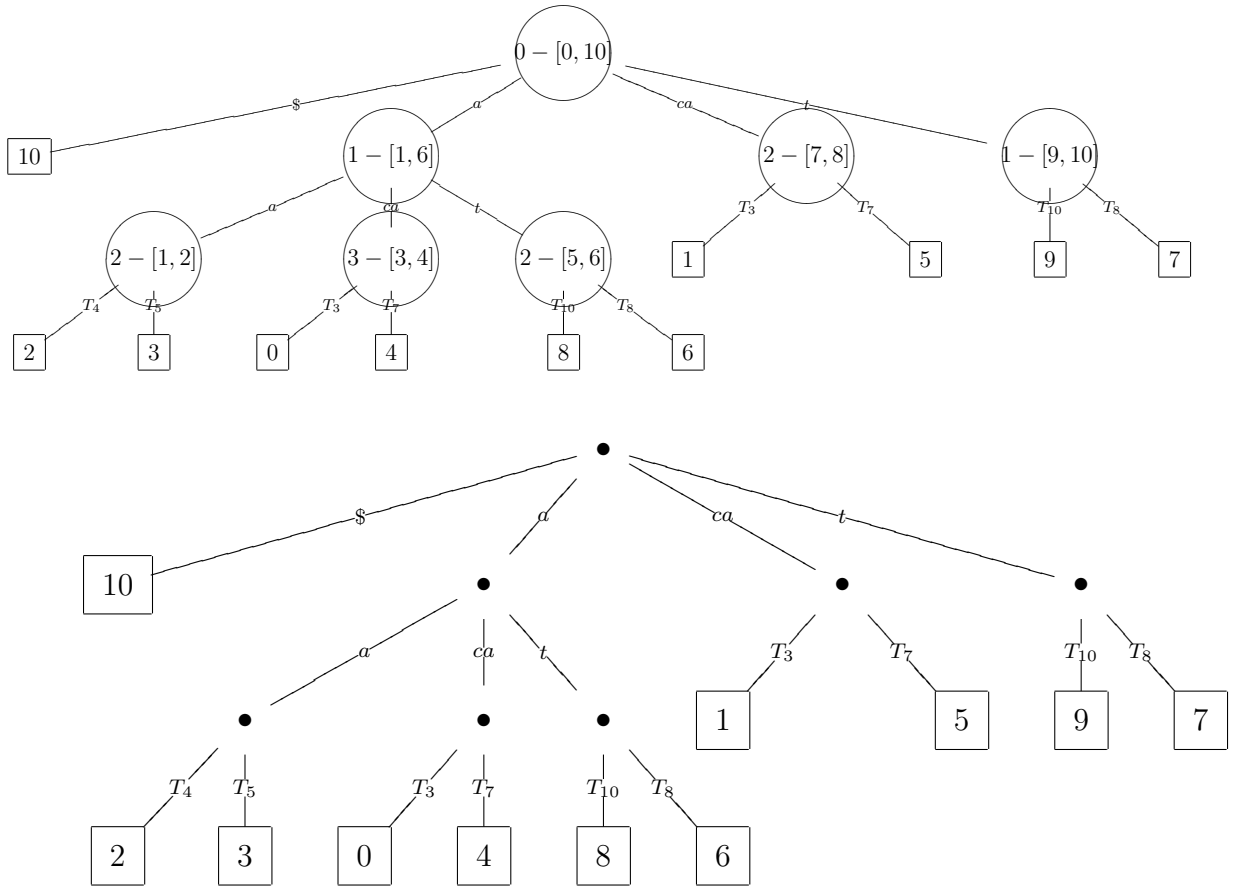


Figura 2.8: A topologia da árvore conceitual (acima) é a mesma da árvore de sufixos associada (abaixo).

Uma vez conhecido como obter a topologia da árvore através dos ℓ-intervalos, é necessário identificá-los, bem como os seus intervalos filhos, para então, conseguir emular o percurso da árvore de sufixos associada, sem que a mesma árvore necessite estar representada explicitamente.

Uma forma de identificar os intervalos filhos é examinando os ℓ-índices de um dado ℓ-intervalo.

Teorema 2.3 (ℓ-índices e intervalos-filhos)
Seja ℓ-$[i, j]$ e tome como $\imath_1 < \imath_2 < \ldots < \imath_k$ os ℓ-índices desse intervalo em ordem crescente de posição. Os intervalos filhos deste ℓ-intervalo, são dados por:

$$[i, \imath_1], [\imath_1 + 1, \imath_2], \ldots, [\imath_k + 1, j]$$

Demonstração
Seja $[l, r]$ *um dos intervalos* $[i, \imath_1], [\imath_1 + 1, \imath_2], \ldots, [\imath_k + 1, j]$. *Se* $[l, r]$ *é do tipo* $[k, k]$, *para* $i \leq k \leq j$ *então* $[l, r]$ *é um intervalo filho. Suponha então que* $[l, r]$ *é um* m*-intervalo. Essa suposição é verdadeira, pois* m-$[l, r]$ *não contém um* ℓ*-índice, além disso* $m > \ell$, *pela definição de* ℓ*-intervalo.*

O ponto é, não pode existir um intervalo filho de $[i, j]$ *que englobe* $[l, r]$, *pois* $[l, r]$ *não pode ser estendido, visto que suas extremidades são* ℓ*-índices.*

Esse argumento pode ser estendido para todos os intervalos e, portanto, não existem outros intervalos filhos além dos intervalos $[i, \imath_1], [\imath_1 + 1, \imath_2], \ldots, [\imath_k + 1, j]$.
□

A partir do Teorema 2.3, ao encontrar os ℓ-índices, é possível identificar os intervalos filhos, e assim emular um percurso *top-down* na árvore de sufixos associada.

Com a ajuda de mecanismos como consultas de RMQ (range-minimum-queries), construção de tabelas extra com pré-processamento ou até mesmo representação por parênteses balanceados, é possível identificar os ℓ-índices e navegar na árvore codificada implicitamente pela informação de LCP, como notado por Abouelhoda *et al.* [AKO04].

Capítulo 3

Índices comprimidos

Os índices comprimidos consomem $o(n \log n)$ bits de espaço para sua representação, assintoticamente menos do que arranjos e árvores de sufixos, ambos estruturas não-comprimidas.

As estruturas comprimidas baseiam-se em propriedades inerentes às estruturas não-comprimidas para alcançar a compressibilidade desejada. Uma destas estruturas é o **arranjo de sufixos comprimido**.

A Figura 3.1 ilustra a revisão da literatura em relação as estruturas de dados comprimidas e não-comprimidas, o que ajuda a localizar o nosso trabalho em relação aos demais presentes. Ao examinar a figura, temos que inicialmente as árvores de sufixos foram propostas na década de 70 por Weiner [Wei73], dando origem a vários outros trabalhos, como os trabalhos de McCreight [McC76], Ukkonen [Ukk95] e Farach [Far97]. No início da década de 90, Manber e Myers verificaram que a árvore de sufixos consumia muito espaço na prática e então, propuseram os arranjos de sufixos [MM90]. Neste trabalho os autores também introduzem a noção de LCP. Uma década depois, Kasai *et al.* propuseram um método eficiente para cálculo de LCP em tempo $O(n)$. Ainda nesta linha de pesquisa, Abouelhoda *et.al* verificaram que todos os problemas tratáveis por árvores de sufixos são tratáveis também por arranjos de sufixos enriquecidos usando assintoticamente o mesmo tempo [AKO04]. No início dos anos 2000, Grossi e Vitter propuseram uma alternativa melhor espaço-eficiente em relação aos arranjos de sufixos, visto que a estrutura ainda consumia muito espaço. A estrutura alternativa foi chamada de arranjos de sufixos comprimidos [GV05]. Paralelamente, Ferragina e Manzini desenvolveram o índice-FM [FM05]. Além disso Sadakane propôs o primeiro método para construção de LCP comprimido [Sad02]. Como era possível comprimir a informação de arranjo de sufixos e LCP surgiram os primeiros trabalhos de árvore de sufixos comprimidas [Sad07]. Trabalhos subsequentes, tais como [FMN08], [CN10] e [OFG10] trouxeram implementações e aspectos teóricos à essa estrutura de dados e por fim, a nossa implementação também visa contribuir com esta estrutura de dados.

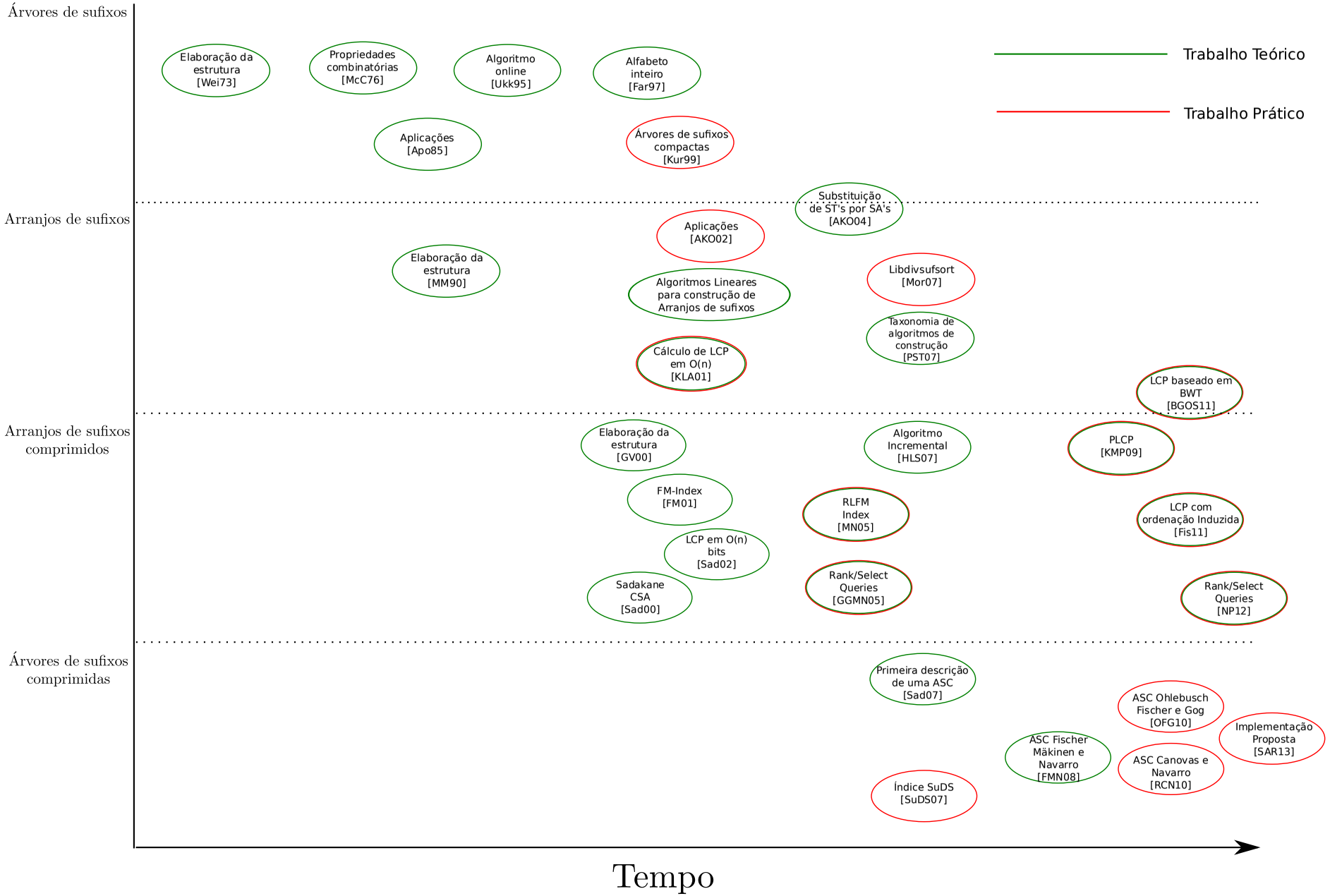


Figura 3.1: Revisão da literatura.

Para introduzir tal conceito, é necessário introduzir algumas noções básicas para o entendimento dessa estrutura de dados.

3.1 Noções fundamentais

Nesta seção conceitos fundamentais para tratar de índices comprimidos serão abordados.

3.1.1 Entropia empírica

De acordo com Ferragina e Manzini [FM05], se existe um índice $\mathcal{I}$ baseado em um texto T de forma que $\mathcal{I}$ contém informação suficiente para recuperar T, então o tamanho da representação de $\mathcal{I}$ tem como cota inferior o tamanho da representação de T, que pode ser mensurado pela entropia empírica.

A entropia empírica é definida com relação às frequências dos símbolos observados no texto T. Mais formalmente, a entropia empírica é definida como:

Definição 3.1 (Entropia empírica)

$$H_0(T) = -\sum_{x \in \Sigma} \frac{|T|_x}{n} \log \left(\frac{|T|_x}{n} \right) \tag{3.1}$$

O valor nH_0 representa o tamanho da representação da saída de uma técnica de compressão que utiliza $-\log \left(\frac{|T|_x}{n} \right)$ bits para codificar o símbolo x.

H_0 também é chamada de entropia de ordem zero. Esse conceito pode ser generalizado para entropia de ordem k se a codificação de um símbolo não depende apenas da frequência deste, mas sim dos k símbolos que o precedem no texto.

Por definição, $H_0 \leq \log \sigma$, como mencionado em [HLS$^+$07]. Se $\sigma \in o(n)$, então $H_0 \in o(\log n)$.

3.1.2 Operações em vetores de bit

Duas operações essenciais em vetores de bits são as operações de $Rank$ e $Select$. As operações sobre um vetor de bits $V[0, n-1]$, são definidas da seguinte forma:

Definição 3.2 ($Rank$)

$$\begin{aligned} Rank_0(V, r) &= \textit{número de } 0\textit{s no prefixo } V[0, r] \\ Rank_1(V, r) &= \textit{número de } 1\textit{s no prefixo } V[0, r] \end{aligned} \tag{3.2}$$

Definição 3.3 ($Select$)

$$\begin{aligned} Select_0(V, r) = j, \textit{ tal que } V[j] \textit{ é o } (r-1)\textit{-ésimo } 0 \\ Select_1(V, r) = j, \textit{ tal que } V[j] \textit{ é o } (r-1)\textit{-ésimo } 1 \end{aligned} \tag{3.3}$$

Em suma, $Rank(V, r)$ informa a quantidade de bits 0 ou 1 no prefixo $V[0, r]$ do vetor de bits V. Já $Select(V, r)$ informa a posição do $(r-1)$-ésimo bit desejado. Note que $Select(V, r)$ está definido para $r \in [1, n]$.

Por exemplo, se $V = (0, 1, 0, 0, 1, 0, 0, 1)$, tem-se o seguinte resultado produzido pelas consultas abaixo:

- $Select_0(V, 1) = 0$.
- $Rank_0(V, 0) = 1$.
- $Rank_0(V, 7) = 5$.
- $Rank_1(V, 7) = n - Rank_0(V, 7) = 3$.
- $Select_1(V, 3) = 7$.

É possível realizar consultas de $Rank$ e $select$ em tempo $O(1)$, dado um pré-processamento de tempo $O(n)$ em cima do vetor de bits V, de acordo com Clark [Cla97] e Pagh [Pag02]. Entretanto, a solução de tempo constante não é adotada na prática, sendo preferível soluções com tempo assintoticamente maior, conforme será detalhado no Capítulo 4.

3.2 Arranjos de sufixos comprimidos

O arranjo de sufixos comprimido, introduzido por Grossi e Vitter [GV05], foi o primeiro índice a conseguir usar $o(n \log n)$ bits para representação. Especificamente, este índice consome apenas $O(nH_0)$ bits para sua representação e é altamente aceitável assumir que $\sigma \in o(\log n)$. Muitas vezes é até razoável assumir que $\sigma \in O(1)$, se o alfabeto considerado for pequeno, como $\Sigma = \{a, c, g, t\}$, o alfabeto de moléculas de ADN, por exemplo.

Para atingir essa cota de espaço, o método proposto consiste em amostrar apenas algumas entradas do arranjo de sufixos original e recuperar as entradas não amostradas através de um mecanismo que efetua computação. É um claro exemplo de troca de espaço por tempo.

A função Ψ é o mecanismo por trás do arranjo de sufixos comprimidos responsável pela computação na recuperação de entradas não amostradas. Ela é definida formalmente como:

Definição 3.4 (Função Ψ)
A função Ψ é definida como:

$$\Psi(i) = j, \quad \mathcal{A}[j] = (\mathcal{A}[i] + 1) \mod n$$

Note que $\Psi(i)$ equivale à $\bar{\mathcal{A}}[(\mathcal{A}[i]+1) \mod n)]$. Ψ pode ser visto como um conceito similar ao dos *links* de sufixos em uma árvore de sufixos, segundo a Definição 2.6.

A função $\Psi(i)$ fornece a posição lexicográfica do sufixo $T_{(i+1) \mod n}$ em relação a todos os outros sufixos de T. Com esta função, o ato de caminhar pelo arranjo de sufixos torna-se possível através da aplicação sucessiva da mesma. A Tabela 3.1 ilustra a função Ψ.

Notação 3.1
O acesso à entrada da estrutura de dados que armazena $\Psi(i)$ é denotado por $\Psi[i]$.

Notação 3.2
A versão iterada de Ψ, denotada por $\Psi^k(i)$ corresponde à $\underbrace{\Psi(\Psi(\ldots(\Psi(i)}_{k}\ldots))$. De maneira análoga, $\Psi^k[i]$, corresponde à $\underbrace{\Psi[\Psi[\ldots[\Psi[i]}_{k}\ldots]]$.

Ingenuamente, Ψ pode ser representada utilizando $O(n \log n)$ bits, como um arranjo de inteiros comum. No entanto, isso não representaria ganho algum de espaço. Contudo, Ψ possui uma propriedade especial que possibilita que esta função possa ser comprimida de maneira eficiente. A propriedade baseia-se no fato de, se consideradas apenas as entradas do arranjo que começam com um mesmo símbolo, Ψ forma uma sequência crescente. No exemplo da Tabela 3.1, ao considerar os sufixos que começam com o símbolo a, Ψ forma a sequência crescente $\Psi(1) = 4$, $\Psi(2) = 5$ e $\Psi(3) = 6$.

Teorema 3.1 (Ψ forma uma sequência crescente)
Considerando apenas sufixos que começam com o mesmo símbolo no arranjo de sufixos, Ψ forma uma sequência crescente.

Demonstração
Seja $[i, j]$ um intervalo qualquer compreendido por Ψ de forma que todos os sufixos nesse intervalo começam com o mesmo símbolo. Se $i = j$, então a propriedade vale. Se $i < j$, $\Psi(i) < \Psi(j)$ pela ordem lexicográfica, visto que $T_{\mathcal{A}[i]} < T_{\mathcal{A}[j]}$ e ao retirar o primeiro símbolo de ambos os sufixos, a propriedade se mantém, uma vez que o primeiro símbolo de ambos os sufixos é igual, e portanto, $T_{\mathcal{A}[i]+1} < T_{\mathcal{A}[j]+1}$.
□

Tabela 3.1: Função Ψ para $T = acaaacatat\$$.

i	$T_{\mathcal{A}[i]}$	$\mathcal{A}[i]$	$\Psi[i]$
0	$\$$	10	3
1	$aaacatat\$$	2	2
2	$aacatat\$$	3	4
3	$acaaacatat\$$	0	7
4	$acatat\$$	4	8
5	$at\$$	8	9
6	$atat\$$	6	10
7	$caaacatat\$$	1	1
8	$catat\$$	5	6
9	$t\$$	9	0
10	$tat\$$	7	5

Uma vez que Ψ tem propriedade crescente para sufixos que começam com o mesmo símbolo, é possível codificar a informação de maneira compacta, como demonstra o Teorema 3.2.

Teorema 3.2 (Codificando uma sequência crescente de maneira eficiente [GV05])
Uma sequência crescente de s inteiros representáveis com palavras de w bits, com $s < 2^w$, pode ser codificada em $s(2 + w - \lfloor \log s \rfloor)$ bits mantendo o acesso constante a cada um dos elementos da sequência.

Demonstração ([GV05])
Para representar a sequência eficientemente, usa-se o código de Rice juntamente com operações de $Rank$ e $Select$.

Primeiramente, são retirados os primeiros $z = \lfloor \log s \rfloor$ bits mais significativos de cada inteiro da sequência. Cada sequência de bits, denominadas de quocientes, são denotadas respectivamente por $q_1, \ldots, q_s$. Pela sequência ser crescente, temos que $0 \leq q_1 \leq \ldots \leq q_s$. Sejam $r_1, \ldots, r_s$ os restos obtidos ao deletar os z primeiros bits mais significativos de cada inteiro, isto é, o que sobrou após a retirada dos quocientes.

Os quocientes são armazenados em um vetor de bits Q ao representar em unário as diferenças $q_1 - 0$, $q_2 - q_1$,... $q_s - q_{s-1}$. O código unário de um inteiro k, por sua vez, é representado como 0^k1. Assim, para recuperar o i-ésimo quociente, basta tomar $Select_1(i) - i + 1$, visto que esse resultado fornece a soma de zeros da sequência que por sua vez corresponde ao quociente q_i.

A tabela R, contendo os restos, é simplesmente a concatenação dos restos $r_0 r_1 \ldots r_s$, que podem ser recuperados em $O(1)$ utilizando operações a nível de bit.

Visivelmente, R gasta $s(w - z)$ bits. Já Q pode ser representado com no máximo $2s$ bits, visto que a soma das diferenças entre os quocientes (o número to tal de 0s não pode exceder s e necessitamos de s 1s para a representação em unário.

A representação de Q e R é possível em $2(s + w - \lfloor \log s \rfloor)$ bits, como q_i e r_i podem ser recuperadas em tempo constante, é possível obter o i-ésimo número da sequência calculando $q_i 2^{w-z} + r_i$, também tem tempo constante.
□

Já que sequências crescentes podem ser representadas de maneira eficiente, Ψ também pode ser representado utilizando apenas $O(nH_0)$ bits.

Teorema 3.3 (Ψ pode ser representado sucintamente)
A função Ψ pode ser armazenada utilizando apenas $O(nH_0)$ bits.

Demonstração ([HLS+07])
Cada sequência crescente de Ψ que começa com c consome $2(|T|_c + \log n - \lfloor \log |T|_c \rfloor) = 2(|T|_c + \log n / \lfloor \log |T|_c \rfloor)$ bits. Portanto, Ψ consome ao todo:

$$\sum_{x \in \Sigma} 2(|T|_x + \log n / \lfloor \log |T|_x \rfloor) = 2n + \sum_{x \in \Sigma} \log n / \lfloor \log |T|_x \rfloor \tag{3.4}$$

Mas $\sum_{x \in \Sigma} \log n / \lfloor \log |T|_x \rfloor$ equivale à nH_0, portanto, Ψ consome ao todo $2n + nH_0 = n(H_0 + 2) \in O(nH_0)$ bits.
□

Uma vez que Ψ fornece um mecanismo para caminhar sobre o arranjo de sufixos, ao amostrá-lo, pode-se economizar espaço. Seja $\mathcal{K}$ o fator de amostragem do arranjo de sufixos, isto é, apenas $n/\mathcal{K}$ entradas são armazenadas explicitamente. As outras entradas que não estão amostradas podem ser recuperadas através de computação, navegando no arranjo com o uso da estrutura Ψ. Já que apenas $n/\mathcal{K}$ estão armazenadas explicitamente, o gasto de espaço representado tais entradas é de $O(n \log n/\mathcal{K})$ bits. Se $\mathcal{K} \in \Theta(\log n)$, então o espaço consumido pelo arranjo é de $O(nH_0)$ bits, o que se gasta para representar Ψ.

Notação 3.3 ($\mathcal{K}$)
O fator de amostragem do arranjo de sufixos comprimido será denotado por $\mathcal{K}$ no decorrer do documento.

Teorema 3.4 (Arranjo de sufixos em $O(nH_0)$ bits)
O arranjo de sufixos pode ser representado utilizando $O(nH_0)$ bits.

Demonstração
Uma vez que Ψ pode ser armazenado em $O(nH_0)$ bits, basta que o fator de amostragem $\mathcal{K} \in O(\log n/H_0)$.
□

A Figura 3.2 ilustra o uso da estrutura Ψ para recuperar uma entrada não amostrada. Uma vez que $\mathcal{A}[2]$ não está amostrada, consultas são efetuadas sobre Ψ quatro vezes para alcançar a primeira entrada amostrada neste percurso, $\mathcal{A}[10]$. Logo $\mathcal{A}[2] = \mathcal{A}[10] - 4 = 3$. De maneira genérica, para recuperar uma entrada não amostrada $\mathcal{A}[i]$, são efetuados k usos da estrutura Ψ até atingir uma entrada $\mathcal{A}[j]$ amostrada. Desta forma, $\mathcal{A}[i] = (\mathcal{A}[j] - k) \mod n$.

Na prática, as entradas amostradas correspondem à múltiplos do fator de amostragem $\mathcal{K}$. Apesar de não fornecerem um tempo de consulta $t_{\mathcal{A}} \in O(\mathcal{K} \cdot t_{\Psi})$ no pior caso,esse comportamento é observado no caso médio, conforme visto em [HLS+04].

i	$T_{\mathcal{A}[i]}$	$\mathcal{A}[i]$	$\Psi[i]$	Flag
0	\$	10	3	1
1	$aaacatat\$$	2	2	0
2	$aacatat\$$	3	4	0
3	$acaaacatat\$$	0	7	0
4	$acatat\$$	4	8	0
5	$at\$$	8	9	1
6	$atat\$$	6	10	0
7	$caaacatat\$$	1	1	0
8	$catat\$$	5	6	0
9	$t\$$	9	0	0
10	$tat\$$	7	5	1

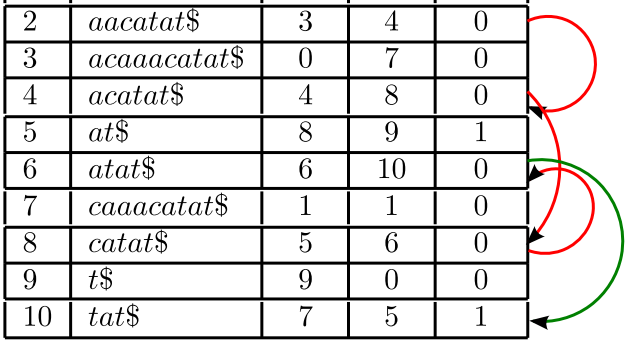


Figura 3.2: Aplicando Ψ para recuperar entradas não amostradas.

Como visto, o foco da estrutura comprimida baseia-se na representação comprimida da função Ψ e na amostragem das entradas dos arranjos de sufixos a partir desta função. No trabalho de Grossi e Vitter [GV05], o arranjo de sufixos comprimidos era construído a partir da compressão do arranjos de sufixos não-comprimidos, logo, apesar da estrutura final ocupar $O(nH_0)$ bits, era necessário uma memória de trabalho de $O(n \log n)$ bits, o custo para armazenar a estrutura não-comprimida. Esta cota torna aplicações que manipulem textos grandes inviável.

No trabalho de Hon *et al.* [HLS+07], os autores introduzem um método para construir o arranjo de sufixos de forma que a memória de trabalho gasta seja assintoticamente igual ao espaço final da estrutura, isto é, $O(nH_0)$ bits.

3.2.1 Índice-FM

O índice-FM foi proposto por Ferragina e Manzini [FM05] e pode ser visto como um arranjo de sufixos comprimidos. Este índice é baseado na transformada Burrows-Wheeler .

A transformada Burrows-Wheeler foi primeiramente descrita em [BW94]. Ela não executa nenhum tipo de compressão sobre o texto, apenas o organiza em uma permutação específica do mesmo. Para isso, ela trabalha com o conceito de ordenação cíclica dos sufixos.

A transformada é reversível, então o texto original T por ser recuperado do texto transformado $BWT(T) = L$.

Notação 3.4
Um sufixo cíclico $W_{i\circlearrowleft}$ de uma palavra W, com $|W| = n$, corresponde aos símbolos $W[i, n-1]W[0, (i-1)]$, com $0 \leq i \leq n-1$.

Seja uma matriz conceitual M^T . Cada linha M_i^T da matriz corresponde a $T_{i\circlearrowleft}$. Por exemplo, se $T = abraca\$$, M^T é dada pela matriz da Tabela 3.2.

Tabela 3.2: M^T para $T = abraca\$$.

0	a	b	r	a	c	a	$
1	b	r	a	c	a	$	a
2	r	a	c	a	$	a	b
3	a	c	a	$	a	b	r
4	c	a	$	a	b	r	a
5	a	$	a	b	r	a	c
6	$	a	b	r	a	c	a

O primeiro passo da transformada é ordenar as linhas dessa matriz conceitual M^T, obtendo assim a matriz M'^T. Denote por F a primeira coluna de M'^T e por L a última coluna dessa mesma matriz. M'^T é ilustrada pela Tabela 3.3.

Tabela 3.3: M'^T para $T = abraca\$$.

	F						L
0	$	a	b	r	a	c	a
1	a	$	a	b	r	a	c
2	a	b	r	a	c	a	$
3	a	c	a	$	a	b	r
4	b	r	a	c	a	$	a
5	c	a	$	a	b	r	a
6	r	a	c	a	$	a	b

Por fim, a transformada dá como resultado a coluna L dessa matriz conceitual. Portanto, ao aplicar a transformada Burrows-Wheeler em $T = abraca\$$, temos como resultado $BWT(T) = L = ac\$raab$.

A partir de L é possível recuperar inteiramente o texto T. A ideia é usar a função $LF(i)$, que mapeia um sufixo cíclico no mesmo sufixo cíclico rotacionado à direita de uma posição, um mapeamento de L em F. Mais formalmente temos:

$$\begin{aligned} &LF : \mathbb{N} \to \mathbb{N} \\ &LF(i) : i \mapsto j, \quad M_i'^T = T_{k\circlearrowleft} \wedge M_j'^T = T_{(k-1) \mod n\circlearrowleft} \end{aligned} \tag{3.5}$$

Note que $T_{k\circlearrowleft}$ é simplesmente $T_{k-1\circlearrowleft}$ rotacionado à esquerda de uma posição. Logo, o símbolo na coluna L correspondente à linha de $T_{i-1\circlearrowleft}$ em M'^T é justamente o símbolo que precede no texto o símbolo que está em L da linha de M'^T, que contém $T_{i\circlearrowleft}$. Portanto, ao aplicar a função $LF(i)$ sucessivas vezes, é possível recuperar o texto. Por exemplo, $L[2] = \$$, $L[LF(2)] = a$, $L[LF(LF(2))] = c$, e seguindo o raciocínio, obtém-se o reverso de T, que com um trabalho linear pode ser transformado para T.

Além disso, o símbolo de $T_{i-1\circlearrowleft}$ que está em F é o mesmo símbolo de $T_{i\circlearrowleft}$ que está em L. Esse tipo de informação ajuda a fornecer uma descrição mais algorítmica de LF.

Pela definição de M'^T, temos que os caracteres de F tem que estar em ordem crescente. Para obter F a partir de L podemos usar o *counting sort* que leva tempo de pior caso $O(\sigma + n)$. O subproduto do *counting sort* é um arranjo $C[0, \sigma]$, tal que $C[x]$ fornece a quantidade de símbolos em F que é menor lexicograficamente que o símbolo x. A partir de C e de uma inspeção linear em seus elementos, é possível reconstruir F, mas não é necessário, o arranjo C é suficiente.

Mais formalmente, C é definido como:

$$C[k] = \begin{cases} C[k-1] + |T|_{(k-1)}, & 0 < k \leq \sigma \\ 0, & k = 0 \end{cases} \tag{3.6}$$

O Algoritmo 3 ilustra a computação C em tempo $O(\sigma + n)$.

Algoritmo 3 Contagem para obter C.

Input L, Σ
Output C
function COUNTING(L, Σ)
  for $i \leftarrow 0$ **to** σ **do**
    $C[i] \leftarrow 0$
  end for
  for $i \leftarrow 0$ **to** $n - 1$ **do**
    $C[L[i]] \leftarrow C[L[i]] + 1$
  end for
  $aux \leftarrow C[0]$
  $C[0] \leftarrow 0$
  for $i \leftarrow 1$ **to** σ **do**
    $temp \leftarrow C[i]$
    $C[i] \leftarrow C[i-1] + aux$
    $aux \leftarrow temp$
  end for
  return C
end function

Agora que é conhecido como calcular C, falta responder a questão de como recuperar o texto. Tome M'^T novamente. Observe que todos os sufixos cíclicos que começam com um mesmo símbolo mantém a mesma ordem relativa se consideramos os mesmos sufixos a partir do segundo caractere. Por exemplo, sejam os seguintes sufixos cíclicos que começam com o símbolo a, ilustrado pela Tabela 3.4. Se considerarmos os sufixos cíclicos que começam a partir da segunda posição, temos que a ordem relativa se mantém, isto é, teremos a seguinte ordem em M'^T, conforme ilustrado pela Tabela 3.5.

Tabela 3.4: Sufixos que começam com o mesmo símbolo.

F						L
a	$	a	b	r	a	c
a	b	r	a	c	a	$
a	c	a	$	a	b	r

Tabela 3.5: Desconsiderando o primeiro símbolo, a ordem dos sufixos se mantém.

F						L
$	a	b	r	a	c	a
b	r	a	c	a	$	a
c	a	$	a	b	r	a

A segunda matriz pode ser encarada como uma rotação à direita da primeira matriz. Portanto, ao mapear a linha i da k-ésima ocorrência do símbolo α em L na linha j da k-ésima ocorrência do símbolo c em F estarão sendo referenciados um sufixo cíclico e o mesmo sufixo cíclico rotacionado à direita de uma posição. Logo, o símbolo que está em L no sufixo rotacionado precede o símbolo (em relação ao texto) que estava em L no primeiro sufixo.

Com essas observações, redefine-se $LF(i)$ como:

$$\begin{gathered} LF : \mathbb{N} \to \mathbb{N} \\ LF(i) : C[L[i]] + \textsc{Occ}(L, L[i], i) - 1 \end{gathered} \tag{3.7}$$

Onde $\textsc{Occ}(L, L[i], i)$ fornece o número de ocorrências do símbolo $L[i]$ no prefixo $L[0, i]$. A função LF corresponde ao mapeamento da coluna L na coluna F da matriz conceitual, ou seja, é mapeado uma linha de L a uma linha de F correspondente. Fica claro também que a matriz usada na transformada não é necessária, de acordo com a Definição 3.7, ela é apenas conceitual, conforme dito anteriormente.

Para ilustrar o comportamento da função LF, tome novamente $T = abraca\$$.

Como visto, $BW(T) = L = ac\$raab$. Obviamente, $L_0 = T_{n-1}$, pois pela condição imposta por Σ e \$, $T_{n-1\circlearrowleft}$ é o primeiro sufixo na matriz conceitual M'^T, conforme ilustrado pela Tabela 3.3.

Sabe-se que $L[2] = \$$. Prosseguindo, $LF(2) = C[\$] + \textsc{Occ}(L, \$, 2) - 1 = 0$, com $L[0] = a$. Continuando o processo, $C[a] + \textsc{Occ}(L, a, 0) - 1 = 1$, com $L[1] = c$. Desta forma, é possível obter T^R, o reverso do texto original, recuperável com um trabalho linear.

Relação com arranjos de sufixos

Como visto, ordenar os sufixos cíclicos é um dos passos da transformada, mas note que com a presença do símbolo \$, ordenar os sufixos cíclicos equivale a ordenar sufixos.

Desta forma, é possível usar métodos de construção de arranjos de sufixos sobre T para determinar L em tempo $O(n)$.

Note que, L pode ser determinado inteiramente de A, o arranjo de sufixos sobre T, pela seguinte relação:

$$L[i] = T[(\mathcal{A}[i] - 1) \mod n] \tag{3.8}$$

Além disso, é possível notar que LF pode ser encarada como uma inversa de Ψ, isto é:

$$LF(\Psi(i)) = i \tag{3.9}$$

Ou seja, a função LF pode ser utilizada como um mecanismo para caminhar no arranjo de sufixos. O mesmo procedimento utilizado nos arranjos de sufixos comprimidos pode ser utilizado: algumas entradas de A são amostradas e as outras são recuperadas através da aplicação da função LF.

Construção do índice-FM

A construção do índice-FM se baseia em quatro etapas:

1) **Transformada Burrows-Wheeler:** Dado o texto de entrada T, aplica-se a transformada Burrows-Wheeler para gerar a última coluna L da matriz conceitual M, ou seja, $L = BWT(T)$.

   A transformada Burrows-Wheeler tende a agrupar caracteres idênticos, conforme visto em [BW94], o que possibilita usar técnicas de eliminação de redundância para alcançar uma alta compressibilidade.

2) **Transformada *move-to-front*:** Após a aplicação da transformada de Burrows-Wheeler, aplicamos a transformada *move-to-front* sobre L, gerando assim a palavra L^{mtf}. Como a transformada de Burrows-Wheeler tende a agrupar símbolos iguais, é esperado que posições consecutivas de L^{mtf} constituídas de zeros ocorram.

3) ***Run-length* encoding:** As posições contíguas de zeros em L^{mtf} são codificas pelo *run-length* encoding, dando origem à palavra $L^{rle} = RLE(L^{mtf})$. Cada ocorrência de 0^m em L^{mtf} é substituída por $m + 1$ em binário, com o dígito menos significativo mais à esquerda e sem o dígito mais significativo. Para codificar em binário, dois símbolos novos são utilizados, $\mathbf{0}$ e $\mathbf{1}$.

   Tome como exemplo a palavra $L^{mtf} = 0^{10}$. Após o *run-length* encoding, $L^{rle} = \mathbf{110}$. Já $L^{mtf} = 0^5$ seria codificado como $L^{rle} = \mathbf{01}$.

   Após o *run-length* encoding, a palavra L^{rle} está bem definida sobre o alfabeto $\Sigma^{rle} = \{\mathbf{0}, \mathbf{1}, 1, 2, 3, \ldots, \sigma - 1\}$.

4) ***Variable length* encoding:** O último passo consiste em atribuir um código de comprimento variável a cada símbolo de L^{rle}. Para o símbolo $\mathbf{0}$ atribui-se o código 10, para o símbolo $\mathbf{1}$, o código 11. Para $i = 1, 2, \ldots, \sigma - 1$, o símbolo i é codificado utilizando $1 + 2\lfloor \log(i + 1) \rfloor$ bits., $\lfloor \log(i+1) \rfloor$ 0, sucedidos pela representação binária de $i+1$, que gasta $1 + \lfloor \log(i+1) \rfloor$ bits.

Nota 3.1
Denomina-se $\mathcal{Z}$ o índice resultante após aplicar todos os procedimentos acima sobre o texto T.

Teorema 3.5 ([FM05])
Para qualquer $k \geq 0$, limita-se a compressão em termos da entropia empírica do texto de entrada T em:

$$|\mathcal{Z}| \leq 5nH_k(T) + O(\log n) \tag{3.10}$$

O teorema 3.5 tem grande importância, pois indica que o índice-FM consegue utilizar um espaço igual ao espaço da entropia de k-ésima ordem, em termos assintóticos, mais um fator logarítmico.

O índice-FM pode ser construído em tempo $O(n)$ dado L, visto que todas as técnicas usadas exigem apenas uma inspeção linear do texto.

Resta agora responder como calcular $LF(i)$ em tempo $O(1)$ para qualquer i. De acordo com a equação 3.7, qualquer entrada de C pode ser recuperada em tempo constante. No entanto, pré-processamento deve ser realizado para poder responder $\textsc{Occ}(L, c, q)$ em tempo constante.

Primeiramente, L é particionada em subpalavras de tamanho $u = \Theta(\log n)$. Cada uma dessas subpalavras é denominada de balde. Cada balde BL_i representa a porção $L[iu, (i+1)u - 1]$ para $0 \leq i < \frac{n}{u}$, ou seja, a subpalavra de tamanho u correspondente.

Essa partição lógica induz uma partição de L^{mtf} em $\frac{n}{u}$ baldes. Cada balde é denominado BL_i^{mtf}, para $0 \leq i < \frac{n}{u}$. Assuma que toda porção contígua de 0 em L^{mtf} está contido em um único balde. O caso geral será explicado mais tarde.

Baseado na premissa discutida acima, cada balde BL_i^{mtf} induzirá em Z uma partição de baldes $BZ_0 \dots BZ_{\frac{n}{u}-1}$ de tamanho variável. Cada balde BZ_i é obtido de BL_i^{mtf} ao aplicar as técnicas de codificação descritas anteriormente.

Para computar $\textsc{Occ}(L, c, q)$ em $O(1)$, as seguintes informações são necessárias:

i) O número de ocorrências de c no maior prefixo $L[0, x-1]$ de $L[0, q]$ tal que x seja múltiplo de u^2.

ii) O número de ocorrências de c no prefixo $L[x, y-1]$, onde y é um múltiplo de u.

iii) O número de ocorrências de c no sufixo de L restante.

Dessa forma, bastaria somar as três informações para obter o número de ocorrências esperado. As seguinte tabelas devem ser computadas para a obtenção das três informações.

i)
- Para $0 \leq j < \frac{n}{u^2}$, a entrada $NO[c][j]$ armazena o número de ocorrências de c em $L[0, ju^2 - 1]$. Note que esse arranjo tem tamanho $O(\sigma \frac{n}{\log^2 n} \log n) = O(\sigma \frac{n}{\log n})$ bits.
- Para $0 \leq j < \frac{n}{u^2}$ a entrada $W[j]$ fornece o valor $\sum_{h=0}^{ju} |BZ_h|$, ou seja, o espaço em bits dos baldes comprimidos $BZ_0 \dots BZ_{ju}$. Observe que $BZ_0 \dots BZ_{ju}$ correspondem aos baldes comprimidos de $L[0, ju^2 - 1]$. Esse arranjo ocupa $O(\frac{n}{\log n})$ bits.

ii)
- Para $0 \leq j < \frac{n}{u}$, a entrada $NO'[c][j]$ possui o número de ocorrências de c em $L[j'u^2, ju-1]$, onde $j' = \lceil \frac{j}{u} \rceil - 1$. Ou seja, $NO'[c][j]$ fornece o número de ocorrências de c entre a posição ju e a posição à esquerda mais próxima que é múltipla de u^2. Note que este arranjo tem tamanho $O(\sigma \frac{n}{\log n} \log(u^2)) = O(\sigma \frac{n}{\log n} \log \log n)$ bits.
- Para $0 \leq j < \frac{n}{u}$, a entrada $W'[j]$ armazena o valor $\sum_{h=j'u}^{j} |BZ_h|$, ou seja, o espaço em bits dos baldes comprimidos $BZ_{j'u} \dots BZ_j$. Observe que $BZ_{j'u} \dots BZ_j$ correspondem aos baldes comprimidos de $L[j'u^2, ju - 1]$. Esse arranjo ocupa espaço de $O(\sigma \frac{n}{\log n} \log \log n)$ bits.

iii)
- Para $0 \leq j < \frac{n}{u}$, o arranjo $MTF[j]$ fornece o estado da lista da transformada *move-to-front* no começo da codificação de BL_j. Note que esse arranjo consome $O(\sigma \log \sigma \frac{n}{\log n})$ bits, pois $MTF[j]$ ocupa $O(\sigma \log \sigma)$ bits.
- Por fim, a tabela $S[c, h, BZ_j, MTF[j]]$ fornece o número de ocorrências de c nos primeiros h caracteres de BL_j, onde $h \leq u$. Isso é possível, pois BZ_j e $MTF[j]$ determinam completamente BL_j. Note que para cada par (c, h) é necessário armazenar todas as possibilidades de balde BZ_j, ou seja, $2^{u'}$ possibilidades, onde u' é o máximo número de bits ocupados por um balde comprimido. Note que o tamanho de S é de $O(u2^{u'} \log u) = O(u2^{u'} \log \log n)$ bits. .

A soma das ocorrências de c em $i)$, $ii)$ e $iii)$ deve ser feita. Portanto, para computar $\text{OCC}(Z, c, q)$, determina-se o balde BL_i ao ter $i := \lfloor \frac{q}{u} \rfloor$. Note que $h = q - iu$ fornece a posição de q dentro do balde BL_i. O parâmetro $t = \lfloor \frac{i}{u} \rfloor$ é computado. O número de ocorrências de c no prefixo $L[0, tu^2 - 1]$ é dado em $NO[c][t]$. Consequentemente, o número de ocorrências de c na subpalavra $L[tu^2, (i-1)u - 1]$ é dada por $NO'[c][i-1]$. Finalmente, é necessário computar as ocorrências de c na subpalavra restante. Para isso, recupera-se o balde BZ_i de Z. O início do balde comprimido em Z se dá em $W[t]$ se $i - 1$ é múltiplo de u ou em $W[t] + W'[i-1]$ caso contrário. O fim do balde é dado de maneira análoga. Dado BZ_i, é possível recuperar o número de ocorrências de c na subpalavra restante $L[(i-1)u, q]$ ao olhar para a tabela $S[c, h, BZ_i, MTF[i]]$. Ao somar os três valores, o valor de $\text{RANK}(L, c, q)$ é obtido.

A partir da informação exposta, o Teorema 3.6 pode ser formulado.

Teorema 3.6 ([FM05])
$\text{RANK}(L, c, q)$ *pode ser computado em* $O(1)$ *utilizando* $|\mathcal{Z}| + O(n\frac{\log \log n}{\log n})$ *bits.*

Demonstração
Por construção, o balde BZ_i *tem no máximo* $u' = (1 + 2\lfloor \log \sigma \rfloor)u$. *É possível escolher* u *de modo que* $u = \Theta(\log n)$ *bits e que* u' *tenha* $\gamma \log n$ *bits, com* $\gamma < 1$, *desde que* Σ *seja constante. Portanto, operações aritméticas e consultas nas tabelas envolvendo operandos de* $O(\log n)$ *bits podem ser efetuadas em tempo constante. Consequentemente,* $\text{RANK}(L, c, q)$ *pode ser determinado em* $O(1)$.

Recapitulando, em $i)$,$ii)$ *e* MTF, *o espaço ocupado é de* $O(n\frac{\log \log n}{\log n})$ *bits, dado mais uma vez,* Σ *constante. A questão agora é verificar o quanto* S *gasta. Como visto, o tamanho de* S *é* $O(u2^{u'} \log \log n)$. *Como* $u' = \gamma \log n$, S *requer* $O(n^{\gamma} \log n \log \log n)$ *bits. Como* $\gamma < 1$, S *não excede* $O(n\frac{\log \log n}{\log n})$ *bits.*

□

O caso mais geral, em que uma porção de zeros contíguos em L^{mtf} esteja em mais de um balde, será explicado agora.

O *run-length encoding* é responsável por substituir cada porção maximal de 0^m em L^{mtf} pela palavra $bin(m)$, que é definida como sendo a representação em binário de $m + 1$ com o bit menos significativo vindo primeiro e com o bit mais significativo descartado. Mais formalmente, temos que se $bin(m) = b_0 b_1 \dots b_k$, com $b_i \in \{0, 1\}$, então $m = \sum_{j=0}^{k} (b_j + 1)2^j$. De acordo com isso, é possível recuperar 0^m se cada bit b_j for substituído por $(b_j + 1)2^j$ símbolos 0.

Suponha que uma sequência de $c = a + b$ zeros esteja dividida entre dois baldes. Suponha, sem perda de generalidade, que a zeros estejam no balde BL_{i-1} e que b zeros estejam no balde BL_i. O processo de *run-length* encoding dá como resultado $\gamma = bin(a + b)$. Interpretando cada bit como visto acima, deve-se designar a BZ_{i-1}, o menor prefixo de γ que seja maior ou igual a a. Os demais símbolos, são designados a BZ_i. Por exemplo, se $a = 3$ e $b = 14$, temos $\gamma = bin(17) = \mathbf{0100}$. Seriam atribuídos os símbolos **01** ao balde BZ_{i-1} (pois a decodificação do mesmo corresponderia á 0^5), e **00** ao balde BZ_i (cuja decodificação é 0^3).

A tabela S pode ser usada sem problemas para contar o número de ocorrências de qualquer símbolo em BZ_{i-1}, pois BZ_{i-1} contém uma informação que não prejudica a consulta da tabela S. No entanto, não se tem a mesma confiança em BZ_i, pois só há disponível a informação de 0^3, consequentemente, faltam 11 zeros. É possível observar também que os símbolos de γ atribuídos a BZ_i não codificam mais do que b zeros, logo, sempre vai haver um número não-negativo de zeros que não estão presentes.

Para resolver o problema, é usada uma tabela adicional, $MZ[i]$, $0 \leq i < \frac{n}{u}$, que fornece o número de zeros de BL_i^{mtf} não codificados em BZ_i. No exemplo anterior, $MZ[i] = 11$.

Como MZ ocupa $O(\frac{n}{u} \log \log u)$, não é observado um aumento em termos assintóticos no espaço usado pelas demais estruturas, incluindo Z.

3.3 Arranjos de sufixos comprimidos enriquecidos

Uma vez que é possível representar um arranjo de sufixos em espaço $O(nH_0)$ é desejável que se consiga representar a informação de LCP também em espaço $o(n \log n)$. E, de acordo com Sadakane [Sad02], é possível representar a informação de LCP de maneira compacta consumindo $2n$ bits no máximo com tempo de acesso $O(t_{\mathcal{A}})$ para qualquer entrada de $LCP\,[i]$. Desta forma, o espaço usado pelo arranjo de sufixos comprimido e pela informação de LCP comprimida continua sendo $O(nH_0)$.

De acordo com o Teorema 2.2, se $LCP\,[i] > 1$, $p = \bar{\mathcal{A}}[\mathcal{A}[i] + 1]$ e $q = \bar{\mathcal{A}}[\mathcal{A}[i + 1] + 1]$, então $LCP\,[p] \geq LCP\,[i] - 1$. Em outras palavras:

$$LCP\,[\Psi[i]] \geq LCP\,[i] - 1 \tag{3.11}$$

Lembrando que $\mathcal{A}[\Psi[0]] = 0$, o seguinte é observado:

$$\begin{aligned} LCP\,[\Psi[0]] &\leq LCP\,[\Psi[\Psi[0]]] + 1 \\ &\leq LCP\,[\Psi^3[0]] + 2 \\ &\vdots \\ &\leq LCP\,[\Psi^n[0]] + n - 1 \\ &= n - 1 \end{aligned} \tag{3.12}$$

Como $\Psi^{(n)}[0] = 0$ e $LCP\,[0] = 0$, visto que $T[n-1] = \$$, a Expressão 3.12 vale. Além disso, essa mesma expressão afirma que a sequência $(LCP\,[\Psi[0]]\,, LCP\,[\Psi^2[0]] + 1, \ldots,$ $LCP\,[\Psi^n[0]] + n - 1)$ constitui uma sequência crescente, podendo então ser codificada consumindo $2n$ bits, de acordo com o Teorema 3.2. E como visto anteriormente, cada elemento dessa sequência pode ser recuperado em tempo $O(1)$.

Tabela 3.6: Se examinados em ordem decrescente de tamanho de sufixo, $\mathcal{A}[i]+LCP\,[i]$ formam uma sequência crescente.

i	$T_{\mathcal{A}[i]}$	$\mathcal{A}[i]$	$LCP[i]$	$\mathcal{A}[i] + LCP[i]$
0	\$	10	0	10
1	$aaacatat\$$	2	2	4
2	$aacatat\$$	3	1	4
3	$acaaacatat\$$	0	3	3
4	$acatat\$$	4	1	5
5	$at\$$	8	2	10
6	$atat\$$	6	0	6
7	$caaacatat\$$	1	2	3
8	$catat\$$	5	0	5
9	$t\$$	9	1	10
10	$tat\$$	7	0	7

Se o desejado for obter $LCP\,[i]$, então deve-se computar k, tal que $i = \Psi^{k+1}[0]$. Pela definição de Ψ:

$$\begin{aligned} &\mathcal{A}[\Psi^k[i]] = \mathcal{A}[i] + k \\ &\mathcal{A}[i] = \mathcal{A}[\Psi^k[\Psi[0]] = \mathcal{A}[\Psi[0]] + k = k \end{aligned} \tag{3.13}$$

Então, $k = \mathcal{A}[i]$.

Em resumo, $LCP\,[i]+\mathcal{A}[i]$ possui uma sequência crescente se as entradas forem examinadas em ordem decrescente de tamanho de sufixo. Tomando o exemplo da Tabela 3.6, a sequência crescente formada é $(3, 3, 4, 4, 5, 5, 6, 7, 10, 10, 10)$, que pode ser codificada eficientemente com no máximo $2n$ bits. Para encontrar o valor de $LCP\,[i]$ para algum i é necessário antes consultar $\mathcal{A}[i]$. Por exemplo, para consultar $LCP\,[5]$ é necessário consultar $\mathcal{A}[5] = 8$ e examinar a quinta entrada da sequência crescente $(3, 3, 4, 4, 5, 5, 6, 7, 10, 10, 10)$, que é 10, e diminuí-lo do próprio valor de $\mathcal{A}[5]$, dando como resultado $LCP\,[5] = 2$. Logo o tempo de consulta é dominado pelo tempo de acesso ao arranjo de sufixos comprimido, $O(t_{\mathcal{A}})$.

Teorema 3.7
A informação de LCP pode ser comprimida gastando apenas $2n$ bits para a sua representação, visto que $\mathcal{A}[i] + LCP\,[i]$ formam uma sequência crescente, desde que inspecionados em ordem decrescente de tamanho de sufixo.

Para recuperar alguma entrada $LCP\,[i]$ é necessário tempo $O(t_{\mathcal{A}})$.

Demonstração
A cota de espaço é consequência direta do Teorema 3.2 e da Desigualdade 3.12.

A cota de tempo é consequência direta da Equação 3.13 e do tempo de acesso ao arranjo de sufixos comprimidos.
□

Ao representar a informação de LCP de maneira comprimida, a topologia da árvore de sufixos associada é codificada de maneira comprimida também. Portanto, com este resultado, obtém-se as árvores de sufixos comprimidas, uma estrutura baseada em arranjos de sufixos comprimidos e informação de LCP comprimida.

3.4 Árvores de sufixos comprimidas

Como visto na Seção 2.3.5, os ℓ-intervalos tem relação de um para um com os nós da árvore de sufixos associadas, e ao identificar os ℓ-índices com o auxílio de algum mecanismo, é possível descobrir esses ℓ-intervalos e emular a árvore de sufixos associada ao arranjo de sufixos sem que seja necessária a representação explícita da árvore de sufixos, pois a informação de LCP codifica implicitamente esta topologia.

Uma árvore de sufixos comprimida é uma estrutura de dados que se baseia em arranjos de sufixos comprimidos e na informação de LCP comprimida e que é munida de mecanismos para identificação de ℓ-índices para navegar na topologia da árvore de sufixos associada.

No entanto, ainda não foi abordado neste documento até o momento algum mecanismo eficiente de identificação de ℓ-índices. A partir da presente seção tais mecanismos serão tratados.

Um dos mecanismos é a consulta de mínimo RMQ (*range-minimum-query*). Este mecanismo é definido formalmente como:

Definição 3.5 (RMQ)
Uma consulta de mínimo RMQ sobre uma dada tabela V, dada por $RMQ_V(i,j)$, corresponde à posição mais esquerda das posições nas quais ocorrem os valores de mínimo nesta tabela sobre um dado intervalo $[i,j]$, isto é:

$$RMQ_V(i,j) = \min\{\arg\min_{i \le k \le j}\{V[k]\}\}$$

Outros mecanismos importantes para realizar percursos mais complexos na árvore de sufixos comprimida são as informações de NSV (*next smaller value*) e PSV (*previous smaller value*), como exposto em [FMN08]. Formalmente elas são definidas como:

Definição 3.6 (NSV)
Uma consulta de $NSV(i)$ sobre uma estrutura $V[0, n-1]$, denotada por $NSV_V(i)$, corresponde ao menor $j > i$ tal que $V[j] < V[i]$.

$$NSV(i) = \begin{cases} \min\limits_{i<j\le n-1}\{V[j] < V[i]\}, & \textit{se definido} \\ n-1, & \textit{caso contrário} \end{cases} \tag{3.14}$$

Definição 3.7 (PSV)
Uma consulta de $PSV(i)$ sobre uma estrutura $V[0, n-1]$, denotada por $PSV_V(i)$, corresponde ao maior $j < i$ tal que $V[j] < V[i]$.

$$PSV(i) = \begin{cases} \max\limits_{0\le j<i}\{V[j] < V[i]\}, & \textit{se definido} \\ -1, & \textit{caso contrário} \end{cases} \tag{3.15}$$

Notação 3.5
Como será usado frequentemente no texto, $RMQ_{LCP}(i,j)$ é denotado simplesmente como $RMQ(i,j)$ quando o contexto estiver claro. O mesmo vale para $NSV(i)$ e $PSV(i)$, que se referem respectivamente à $NSV_{LCP}(i)$ e $PSV_{LCP}(i)$.

Para ilustrar o poder das consultas de RMQ, uma consulta de mínimo $RMQ(i, j-1)$ sobre um intervalo ℓ-$[i,j]$ fornece justamente a posição do primeiro ℓ-índice. Consequentemente, ao

Tabela 3.7: Operações suportadas.

Operação	Descrição
ROOT	Retorna a raiz da árvore de sufixos.
LEAF(v)	Retorna verdadeiro se v é uma folha, falso caso contrário.
LOCATE(v)	Retorna o valor de $\mathcal{A}[i]$ se v for uma folha identificada por $[i, i]$.
PARENT(v)	Retorna o pai do nó v.
CHILDREN(v)	Retorna todos os filhos de v.
CHILD(v, c)	Retorna o filho de v conectado pela aresta que inicia com c.
DEPTH(v)	Retorna o tamanho da palavra formada ao percorrer da raiz ao nó u.
EDGE(u, v)	Retorna a aresta que conecta u a v.
LCA(u, v)	Retorna o ancestral comum mais baixo de u e v.
SLINK(v)	Retorna o nó conectado a v por um *link* de sufixo.

usar a consulta diversas vezes, obtém-se todos os intervalos filhos. Por exemplo, tomando a Figura 2.7 como referência, observa-se o seguinte:

- $RMQ(0, 7) = 0$
- $RMQ(1, 7) = 3$
- $RMQ(4, 7) = 5$
- $RMQ(6, 7) = 6$

Consequentemente, os intervalos filhos são $[0, 0]$, $[1, 3]$, $[4, 5]$, $[6, 6]$ e $[7, 7]$.

Logo, ao utilizar consultas de RMQ é possível emular um percurso *top-down* sem a necessidade de representar explicitamente uma árvore de sufixos, a qual pode ser substituída completamente pelo arranjo de sufixos e informação de LCP.

Já consultas de PSV e NSV são importantes para determinar as extremidades de um dado ℓ-intervalo.

A Tabela 3.7 lista operações suportadas pela árvore de sufixos comprimida quando acompanhada das informações de RMQ, NSV e PSV.

Cada operação será descrita com detalhes a seguir.

ROOT : Para retornar o ℓ-intervalo que corresponde ao nó raiz da árvore de sufixos associada, basta retornar o intervalo $[0, n - 1]$.

LEAF(v) : Como folhas são representados por intervalos $[i, i]$, com $0 \leq i \leq n - 1$ basta verificar os limites do intervalo. Caso os limites sejam iguais, então v é uma folha e o valor retornado é verdadeiro, caso contrário, retorna-se falso.

LOCATE(v) : Se o nó v for uma folha identificada pelo intervalo $[i, i]$, o valor retornado é o valor $\mathcal{A}[i]$, isto é, a posição de início do i-ésimo sufixo na ordem lexicográfica.

PARENT(v) : O nó v é identificado por algum intervalo $[i, j]$. Deve-se encontrar um intervalo $[i', j']$ que contenha $[i, j]$ de modo que não exista um intervalo $[i'', j'']$ que esteja contido em $[i', j']$ e que contenha $[i, j]$. Para isto, basta usar as consultas de NSV e PSV de forma que $i' = PSV(k) + 1$ e $j' = NSV(k)$, onde $k = \arg\max\{LCP[i - 1], LCP[j]\}$. O valor de k tem que ser o maior possível pois ele representa um ℓ-índice, desta forma fica

garantido que não existe outro intervalo que contenha $[i, j]$ e que esteja contido em $[i', j']$. A operação é ilustrada pela Figura 3.3.

$$k = \arg\max\{LCP(i-1), LCP(j)\}$$

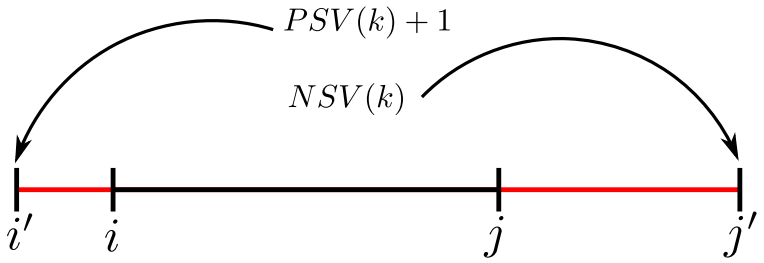


Figura 3.3: Operação de retornar o nó pai.

CHILDREN(v) : Dado que v não é uma folha, ele é definido como um ℓ-intervalo $[i, j]$. Todos os filhos de v são determinados pelos ℓ-índices, de acordo com o Teorema 2.3. Por sua vez, cada ℓ-índice $\imath_1.\imath_2, \ldots, \imath_k$ de $[i, j]$ pode ser recuperado com consultas de RMQ, isto é, $\imath_1 = RMQ(i, j-1)$, $\imath_2 = RMQ(\imath_1 + 1, j-1)$,..., $\imath_k = [\imath_{k-1} + 1, j]$. Existem no máximo σ filhos já que cada aresta possui um símbolo distinto. A figura 3.4 exemplifica a operação de retornar os filhos para um nó que possui três filhos.

$$\imath_1 = RMQ(i, j-1)$$
$$\imath_2 = RMQ(\imath_1 + 1, j-1)$$

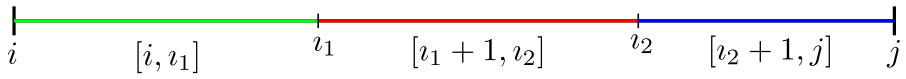


Figura 3.4: Operação de retornar os nós filhos.

CHILD(v, c) : O mesmo procedimento de CHILDREN(v) é efetuado para obter os filhos, se v não for uma folha. Uma vez identificado os intervalos-filhos do ℓ-intervalo $[i, j]$ correspondente a v, basta verificar se $T[\mathcal{A}[i'] + \ell] = c$, para cada intervalo filho $[i', j']$.

DEPTH(v) : Se v for uma folha rotulada por $\mathcal{A}[i]$, basta retornar $n - \mathcal{A}[i] - 1$. Caso contrário, uma vez que v é um ℓ-intervalo $[i, j]$, basta descobrir o valor de ℓ para responder DEPTH(v), que é obtido inspecionando $LCP[RMQ(i, j-1)]$.

EDGE(u, v) Dado que u não é uma folha, que u é pai de v e que o tamanho da palavra que rotula as arestas que saem da raiz e chegam ao nó u seja conhecida, é possível determinar EDGE(u, v). Considerando que u é denotado pelo ℓ-intervalo $[i, j]$ e que v é filho de u, e é denotado por $[i', j']$, basta recuperar a palavra $T[\mathcal{A}[i'] + \ell, T[\mathcal{A}[i'] + \ell + |S|]$. Caso o tamanho da palavra que rotula as aresta que saem da raiz e chegam ao nó u não seja conhecida, é necessário realizar uma consulta de DEPTH(u).

LCA(u, v) : Seja u representado pelo intervalo ℓ-$[i, j]$ e v pelo intervalo ℓ'-$[i', j']$.

Se u é pai de v, então retorne u. O caso em que v é pai de u é análogo. Caso contrário, o ancestral comum mais baixo de u e v é denotado por w, que é representado por um intervalo ℓ''-$[i'', j'']$.

Como u e v são ancestrais de algum nó, ambos tem que estar inclusos no intervalo $[i'', j'']$, e portanto, ℓ'', isto é, DEPTH(w), pode ser obtido ao realizar uma consulta $k = RMQ(j, i' - 1)$ e tomar $LCP\,[k]$. Os índices i'' e j'' são obtidos ao consultar $i'' = PSV(k)$ e $j'' = NSV(k)$ pela definição de ℓ''-intervalo e pelas definições de NSV e PSV. A Figura 3.5 ilustra a operação de retornar o ancestral comum mais baixo.

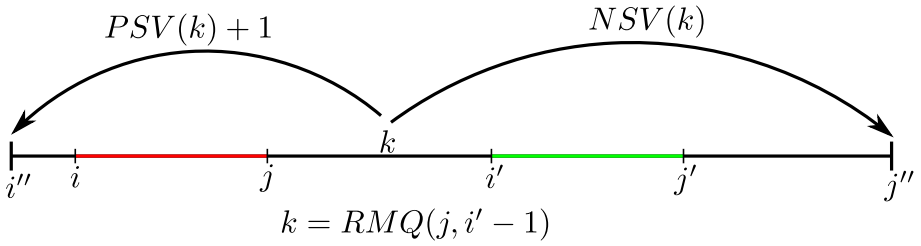


Figura 3.5: Operação de retornar o ancestral comum mais baixo.

SLINK(v) : Seja v determinado por um ℓ-intervalo $[i, j]$. Se v for uma folha, basta retorna $\Psi[i]$. Caso contrário, tome w o nó conectado a v por um *link* de sufixo. O intervalo que representa w é denotado por ℓ'-$[i'', j'']$. O valor ℓ' pode ser recuperado ao tomar $\ell' = LCP\,[k]$, onde $k = RMQ(i', j' - 1)$, com $i' = \Psi(i)$ e $j' = \Psi(j)$. As extremidades i' e j'' são recuperadas respectivamente com consultas $PSV(k)$ e $NSV(k)$. A figura 3.6 ilustra esse processo.

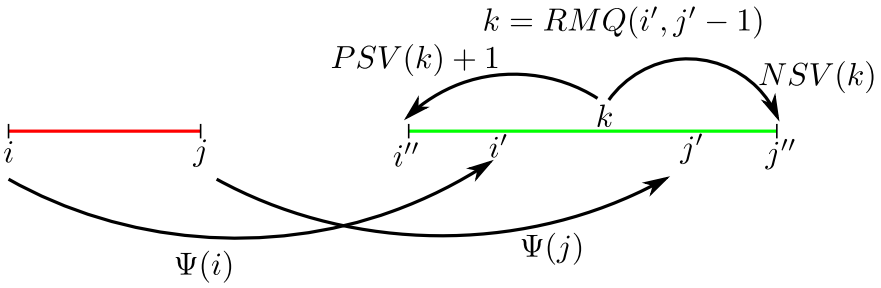


Figura 3.6: Operação de atravessar o elo de sufixo.

Portanto, mesmo comprimindo informação, é possível emular funcionalidades importantes de uma árvore de sufixos não-comprimida. Algumas funcionalidades como as consultas de LCA e *links* de sufixos só seriam possíveis em árvores não-comprimidas se espaço extra fosse utilizado, conforme mencionado em [Gus97]. No caso das árvores de consultas comprimidas, tais consultas estão inclusas nas operações básicas suportadas pela estrutura, visto que a maioria delas precisa de informação de RMQ, PSV e NSV.

Capítulo 4

Soluções propostas

Para um entendimento completo da implementação proposta, é necessário entender as características que estão por trás dela. A maneira que cada característica foi implementada será descrita neste Capítulo. Resumidamente, a implementação desenvolvida conta com os seguintes componentes:

- Implementação de consultas de $Rank$ e $Select$ de acordo com [GGMN05].
- Arranjos de sufixos comprimidos de foram construídos com o algoritmo incremental de Hon *et al.* [HLS$^+$07].
- Informação de LCP construída de acordo com as metodologias de Kasai *et al* [KLA$^+$01] e Sadakane [Sad02].
- Implementação da estrutura de dados proposta por Cánovas e Navarro [CN10] a fim de responder consultas de RMQ, PSV e NSV.

4.1 Operações em vetores de bit

Apesar das operações de $Rank$ e $Select$ poderem ser efetuadas em tempo constante, este tipo de solução não é recomendável na prática. Há soluções propostas que são mais econômicas em espaço e mais eficientes em relação ao tempo devido ao melhor uso de memória cache, apesar de assintoticamente, o tempo de consulta ser de $\omega(1)$, conforme visto no trabalho de González *et al.*[GGMN05]. A implementação foi feita com base neste resultado, uma vez que os autores demonstram empiricamente que as soluções que levam tempo $\omega(1)$ são melhores na prática.

4.1.1 Operação de $Rank$

Para a operação de $Rank$, o vetor de bits V é divido em $\mathcal{X}$ blocos de tamanho igual, cada bloco, corresponde à uma entrada do arranjo $R = [0, \lfloor \frac{n}{\mathcal{X}} \rfloor]$, cada entrada é preenchida da seguinte forma:

$$R[i] = \begin{cases} Rank_1(V, i\mathcal{X} - 1), & i > 0 \\ 0, & i = 0 \end{cases} \tag{4.1}$$

Assim cada entrada informa a quantidade 1s no prefixo associado.

Para recuperar $Rank_1(V, i)$, é possível usar R para recuperar quantidade de 1s através da entrada $R[\mathcal{X}\lfloor\frac{i}{\mathcal{X}}\rfloor]$. Uma vez determinada a quantidade de 1s no prefixo de V, uma pesquisa sequencial contando cada bit 1 é feita no intervalo $[\mathcal{X}\lfloor\frac{i}{\mathcal{X}}\rfloor, i]$ através de uma técnica conhecida como *popcount*. Esta técnica divide cada palavra de memória do intervalo dado, em palavras menores. Através da indexação destas palavras menores em um arranjo pré-computado de tamanho constante que mapeia um inteiro na quantidade de 1s que ele possui em binário, é possível descobrir quantos 1s a palavra de memória contém. Desta forma, é possível responder a consulta de $Rank_1(V, i)$. Uma pesquisa individual de bits é feita nos bits restantes, caso o número de bits restante não seja divisível pelo tamanho da subpalavra.

Para construir R leva-se tempo $O(\frac{n}{\mathcal{X}})$ e R gasta $O(\frac{n}{\mathcal{X}}\log n)$ bits. Se $\mathcal{X} = \log^{\epsilon}(n)$, $\epsilon \geq 1$, gasta-se espaço $O(n)$ bits e tempo $O(\frac{n}{\log^{\epsilon} n})$ para construção de R.

Para responder $Rank_1(V, i)$, gasta-se tempo $O(\mathcal{X})$. O termo $\mathcal{X}$ da notação, deve-se ao fato que uma inspeção linear utilizando *popcount* deve ser feita em um bloco de tamanho $\mathcal{X}$, no pior caso. Além disso, como o número de subpalavras de memória é constante, a técnica de *popcount* pode ser feita em tempo constante para cada palavra de memória.

Na prática, se o tamanho da palavra de memória é de 4 bytes, pode-se usar dois *popcounts* para descobrir o número de 1s da palavra, basta indexar os 16 primeiros bits no arranjo pré-computado, e os últimos 16 bits nesse mesmo arranjo. Note que o arranjo utilizado pela técnica de *popcount* neste caso tem que possuir 2^{16} entradas.

A consulta de $Rank_0$ pode ser efetuada a partir de $Rank_1$, basta notar que:

$$Rank_0(V, i) = i - Rank_1(V, i) + 1$$

4.1.2 Operação de $Select$

A operação de $Select_1(V, i)$ pode ser feita realizando uma busca binária sobre R. A busca binária é feita para achar o maior índice k de R tal que $R[k] < i$. Uma vez que a busca binária é feita, basta executar operações de *popcount* e de inspeção individual de bits para determinar $Select_1(V, i)$.

A operação de $Select_1(V, i)$ requer tempo $O(\log\frac{n}{\mathcal{X}} + \mathcal{X})$. O termo $\mathcal{X}$ é da inspeção individual de bits e das operações de *popcount*. Já o termo $\frac{n}{\mathcal{X}}$ representa a busca binária sobre R.

$Select_0(V, i)$ é respondido de maneira análoga, pois a partir de $R[i]$, é possível inferir o número de zeros, como visto anteriormente.

Note que em teoria, t_Ψ, o tempo para um acesso à Ψ, leva tempo $O(1)$, mas na prática, $t_\Psi \in O(\log\frac{n}{\mathcal{X}} + \mathcal{X})$ se o esquema proposto for usado. No entanto, no decorrer do documento, consideraremos que as operações de $Rank$ e $Select$ levam tempo constante, como descrito em [Cla97], por motivo de simplicidade.

4.2 Arranjos de sufixos comprimidos

Uma vez que a construção de arranjos de sufixos comprimidos baseia-se na compressão de um arranjo de sufixos não-comprimido, o total de espaço necessário para a construção da estrutura é de $O(n \log n)$ bits. Este espaço pode ser excessivo na prática para entradas grandes.

O método de Hon *et al.* [HLS+07], utilizado na implementação proposta, é capaz de construir o arranjo de sufixos comprimidos utilizando não mais do que $O(nH_0)$ bits de memória de trabalho, isto é, assintoticamente o mesmo para representar a estrutura final. Então, na prática é necessário um pouco mais do que a memória final para representar a estrutura de dados comprimida durante a construção desta.

Este método baseia-se em computar Ψ incrementalmente. Para isto, o texto é dividido em $\mathcal{B}' = \frac{n}{\mathcal{B}}$ blocos de tamanho $\mathcal{B}$, isto é, os blocos são $T[0, \mathcal{B}-1], T[\mathcal{B}, 2\mathcal{B}-1] \ldots T[n-\mathcal{B}-1, n-1]$. O caso base do método é calcular a porção que corresponde aos últimos sufixos do texto, ou seja, calcular Ψ para $T[n-\mathcal{B}-1, n-1]$.

O passo incremental consiste em, dado que Ψ foi calculado para $T' = T[i\frac{n}{\mathcal{B}}, n-1]$, o cálculo de Ψ para o texto $U = T''T' = T[(i-1)\frac{n}{\mathcal{B}}, n-1]$ é efetuado, onde $T'' = T[(i-1)\mathcal{B}, i\mathcal{B}-1]$. Com esta metodologia incremental, é possível construir o arranjo de sufixos comprimido em tempo $O(n \log n)$, se $\mathcal{B}' = \Theta(\log n)$.

Notação 4.1 (Notação do método incremental)
O algoritmo é incremental, então, antes da iteração, os valores se baseiam na configuração de T'. Após a iteração, os valores se baseiam em $U = T''T'$. Portanto, os valores de Ψ antes e depois da iteração são denotados por $\Psi_{T'}$ e Ψ_U. Outros valores são denotados de maneira análoga.

Notação 4.2 ($\mathcal{B}$ e $\mathcal{B}'$)
O símbolo $\mathcal{B}$ se referirá ao tamanho dos blocos durante o método incremental. Já $\mathcal{B}'$ corresponderá à quantidade de blocos, sempre que referenciado.

Para o caso base, o arranjo de sufixos $\mathcal{A}_U[0, \mathcal{B}-1]$ é calculado para $T[n-\mathcal{B}-1, n-1]$ usando um algoritmo comum para construção de arranjos de sufixos. A_U consome apenas $O(\mathcal{B} \log n)$ bits e pode ser calculado em tempo $O(\mathcal{B})$. Como $\mathcal{B} \in \Theta(n/\log n)$, A_U pode ser calculado em tempo $o(n)$ e usando apenas $O(\mathcal{B} \log n) \in O(n)$ bits. Uma vez que o cálculo de $\mathcal{A}_U$ tiver sido realizado, $\bar{\mathcal{A}}_U$ pode ser obtido em tempo $o(n)$ a partir de $\mathcal{A}_U$ usando também $O(n)$ bits. O cálculo de Ψ é efetuado a partir de $\mathcal{A}_U$ e $\bar{\mathcal{A}}_U$, visto que $\Psi(i) = \bar{\mathcal{A}}_U[\mathcal{A}_U[i]+1]$ e, como abordado anteriormente, Ψ pode ser armazenado utilizando apenas $O(\mathcal{B}H_0) \in O(nH_0)$ bits.

Os passos seguintes do algoritmo baseiam-se na divisão de sufixos curtos $\mathcal{SS}$ e sufixos longos $\mathcal{LS}$. Os sufixos curtos correspondem à porção do texto em relação à qual foi calculada a função Ψ na iteração anterior. Já os sufixos longos correspondem à porção do texto adicionada, a qual a nova função Ψ deverá ser atualizada em relação.

Durante as iterações, $\mathcal{SS} = \{T_{i\mathcal{B}}, T_{i\mathcal{B}+1}, \ldots, T_{n-1}\}$, enquanto $\mathcal{LS} = \{T_{(i-1)\mathcal{B}}, T_{(i-1)\mathcal{B}+1}, \ldots, T_{i\mathcal{B}-1}\}$. Desta forma, $|\mathcal{LS}| = \mathcal{B}$.

Para calcular Ψ_U, a estrutura de Ψ correspondente à U, é necessário computar para cada $S \in \mathcal{LS}$:

1) A posição lexicográfica de S em relação aos sufixos de $\mathcal{LS}/\{S\}$, denotado por $Pos(S, \mathcal{LS})$.

2) A posição lexicográfica de S em relação aos sufixos de $\mathcal{SS}$, denotado por $Pos(S, \mathcal{SS})$.

Notação 4.3
De maneira mais genérica $Pos(S, \mathcal{S})$ fornece a posição de um sufixo S em relação aos demais sufixos de $\mathcal{S}/\{S\}$.

Para calcular $Pos(S, \mathcal{LS})$, todos os sufixos poderiam ser ordenados e as posições desejadas extraídas através do arranjo inverso, mas isso gastaria tempo $O(n)$ e espaço $O(n \log n)$. No entanto, é possível ordenar os sufixos de $\mathcal{LS}$ entre si considerando apenas os $2\mathcal{B}$ primeiros caracteres de cada um em tempo $O(\mathcal{B} \log \mathcal{B})$ e isso pode ser alcançado aplicando o algoritmo de Manber e Myers [MM90] ou Larsson e Sadakane [LS07] (cujo código foi usado na implementação proposta), baseados na técnica de *prefix-doubling*, através de $\log \mathcal{B}$ iterações. O arranjo de sufixos inverso produzido por esta ordenação é denotado por P.

Como apenas foram considerados os primeiros $2\mathcal{B}$ símbolos dos sufixos longos, podem haver entradas repetidas em P, isto é, empates. O desempate é realizado considerando a informação calculada previamente que está armazenada em $\Psi_{T'}$, que possui a ordem dos sufixos curtos entre si considerando o tamanho integral. Note que $\Psi_{T'}(0) = \bar{\mathcal{A}}_{T'}[0]$, $\Psi_{T'}(\Psi_{T'}(0)) = \bar{\mathcal{A}}_{T'}[1]$, logo, $\texttt{Pos}(T'_k, \mathcal{SS}) = \Psi_{T'}^{(k+1)}[0], 0 \leq k \leq \mathcal{B}-1$. Um arranjo Q guarda as posições relativas dos sufixos $T'_0, \ldots, T'_{\mathcal{B}-1}$ no conjunto $\mathcal{SS}$, obtidas através da iteração de $\Psi_{T'}[0], \ldots, \Psi_{T'}^{(\mathcal{B})}[0]$ em tempo $O(\mathcal{B})$ e espaço $O(n)$ bits.

Desta forma, para obter $Pos(S, \mathcal{LS})$, onde $S \in \mathcal{LS}$, basta aplicar radix-sort nas tuplas $(P[j], Q[j])$ para $0 \leq j \leq \mathcal{B} - 1$. Sempre que duas entradas de $P[i]$ são iguais, o desempate é realizado pelo valor $Q[i]$ associado. O *radix-sort* leva tempo $O(\mathcal{B} \log \mathcal{B}) \in O(n)$, visto que a tupla consiste de um par.

É importante ressaltar que P e Q são arranjos intermediários e que gastam $O(\mathcal{B} \log n) = O(n)$ bits. O valor de $Pos(S, \mathcal{LS})$, isto é, o resultado do *radix-sort*, é armazenado um arranjo M, de $O(\mathcal{B} \log n) \in O(n)$ bits. O inverso de M, denotado por $\bar{M}$ também é processado e armazenado em tempo $O(\mathcal{B})$ e espaço $O(n)$ bits, pois será necessário em uma fase posterior do algoritmo.

Ao todo, é necessário espaço para armazenar P, Q, M e $\bar{M}$, portanto uma memória de trabalho $4\mathcal{B} \log n$ bits é requisitada. Após computados, P e Q podem ser liberados da memória.

Para calcular $Pos(S, \mathcal{SS})$ para todo $S \in \mathcal{LS}$, uma abordagem de programação é adotada.

Relembrando a Equação 3.6, $C_{T'}[0, \sigma]$ é um arranjo que informa a quantidade de símbolos lexicograficamente menores que o desejado no texto T'.

$$C_{T'}[k] = \begin{cases} C_{T'}[k-1] + |T'|_{(k-1)}, & 0 < k \leq \sigma \\ k = 0 \end{cases}$$

No final de cada iteração do algoritmo incremental, $C_{T'}$ é atualizado com os símbolos de T'', isto é, para C_U. Esta atualização leva tempo $O(\mathcal{B})$. C por sua vez gasta apenas $O(\sigma \log n)$ bits. Note que não é preciso armazenar explicitamente $|T'|_x$ visto que pode ser calculado em tempo constante ao executar a operação $C[x+1] - C[x]$.

Seja $T''[k] = x$. O resultado de $Pos(T''_k, \mathcal{SS})$ será armazenado no arranjo $L[0, k]$, $0 \leq k \leq \mathcal{B} - 1$ baseando na seguinte recorrência:

$$L[k] = \begin{cases} \max\{C_{T'}[x] \leq r < C_{T'}[x+1] | \Psi_{T'}(r) \leq L[k+1]\} + 1, & 0 \leq k < \mathcal{B} - 1 \\ \max\{C_{T'}[x] \leq r < C_{T'}[x+1] | \Psi_{T'}(r) \leq \Psi(0)\} + 1, & k = \mathcal{B} - 1 \\ C_{T'}[x], & \text{se } r \text{ não está definido.} \end{cases} \tag{4.2}$$

O ponto chave da Expressão 4.2 é que T''_k pode ser expresso como xT''_{k+1}. Logo é possível reaproveitar o valor já computado de $L[k+1]$ para o cálculo de $L[k]$. Como $Pos(T''_k, \mathcal{SS})$

representa o número de sufixos de $\mathcal{LS}$ lexicograficamente menores que T''_k, após a retirada do primeiro símbolo de T''_k, o problema se reduz a encontrar $Pos(T''_{k+1}, \mathcal{SS}) = L[k+1]$.

Assim, ao maximizar r em $\Psi_{T'}(r) \leq L[k+1]$, o número de sufixos de $\mathcal{SS}$ que é lexicograficamente menor que T_k se torna conhecido, bastando então somar r de uma unidade para que $Pos(T''_k, \mathcal{SS}) = L[k]$ ocupe a posição lexicográfica correta em relação aos sufixos de $\mathcal{SS}$.

O caso base, quando $k = \mathcal{B} - 1$, segue o mesmo procedimento. No entanto, deve-se maximizar r em $\Psi_{T'}(r)$ com relação a $\Psi_{T'}(0)$, já que $\Psi_{T'}(0) = Pos(T'_0, \mathcal{SS})$, o primeiro sufixo curto de T'.

Quando não existe r em tais condições, o algoritmo deve informar justamente $C[x]$, que por definição, fornece a posição de onde o primeiro sufixo começando com x se enquadraria no arranjo de sufixos.

Como Ψ fornece uma sequência crescente no intervalo dado, para maximizar r, basta efetuar uma busca binária, logo, cada entrada $L[k]$ pode ser computada em tempo $O(\log n)$. Já que o número de entradas é igual a $\mathcal{B}$, o tempo gasto é $O(\mathcal{B} \log n) = O(n)$. O arranjo L gasta apenas $O(\mathcal{B} \log n) \in O(n)$ bits em sua representação.

Ψ_U deve ser construído para finalizar a iteração do algoritmo incremental. Até então, as informações obtidas são:

1) $M[k]$ armazena $Pos(T''_k, \mathcal{LS})$, isto é, a posição lexicográfica relativa dos sufixos longos entre si.

2) $\bar{M}[k]$, o inverso de M, contém a posição l de início do k-ésimo sufixo longo na ordem lexicográfica entre sufixos longos, isto é, $\bar{M}[k] = l$.

3) $L[k]$ armazena $Pos(T''_k, \mathcal{SS})$, ou seja, a posição lexicográfica dos sufixos Longos em relação somente aos sufixos curtos.

A posição de cada sufixo longo envolvendo todos os sufixos, isto é, $Pos(S, \mathcal{SS} \cup \mathcal{LS})$, $S \in LS$, pode ser obtida ao somar a posição de S entre os sufixos longos e a posição entre os sufixos curtos, isto é, $Pos(T''_k, \mathcal{SS} \cup \mathcal{LS}) = M[k] + L[k]$.

O maior problema é computar as posições de cada sufixo curto em relação a todos os sufixos. Isto é obtido ao ajustar a posição dos próprios sufixos curtos de acordo com a distribuição dos sufixos longos. Esta distribuição é representada através de um vetor de bits. Por sua vez, este vetor de bits V é definido como:

Definição 4.1 (O vetor de bits V)

$$V[k] = \begin{cases} 1, & \textit{se } Pos(S, \mathcal{SS} \cup \mathcal{LS}) = k \wedge S \in \mathcal{LS} \\ 0, & \textit{se } Pos(S, \mathcal{SS} \cup \mathcal{LS}) = k \wedge S \in \mathcal{LS} \end{cases} \tag{4.3}$$

Assim $V[k]$ tem valor 1 se o k-ésimo sufixo dentre sufixos longos e curtos é um sufixo longo, e 0 caso ele seja um sufixo curto.

Através de consultas de $Rank$ e $Select$ sobre V, é possível inferir sobre a distribuição dos sufixos longos e sufixos curtos segundo à ordem lexicográfica.

É possível construir V através de L da seguinte maneira. L é ordenado em ordem crescente dando origem a outro arranjo L'. Desta forma, L' contém as posições dos sufixos longos em ordem crescente em relação aos sufixos curtos segundo a ordem lexicográfica. Então, se $L = (r_0, r_1, \ldots, r_{\mathcal{B}-1})$, V é construído utilizando a codificação unária baseada na diferença entre

$r_k - r_{k+1}$ ou seja, V é formado por r_0 0's seguidos de um 1, $r_1 - r_0$ 0's seguidos de um 1,. . ., e finalmente $r_{\mathcal{B}-1} - r_{\mathcal{B}-2}$ 0s seguidos de um 1.

É importante notar que V gasta apenas $O(n)$ bits e pode ser construído em tempo $O(\mathcal{B} \log \mathcal{B}) \in O(n)$, pois a ordenação de L para construir L' é necessária para a construção de V. Em relação ao espaço, L' ocupa apenas $O(\mathcal{B} \log n) \in O(n)$ bits.

O seguinte lema mostra como é possível inferir a distribuição dos sufixos curtos utilizando consultas sobre V.

Lema 4.1 ([HLS+07])
Pra qualquer sufixo curto S de U, seja $r = Pos(S, \mathcal{SS} \cup \mathcal{LS})$ e $r' = Pos(S, \mathcal{SS})$. Então $r = Select_0(V, r' + 1)$ e $r' = Rank_0(V, r) - 1$.

Demonstração ([HLS+07])
Por definição $V[r] = 0$. No subarranjo $V[0, \ldots, r]$, o número de 0's é igual ao número de sufixos curtos lexicograficamente menores do que S , que é igual a r'.No mais, $V[r]$ contém o r'-ésimo zero, logo, $r' = Rank_0(V, r) - 1$.

Consequentemente $Select_0(V, r' + 1)$ seleciona o r'-ésimo sufixo curto em relação à todos os sufixos (longos e curtos), pela própria definição de V.

□

Após computado V, todas as informações necessárias para a computação de Ψ_U estão disponíveis. A próxima observação é crucial para computar $\Psi_U(k)$: se $Pos(U_k, \mathcal{SS} \cup \mathcal{LS}) = r$ então $\Psi_U(r) = Pos(U_{k+1}, \mathcal{SS} \cup \mathcal{LS})$. A partir dos próximos lemas, a computação de Ψ_U se torna clara, dependendo apenas de U_k ser sufixo longo ou curto.

Lema 4.2 ([HLS+07])
Considere U_k tal que $U_k \in \mathcal{SS}$. Além disso, suponha que $Pos(U_k, \mathcal{SS} \cup \mathcal{LS}) = r$. Então, as seguintes propriedades valem:

1) $Pos(U_{k+1}, \mathcal{SS}) = \Psi_{T'}(Rank_0(V, r) - 1)$.

2) $Pos(U_{k+1}\mathcal{SS} \cup \mathcal{LS}) = Select_0(V, \Psi_{T'}(Rank_0(V, r) - 1) + 1)$.

Demonstração ([HLS+07])
A primeira propriedade vale, pois como $r = Rank_0(V, r) - 1$ pelo lema 4.1 e pela hipótese que $\Psi_{T'}$ foi calculado corretamente o resultado $Pos(U_{k+1}, \mathcal{SS}) = \Psi_{T'}(Rank_0(V, r) - 1)$ é obtido.

A segunda propriedade também é verdadeira. Pelo 4.1 e pela propriedade anterior, $Pos(U_{k+1}, \mathcal{SS} \cup \mathcal{LS}) = Select_0(V, \Psi_{T'}(Rank_0(V, r) - 1) + 1)$.

□

Lema 4.3 ([HLS+07])
Considere U_k tal que $U_k \in \mathcal{LS}$. Além disso, suponha que $Pos(U_k, \mathcal{SS} \cup \mathcal{LS}) = r$. Então as seguintes propriedades valem:

1) $k = \bar{M}[Rank_1(V, r) - 1]$

2) Se $k < \mathcal{B} - 1$ então $Pos(U_{k+1}, \mathcal{SS} \cup \mathcal{LS}) = M[k + 1] + L[k + 1]$, caso contrário, se $k = \mathcal{B} - 1$ então $Pos(U_{k+1}, \mathcal{SS} \cup \mathcal{LS}) = Select_0(V, \Psi_{T'}(0) + 1)$. Como U_k é um sufixo longo, $k \in [0, \mathcal{B} - 1]$.

Demonstração ([HLS$^+$07])
Como U_k é um sufixo longo, e pela definição de V, $Pos(U_k, \mathcal{LS}) = r'$. Pela definição de $\bar{M}$, $k = \bar{M}[r']$, como enunciado.

Se $k < \mathcal{B} - 1$, então, $Pos(U_{k+1}, \mathcal{SS} \cup \mathcal{LS}) = M[k+1] + L[k+1]$, pois $M[k+1]$ armazena $Pos(U_{k+1}, \mathcal{LS})$, enquanto $L[k+1]$ armazena $Pos(U_{k+1}, \mathcal{SS})$.

Se $k = \mathcal{B} - 1$, $U_{k+1} = T'_0$, o maior dos sufixos curtos. Logo, pelos lemas 4.1 e 4.2, $Pos(U_{k+1}, \mathcal{SS} \cup \mathcal{LS}) = Select_0(V, \Psi_{T'}(0) + 1)$.

□

Os Lemas 4.2 e 4.3 fornecem uma maneira de calcular Ψ_U para finalizar a iteração do algoritmo. O cálculo de Ψ_U é ilustrado pelo Algoritmo 4.

Algoritmo 4 Computação de Ψ_U.

Input $\Psi_{T'}, L, M, \bar{M}, V$
Output Ψ_U
  $\Psi_U[0] \leftarrow M[0] + L[0]$
  for $r \leftarrow 1$ **to** $m - 1$ **do**
    if $V[i] = 0$ **then**
      $r' \leftarrow Rank_0(V, r) - 1$
      $p \leftarrow \Psi_{T'}[r']$
      $\Psi_U[r] \leftarrow Select_0(V, p + 1)$
    else
      $r' \leftarrow Rank_1(V, r) - 1$
      $k \leftarrow \bar{M}[r']$
      if $k < \mathcal{B} - 1$ **then**
        $\Psi_U[r] \leftarrow M[k + 1] + L[k + 1]$
      else
        $p \leftarrow \Psi_{T'}[0]$
        $\Psi_U[r] \leftarrow Select_0(V, p + 1)$
      end if
    end if
  end for

Teorema 4.1
A construção de um arranjo comprimido baseado no método de Hon et al. *[HLS$^+$07] usa uma memória de trabalho de apenas $O(nH_0)$ bits e leva tempo $O(n \log n)$, no pior caso.*

Demonstração
Os arranjos adicionais para computar Ψ_U em cada iteração são M,$\bar{M}$,L, L' e V. Todos eles tem um custo de $O(n) \in (nH_0)$ bits. O espaço necessário para manter $\Psi_{T'}$ e Ψ_U, tem um custo de $O(nH_0)$ bits, o que justifica a cota do teorema.

Como visto, a computação dos arranjos adicionais levam apenas tempo $O(n)$, bem como a computação de Ψ_U, já que consultas de $\Psi_{T'}$, $Rank$ e $Select$ levam tempo constante. Logo, o Algoritmo 4, também leva tempo $O(n)$.

Como o texto foi dividido em $\mathcal{B}' = n/\mathcal{B} \in \Theta(\log n)$ blocos, então o tempo necessário é $O(n \log n)$ para a computação do arranjo de sufixos comprimidos, visto que cada iteração leva tempo $O(n)$.

□

Uma vez que Ψ foi computado, para calcular $\mathcal{A}$, o arranjo de sufixos comprimidos, basta iterar $j = \Psi^k[0]$ para $1 \leq k \leq n$ e armazenar o valor $\mathcal{A}[j] = k$ sempre que j for múltiplo de $\mathcal{K}$. Desta forma a informação de $\mathcal{A}$ estará amostrada em todas as entradas múltiplas de $\mathcal{K}$.

4.3 Arranjos de sufixos comprimidos enriquecidos

Como visto, a construção da informação de LCP segundo a Seção 2.3.4 requer o uso de informação extra como o arranjo de sufixos inverso $\bar{\mathcal{A}}$ e portanto, requer $O(n \log n)$ bits.

No entanto, é possível adaptar o Algoritmo 2 para trabalhar apenas aplicando a função Ψ e o arranjo de sufixos comprimido A, gastando assim apenas $O(nH_0)$ bits para a computação da informação de LCP comprimida. Basta notar que o único valor necessário de $\bar{\mathcal{A}}$ é o valor $\bar{\mathcal{A}}[\mathcal{A}[i] + 1] = \Psi(i)$.

O Algoritmo 5 ilustra as modificações necessárias para adaptar o algoritmo anterior. Este algoritmo foi utilizado na implementação proposta.

Algoritmo 5 Computação de LCP de acordo com os métodos de Kasai *et al.* [KLA+01] e Sadakane [Sad02].

Input $\mathcal{A}$, Ψ, T
Output LCP

```
function COMPARA(i, j, h)
    return max{k | T_{i+h} =_k T_{j+h}}
end function

function COMPUTALCP
    h ← 0
    rank ← 0
    for i ← 0 to n − 1 do    // Computa LCP [Ā[i]]
        rank ← Ψ[rank]
        if rank < n − 1 then
            k ← A[rank + 1]
            prefix ← prefix + COMPARA(i, k, prefix)
            LCP[rank] ← prefix + A[rank]
            if prefix > 1 then
                prefix − −
            end if
        else
            LCP[rank] ← 0
        end if
    end for
end function
```

Como o método de Kasai *et al.* [KLA+01] computa os valores de LCP na ordem do texto, ele se encaixa perfeitamente na metodologia da compressão da informação de LCP vista na Seção 3.3, pois essa informação segue uma ordem crescente se as entradas $\mathcal{A}[i] + LCP[i]$ forem inspecionadas na ordem do texto.

A representação da informação de LCP requer $2n$ bits de acordo com o Teorema 3.7.

O Algoritmo 5 leva tempo $O(n \cdot t_{\mathcal{A}})$, pelo mesmo motivo da análise do Algoritmo 2 e pelo fato do tempo de acesso ao arranjo A não ser constante, isto é, levar tempo $t_{\mathcal{A}}$.

Como a informação de LCP está codificada segundo a ordem do texto, é necessário um acesso a $\mathcal{A}[i]$ para descobrir o valor de $LCP[i]$, isto é, se $\mathcal{A}[i] = k$, então o valor codificado de $\mathcal{A}[i] + LCP[i]$ está na posição k da lista ordenada. Tal valor pode ser recuperado com consulta de $Select_1$, como demonstrado pelo Algoritmo 6.

Algoritmo 6 Recuperação do valor $LCP[i]$.

Input $\mathcal{A}$,V_{LCP},i
Output $LCP[i]$
return $Select_1(V_{LCP}, \mathcal{A}[i] + 1) - \mathcal{A}[i]$

No Algoritmo 6, V_{LCP} é o vetor de bits codificado de acordo com o mesmo esquema proposto para a codificação de Ψ, notando-se que esse gasta apenas $2n$ bits.

Como o tempo do Algoritmo 6 é dominado por um acesso a $\mathcal{A}$ e uma consulta de $Select_1$, $t_{LCP} = \Theta(t_{\mathcal{A}})$

O resultado acima, juntamente com a construção usando o método incremental de Hon *et al.* [HLS+07], foi publicado no Brazilian Symposium on Bioinformatics 2012 pelos autores [SAR12].

4.4 Árvores de sufixos comprimidas

Para tratar com as consultas de RMQ, NSV e PSV apenas uma estrutura, proposta por Cánovas e Navarro [CN10], é necessária. Essa mesma estrutura foi adotada na implementação proposta e ela corresponde à uma árvore de ordem $\mathcal{L}$ completa. Desta forma, cada nó tem $\mathcal{L}$ filhos. As folhas dessa árvore correspondem à própria informação de LCP. Cada nó interno é rotulado com a posição mais à esquerda dentre as posições as quais ocorre o menor valor de LCP considerando apenas os filhos deste nó. Cada conjunto de $\mathcal{L}$ folhas representando um intervalo $[i, j]$ que dão origem à um ancestral é denominado de bloco.

Notação 4.4 ($\mathcal{L}$)
Sempre que $\mathcal{L}$ for usado no texto, estará se referindo à ordem da árvore completa proposta por Cánovas e Navarro [CN10].

Tomando os valores de LCP da Tabela 3.6, a estrutura de dados proposta tomando $\mathcal{L} = 3$ é ilustrada pela Figura 4.1. Note que assim como na figura, na prática não é necessário representar todos os nós da árvore completa, desta forma é possível obter uma economia de espaço desde que cada folha possua a mesma altura.

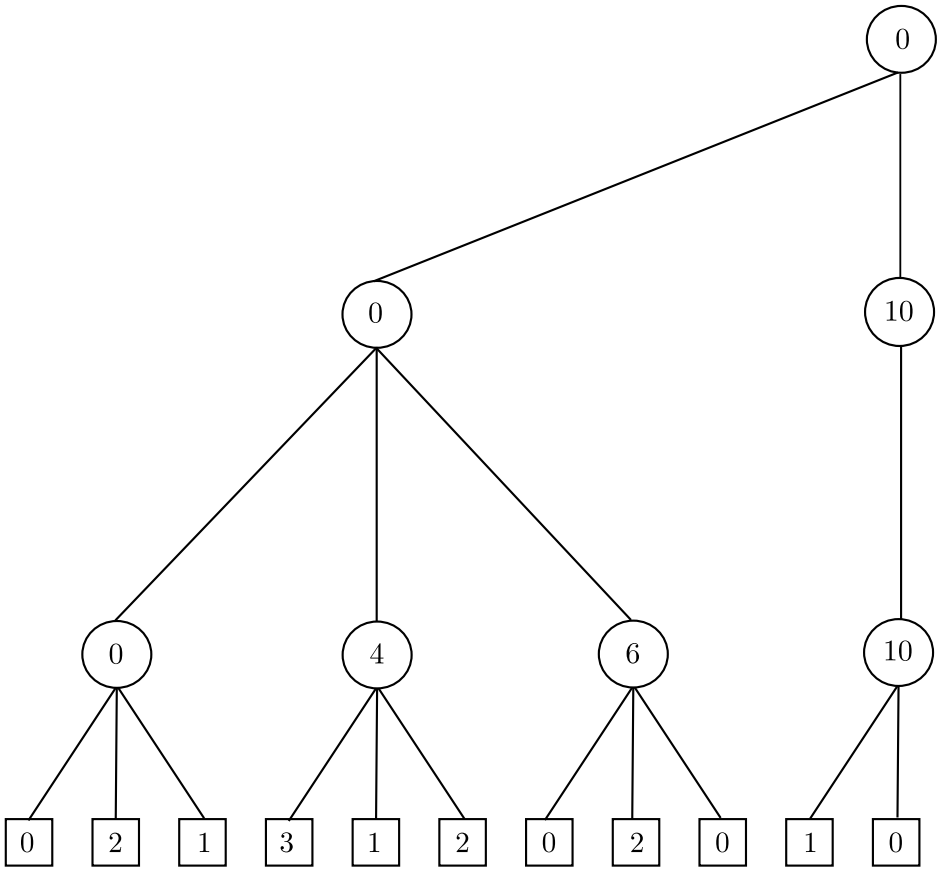


Figura 4.1: Estrutura de dados proposta por Cánovas e Navarro [CN10].

Como existem no máximo $O(\frac{n}{\mathcal{L}})$ nós internos, a estrutura em árvore gasta $O(\frac{n}{\mathcal{L}} \log n)$ bits. Se $\mathcal{L} = \Omega(\log n)$, então a estrutura final gasta $O(n)$ bits.

Para obter $RMQ(i, j)$, caso o intervalo $[i, j]$ estiver no mesmo bloco, basta realizar uma inspeção linear, que leva tempo $O(t_{LCP} \cdot \mathcal{L})$ para encontrar $RMQ(i, j)$, pois cada inspeção individual em um bloco com $\mathcal{L}$ folhas leva tempo t_{LCP}. Caso contrário, para realizar consultas de RMQ sobre o intervalo $[i, j]$, é necessário achar três valores:

- RMQ_i: A posição de RMQ do intervalo $[i, \mathcal{L}\lceil\frac{i+1}{\mathcal{L}}\rceil - 1]$.
- RMQ_j A posição de RMQ do intervalo $[\mathcal{L}\lfloor\frac{j}{\mathcal{L}}\rfloor, j]$.
- RMQ_{ij} a Posição de RMQ do intervalo $[\mathcal{L}\lceil\frac{i+1}{\mathcal{L}}\rceil, \mathcal{L}\lfloor\frac{j}{\mathcal{L}}\rfloor - 1]$.

As posições RMQ_i e RMQ_j podem ser encontradas em tempo $O(t_{LCP} \cdot \mathcal{L})$ com uma inspeção linear sobre os blocos de i e j, respectivamente.

A posição RMQ_{ij} pode ser encontrado com uma pesquisa *bottom-up* da árvore, partindo do nó v, pai do bloco em que i se encontra e partindo do nó w, pai do bloco em que j se encontra. A partir dos nós v e w, ao realizar uma pesquisa bottom-up alternando entre as duas origens, é possível descobrir o valor de RMQ de um intervalo cada vez maior ao examinar os filhos dos ancestrais durante o percurso, desde que estes filhos alcancem apenas blocos dentro de $[i, j]$. Ao examinar o rótulo dos filhos, atualiza-se a posição k mais à esquerda que contém o valor de mínimo $LCP[k]$. Eventualmente, o ancestral comum mais baixo $u = LCA(v, w)$, que também

é o critério de parada, é alcançado pela árvore ser completa e pela busca ser alternante entre as origens. De acordo com definição da estrutura proposta, $RMQ_{ij} = k$. Esse processo é descrito no Algoritmo 7.

Algoritmo 7 Obtenção de $RMQ(i, j)$.

Input LCP
Output $RMQ(i, j)$
  if i e j estão no mesmo bloco **then**
    Pesquisa linear por $[i, j]$
  else
    Ache RMQ_i com base no bloco de i
    Ache RMQ_j com base no bloco de j
    Sejam v e w os pais dos blocos de i e j
    repeat
      for all Filhos de v que alcancem blocos contidos em $[i, j]$ **do**
        Atualize o valor de mínimo
      end for
      for all Filhos de w que alcancem blocos contidos em $[i, j]$ **do**
        Atualize o valor de mínimo
      end for
      $v \leftarrow$ PARENT(v)
      $w \leftarrow$ PARENT(w)
    until $v \neq w$
  end if
  return $RMQ(i, j)$

Após determinar as três posições, basta tomar a que possui o menor valor LCP dentre eles. Em caso de empate, a posição mais à esquerda é escolhida.

Como a árvore é completa, a altura da árvore é dada por $\Theta(\log \frac{n}{\mathcal{L}})$. Como cada nó durante o percurso da árvore é acompanhado pela examinação de no máximo $\mathcal{L}$ filhos para obtenção de RMQ_{ij}, a consulta de RMQ_{ij} leva tempo $t_{RMQ} = O(t_{LCP} \cdot \mathcal{L} \cdot \log \frac{n}{\mathcal{L}})$.

Para responder consultas de $NSV(i)$, primeiramente, a solução é procurada à direita de i ao examinar somente o bloco em que i se encontra. Esta inspeção linear leva tempo $O(t_{LCP} \cdot \mathcal{L})$. Caso a busca falhe em encontrar $k > i$ tal que $LCP[k] < LCP[i]$, a estrutura de dados é utilizada por meio de uma busca *bottom-up* sucedida por uma busca *top-down*. Seja v o nó pai do bloco de i. A partir de v a árvore é percorrida de baixo para cima seguindo os ancestrais. Durante esse percurso, para cada ancestral, os filhos que alcançam somente os blocos à direita do bloco de i são examinados. Se em algum dos filhos, o rótulo presente possuir um valor menor que $LCP[i]$ a busca é encerrada. Caso contrário, a busca é efetuada até chegar à raiz.

Após isso, um percurso *top-down* é efetuado ao examinar os descendentes do nó encontrado na etapa anterior. Para cada descendente, os filhos que alcançam somente os blocos à direita do bloco de i são examinados. Caso haja mais de um filho cujo valor de LCP seja menor que o valor de $LCP(i)$, a posição à esquerda é escolhida. O critério de parada é quando se alcança um bloco. A partir disto, uma inspeção linear é feita dentro do bloco para encontrar o valor $NSV(i)$, caso exista. Este processo é descrito pelo Algoritmo 8.

Algoritmo 8 Obtenção de $NSV(i)$.

Input LCP
Output $PSV(i)$
  Pesquisa linear no bloco de i
  if $NSV(i)$ não está no bloco de i **then**
    Seja v o pai do bloco de i
    while $v \neq$ ROOT **do**
      for all Filhos de v que alcancem blocos à direita do bloco de i **do**
        Escolha o filho mais à esquerda rotulado por k tal que $LCP[k] < LCP[i]$
        Saia do While
      end for
      $v \leftarrow$ PARENT(v)
    end while
    while v não for pai de um bloco **do**
      for all Filhos de v que alcancem blocos à direita do bloco de i **do**
        Escolha o filho mais à esquerda rotulado por k tal que $LCP[k] < LCP[i]$
        Atualize v para o filho escolhido
      end for
      if v não foi atualizado **then**
        $NSV(i) = n - 1$
      end if
    end while
    Faça uma busca linear no bloco filho de v
  end if
  return $NSV(i)$

A consulta de $PSV(i)$ é simétrica, basta maximizar a posição em vez de minimizar. Como exposto pelo Algoritmo. 9.

Da mesma maneira que t_{RMQ}, o tempo das consultas de $NSV(i)$ e $PSV(i)$ é dominado pelas pesquisas *bottom-up* e *top-down* da árvore, que levam tempo $t_{PNSV} = O(t_{LCP} \cdot \mathcal{L} \log \frac{n}{\mathcal{L}})$. Esse tempo é justificado pois a árvore tem altura $\Theta(\frac{n}{\mathcal{L}})$ e para cada nó durante o percurso da árvore, são inspecionados $\mathcal{L}$ filhos.

As operações da árvore de sufixos comprimida segundo a implementação proposta, foram implementadas como descrito na Seção 3.4. A complexidade de cada operação na implementação comprimida é descrita em função das consultas utilizadas e é sumarizada pela tabela 4.1. Como forma de contraste, a complexidade das mesmas operações em uma árvore de sufixos comum é exposta na mesma tabela.

Algoritmo 9 Obtenção de $PSV(i)$.

Input LCP
Output $PSV(i)$
  Pesquisa linear no bloco de i
  if $NSV(i)$ não está no bloco de i **then**
    Seja v o pai do bloco de i
    while $v \neq$ ROOT **do**
      for all Filhos de v que alcancem blocos à esquerda do bloco de i **do**
        Escolha o filho mais à direita rotulado por k tal que $LCP[k] < LCP[i]$
        Saia do While
      end for
      $v \leftarrow$ PARENT(v)
    end while
    while v não for pai de um bloco **do**
      for all Filhos de v que alcancem blocos à direita do bloco de i **do**
        Escolha o filho mais à direita rotulado por k tal que $LCP[k] < LCP[i]$
        Atualize v para o filho escolhido
      end for
      if v não foi atualizado **then**
        $PSV(i) = -1$
      end if
    end while
    Faça uma pesquisa linear no bloco filho de v
  end if
  return $PSV(i)$

Tabela 4.1: Complexidade das operações suportadas.

Operação	Implementação comprimida	Árvore de sufixos comum
ROOT	$O(1)$	$O(1)$
LEAF(v)	$O(1)$	$O(1)$
LOCATE(v)	$O(t_{\mathcal{A}})$	$O(1)$
PARENT(v)	$O(t_{PNSV})$	$O(1)$[1]
CHILDREN(v)	$O(\sigma \cdot t_{RMQ})$	$O(\sigma)$
CHILD(v, c)	$O(\sigma \cdot t_{RMQ})$	$O(\sigma)$
DEPTH(v)	$O(t_{RMQ})$	$O(1)$[1]
EDGE(u, v)	$O(t_{\mathcal{A}} + \|S\|)$	$O(1)$[2]
LCA(u, v)	$O(t_{RMQ})$	$O(1)$[3]
SLINK(v)	$O(t_{RMQ})$	$O(1)$[1]

[1]Desde que haja espaço extra para armazenar a informação.
[2]Desde que o texto esteja armazenado em memória principal.
[3]Desde que haja espaço extra para armazenar a informação e um pré-processamento tenha sido realizado.

Capítulo 5

Experimentos e comparações

Experimentos foram realizados com a implementação do índice. Além disso foram feitas comparações com outros índices. Mais especificamente, compararam-se:

- A implementação proposta por este trabalho que, conforme citado no Capítulo 4, é composta pelo método de Gonzáles e Navarro [GGMN05] para as consultas de $Rank$ e $Select$, pelo algoritmo incremental de construção de arranjos de sufixos comprimidos de Hon *et al.* [HLS$^+$07], pelos métodos de Kasai *et.al* [KLA$^+$01] e Sadakane [Sad02] para construção de LCP e pela estrutura de dados de Cánovas e Navarro para responder as consultas de RMQ, NSV e PSV;
- Um índice não-comprimido implementado pelos autores a fim de servir de controle para os testes. Este índice foi baseado em arranjo de sufixos não-comprimido enriquecido com informação de LCP não-comprimida e adicionado com a estrutura proposta por Cánovas e Navarro [CN10] para responder as consultas de RMQ, PSV e NSV. O arranjo de sufixos foi construído com a biblioteca `libdivsufsort` [Mor07]. O método de Kasai *et al.* [KLA$^+$01], abordado na Seção 2.3.4, foi implementado para calcular a informação de LCP não-comprimida. A implementação da estrutura de [CN10] é a mesma do índice comprimido implementado;
- Um índice comprimido implementado pelo grupo SuDS (Succinct Data Structures) da universidade de Helsinque [V. 13]. A implementação é baseada no trabalho de Sadakane [Sad07], onde a topologia da árvore é codificada sucintamente na forma de parênteses balanceados. As operações básicas também agem sobre essa representação de parênteses. Portanto, este índice difere da implementação do nosso trabalho por usar outra técnica para realização de percursos na árvore de sufixos.

A implementação foi codificada em C++ para um arquitetura de 32-bits, portanto a manipulação de arquivos extremamente grandes é impossibilitada por este fator, devido à capacidade de endereçamento das estruturas de dados que compõem o índice. Desta forma, a implementação apenas permite a manipulação de textos da ordem de centenas de megabytes. Todos os testes foram realizados em um computador *core i7-3700k* com $8\mathrm{GB}$ de memória RAM sobre o sistema operacional GNU/Linux Ubuntu 12.10.

Os índices foram avaliados e comparados segundo 3 quesitos:

1. Tempo de construção: tempo necessário para construção do índice.

2. Espaço: espaço final necessário para representar o índice e o pico de memória usado durante a construção do mesmo.

3. Custo de operação: tempo de operações básicas sobre o índice tal como:

   - Tempo de acesso a $\mathcal{A}[i]$. Obtido através da média aritmética do tempo de 10^6 acessos em posições arbitrárias de $\mathcal{A}$.
   - Tempo de acesso a $LCP[i]$ para algum i. Obtido através da média aritmética do tempo de 10^6 acessos em posições arbitrárias de LCP.
   - Tempo de acesso a $RMQ(i, j)$. Obtido através da média aritmética do tempo de 10^5 acessos em intervalos arbitrárias.
   - Tempo de acesso a $NSV(i)$. Obtido através da média aritmética do tempo de 10^5 acessos em posições arbitrárias i.
   - Tempo de acesso a $PSV(i)$ para algum i. Obtido através da média aritmética do tempo de 10^5 acessos em posições arbitrárias i.
   - Tempo de acesso a $LCA(u, v)$. Obtido através da média aritmética do tempo de 10^5 acessos em folhas arbitrárias u e v quaisquer.

Para verificar diferentes configurações do índice proposto, os parâmetros $\mathcal{B}'$, $\mathcal{L}$ e $\mathcal{K}$, mencionados em capítulos anteriores, foram ajustados. Desta forma, o índice pode gastar menos (mais) espaço para a sua representação e gastar mais (menos) tempo para consulta e construção, isto é, ao diminuir o uso de um recurso, aumenta-se o tempo do outro, já que o índice comprimido baseia-se na troca de espaço por tempo. Foram criados quatro cenários para testar o comportamento da implementação proposta:

1) Cenário pior espaço-eficiente e mais rápido com parâmetros $\mathcal{B}' = 60$, $\mathcal{K} = 10$ e $\mathcal{L} = 8$.

2) Cenário intermediário com parâmetros $\mathcal{B}' = 60$, $\mathcal{K} = 20$ e $\mathcal{L} = 16$.

3) Cenário melhor espaço-eficiente e mais lento com parâmetros $\mathcal{B}' = 60$, $\mathcal{K} = 20$ e $\mathcal{L} = 32$.

4) Cenário melhor espaço-eficiente com parâmetros $\mathcal{B}' = 60$, $\mathcal{K} = 20$ e $\mathcal{L} = 32$. A memória neste cenário foi limitada a $580\mathrm{MB}$ de memória RAM.

Em todos os cenários $\mathcal{B}'$ foi configurado com o valor 60 em todos os cenários pois demonstrou empiricamente ser uma boa escolha para balancear o tempo de construção e memória de pico utilizada.

Para mensurar o tempo gasto, o comando de sistema `time` foi utilizado. Cada operação foi executada 10^5 vezes em posições aleatórias e a média foi calculada.

Para obter a memória de pico, um `shellscript` que monitora periodicamente a memoria utilizada pelo processo foi usado.

A informação do espaço final requerido pela estrutura foi coletada usando o próprio sistema de arquivos, visto que ele informa o tamanho final do índice. Uma vez que o índice era construído, ele era salvo em disco para que, quando fosse utilizado novamente, bastasse carregá-lo deste local sem necessidade da reconstrução do índice.

Para os testes geraram-se aleatoriamente textos contendo o alfabeto de ADN $\Sigma = \{a, c, t, g\}$ com um programa codificado em C++.

Até o momento da defesa, os seguintes resultados foram submetidos para a *Conferencia Latinoamericana en Informática* (CLEI 2013).

A Tabela 5.1 expõe a nomenclatura adotada para o entendimento dos gráficos apresentados. Além disso, em cada gráfico uma legenda foi colocada indicando o espaço final, em bits por símbolo, que cada estrutura ocupa.

Tabela 5.1: Nomenclatura adotada para os gráficos.

Nome	Significado
CST	Implementação proposta
SuDS CST	Implementação do índice SuDS
Raw ST	Implementação da estrutura não-comprimida

5.1 Tempo de construção

O primeiro critério avaliado foi o tempo de construção. O gráfico da Figura 5.1 ilustra o cenário mais rápido e pior espaço-eficiente da implementação para este quesito. Por sua vez o gráfico da Figura 5.2 ilustra o cenário intermediário. Por fim, o gráfico da Figura 5.3 ilustra o comportamento do cenário melhor espaço-eficiente da implementação.

Após a observação destes gráficos fica claro a troca de espaço por tempo mesmo na construção do índice proposto.

O cenário pior espaço-eficiente é o mais rápido, visto que os fatores de amostragem $\mathcal{K}$ e $\mathcal{L}$ são os menores. Isso pode ser explicado pois a construção da informação de LCP depende do acesso a $\mathcal{A}$. Por sua vez, a montagem da estrutura de Cánovas e Navarro [CN10], depende de acessos à LCP.

O cenário intermediário perde do cenário melhor espaço-eficiente pois o fator de amostragem $\mathcal{K}$ é o mesmo, diferindo no fator $\mathcal{L}$ que é menor no cenário intermediário. Com isso, a árvore proposta por Cánovas e Navarro [CN10] tem mais nós internos, e logo, demora mais tempo para ser construída, enquanto no cenário melhor espaço-eficiente esta mesma estrutura possui menos nós. No entanto, o cenário intermediário ganha do melhor espaço-eficiente pelo fator de amostragem $\mathcal{K}$ ser maior.

Em comparação a outros índices, temos que o tempo de construção do índice proposto é maior do que os dois comparados. A implementação proposta é pelo menos duas vezes mais lenta que o índice SuDS. O tempo de construção do índice não-comprimido é desprezível comparado tanto à implementação proposta quanto ao índice SuDS.

No entanto, para diversas aplicações, como mapeamento de *reads* em um genoma de referência, visto na Seção 1.1.2, o tempo de construção do índice não é crucial, visto que o índice do genoma só necessita ser construído uma vez para possibilitar o mapeamento de quantas *reads* forem necessárias. Esta é a ideia por trás de diversos software de mapeamento, como o BWA [LD09], Bowtie [LTPS09] e Segemelh [HOK+09]. Logo, nossa implementação pode ser mais útil para aplicações cujo índice necessite ser construído uma única vez, como no mapeamento de *reads*.

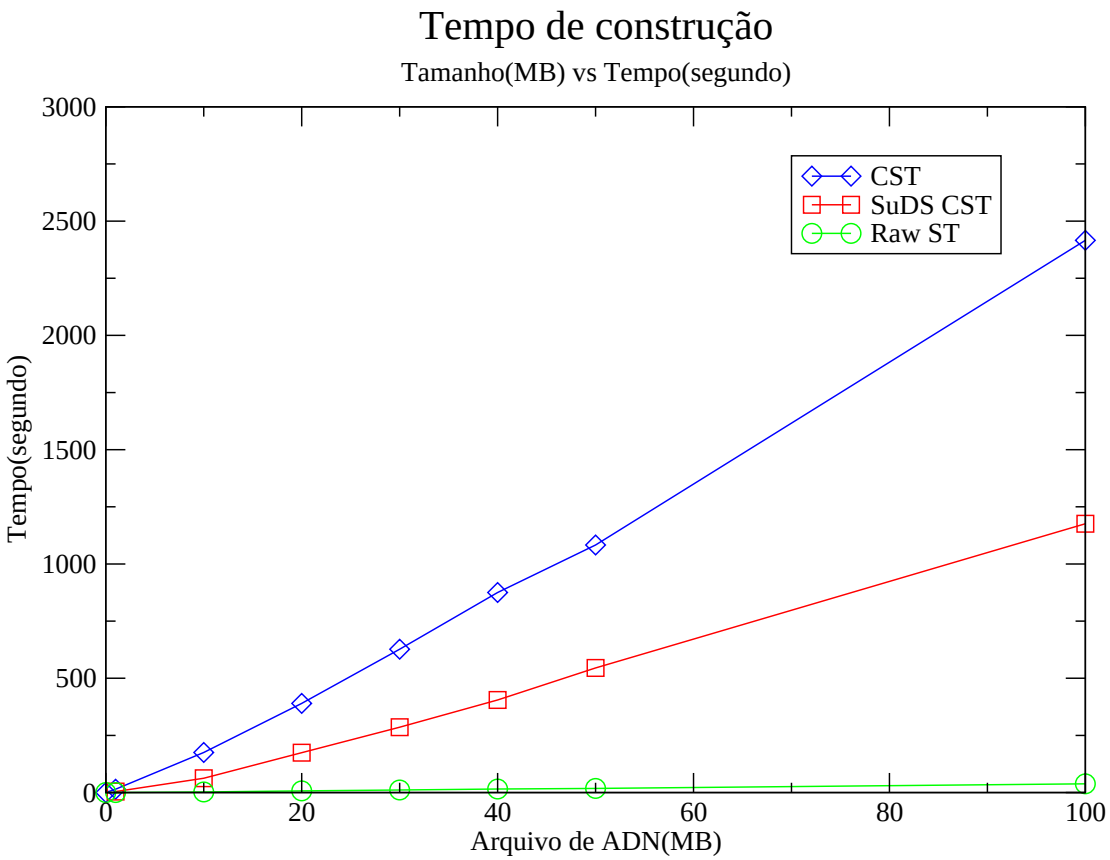


Figura 5.1: Tempo de construção para $\mathcal{K} = 10$ e $\mathcal{L} = 8$. ≈ 19 bits por símbolo.

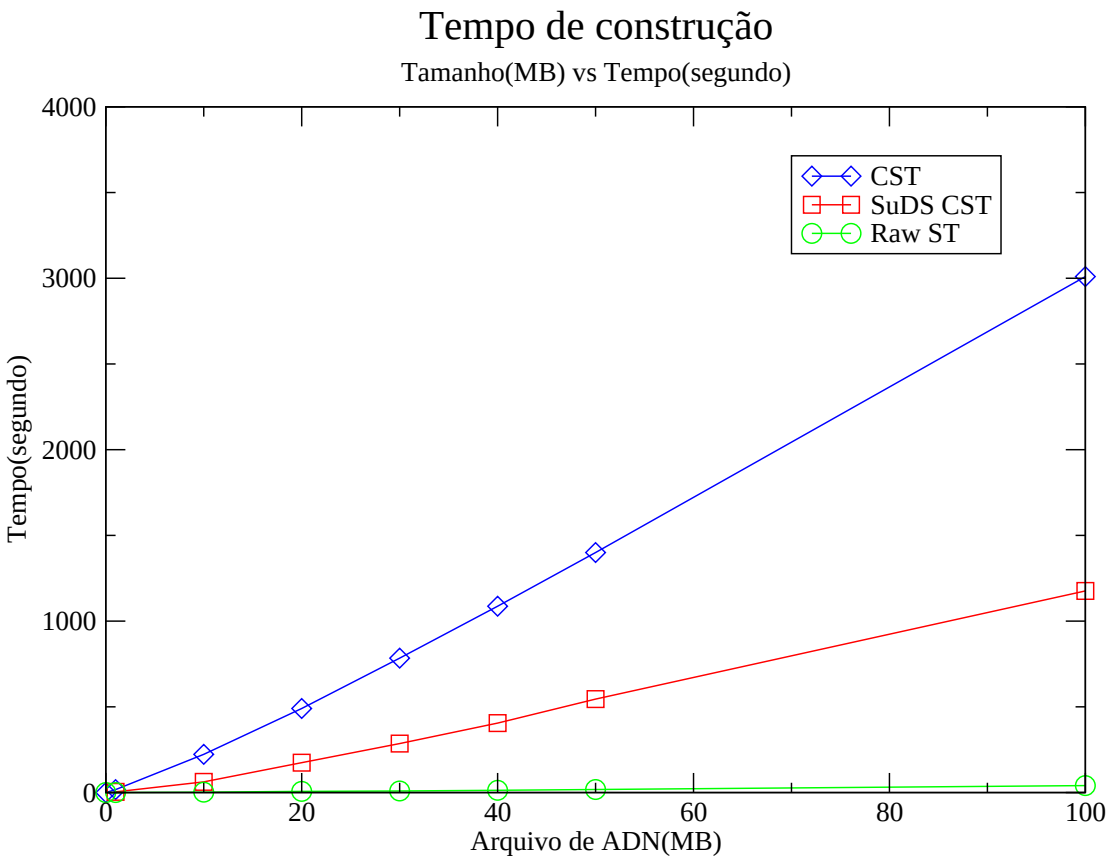


Figura 5.2: Tempo de construção para $\mathcal{K} = 20$ e $\mathcal{L} = 16$. ≈ 14 bits por símbolo.

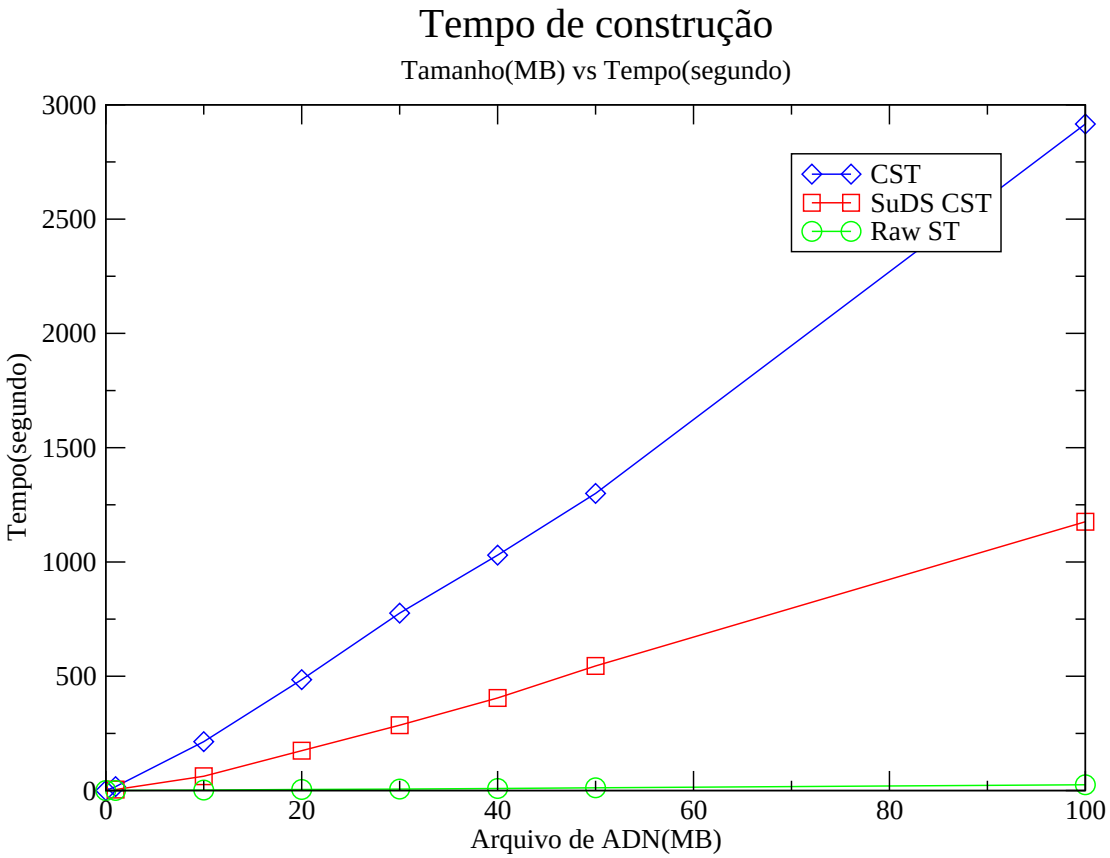


Figura 5.3: Tempo de construção para $\mathcal{K} = 20$ e $\mathcal{L} = 32$. ≈ 13 bits por símbolo.

5.2 Espaço

Ao analisar o recurso de espaço foram considerados o espaço final que a estrutura ocupa e a memória de pico, isto é, a maior quantidade de memória que a estrutura ocupa considerando todo o seu tempo de vida.

O gráfico da Figura 5.4 expõe o cenário pior espaço-eficiente. O gráfico da Figura 5.5 ilustra o cenário intermediário. Finalmente, o gráfico da Figura 5.6 refere-se ao cenário melhor espaço-eficiente e mais lento.

É observado que a implementação proposta possui a menor memória de pico dentre todos os cenários, requerendo um pouco mais na construção do que o necessário para a representação final da estrutura. Considerando o índice SuDS um fator pouco maior que $4\times$ sobre a entrada é observado ao considerar a memória de pico.

No primeiro cenário, ilustrado pelo gráfico da Figura 5.4, os fatores de amostragem $\mathcal{K}$ e $\mathcal{L}$ são os menores utilizados no experimento, portanto a implementação mostra-se pior espaço-eficiente em relação aos outros cenários. Para este cenário, o pico de memória possui um fator

de $\approx 2.5\times$ sobre o tamanho da entrada. Ao considerar o espaço de representação da estrutura, a implementação proposta gasta mais espaço que o índice SuDS.

No segundo cenário, ilustrado pelo gráfico da Figura 5.5, a implementação proposta é equivalente em termos de espaço de representação final da estrutura em relação ao índice SuDS, um fator um pouco menor que $1.8\times$ em relação ao tamanho da entrada. A memória de pico da implementação requer pouco espaço adicional em comparação ao espaço final utilizado pela mesma.

No terceiro cenário, ilustrado pelo gráfico da Figura 5.6, a implementação proposta demonstra ser mais espaço eficiente do que nos outros cenários. O espaço utilizado pela estrutura possui um fator de $\approx 1.5\times$ sobre o tamanho da entrada, gastando menos espaço que o índice SuDS. A memória de pico também representa os menores valores dentre os três cenários.

Nota-se que o índice não-comprimido requer um uso enorme de espaço, pouco importando a escolha de $\mathcal{L}$. Ele requer um fator de $12\times$ de memória de pico e um fator de $\approx 10\times$ para representação final da estrutura nos cenários utilizados. Este fator torna a manipulação de textos grandes inviável na prática quando não há memória suficiente.

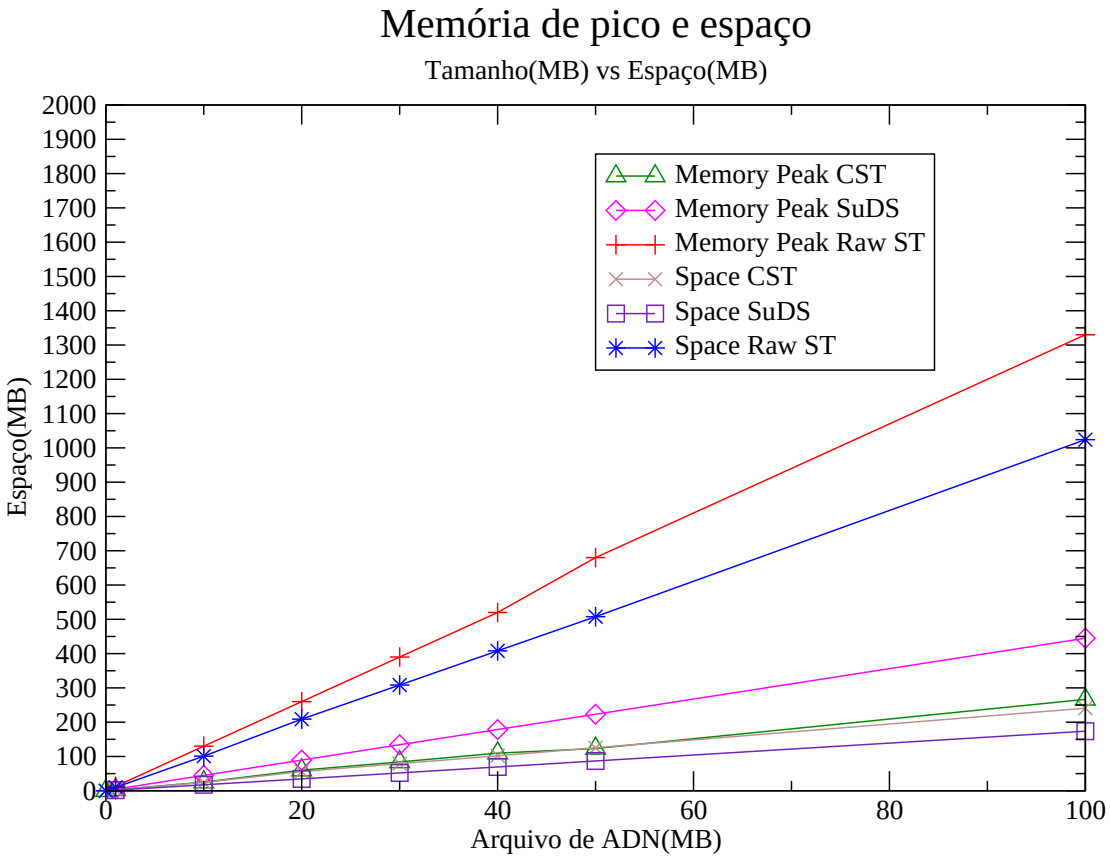


Figura 5.4: Espaço e pico de memória para $\mathcal{K} = 10$ e $\mathcal{L} = 8$. ≈ 19 bits por símbolo.

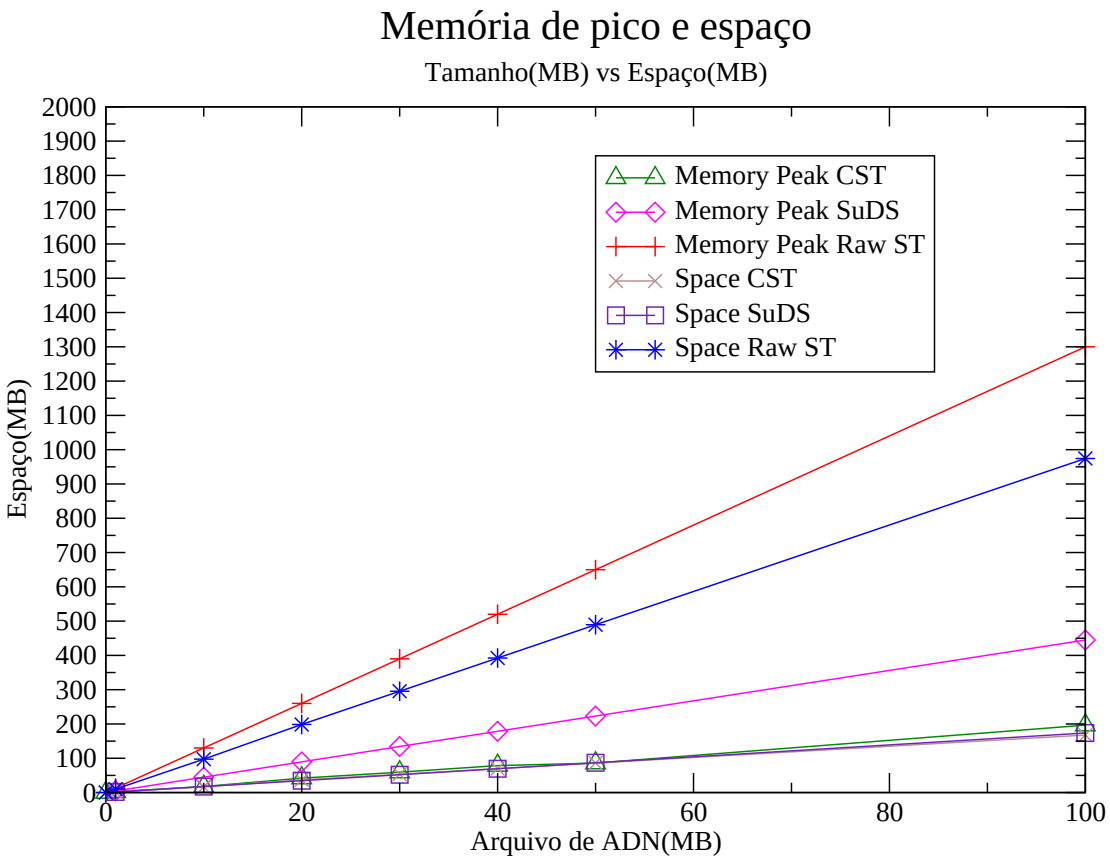


Figura 5.5: Espaço e pico de memória para $\mathcal{K} = 20$ e $\mathcal{L} = 16$. ≈ 14 bits por símbolo.

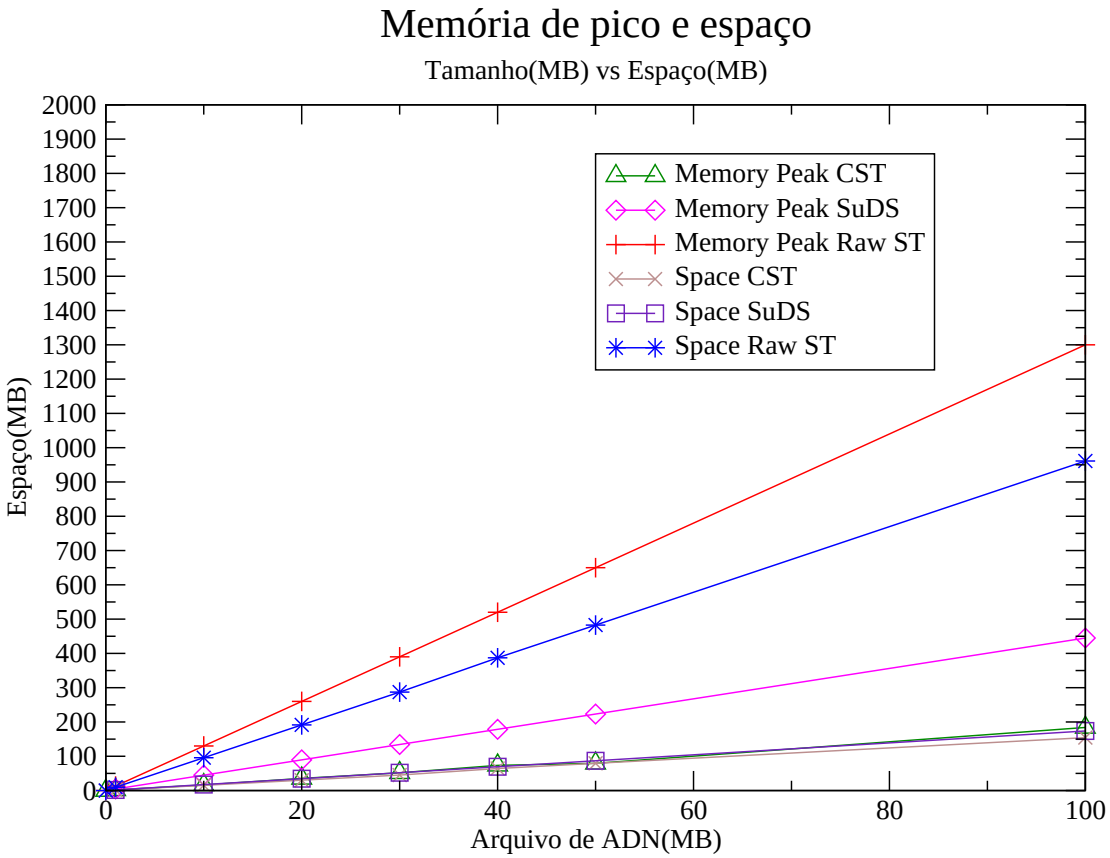


Figura 5.6: Espaço e pico de memória para $\mathcal{K} = 20$ e $\mathcal{L} = 32$. ≈ 13 bits por símbolo.

O gráfico da Figura 5.7 ilustra o espaço consumido apenas pela implementação proposta nos três diferentes cenários. O cenário mais rápido, com $\mathcal{K} = 10$ e $\mathcal{L} = 8$; o cenário intermediário, com$\mathcal{K} = 20$ e $\mathcal{L} = 16$ e o cenário mais lento, com $\mathcal{K} = 20$ e $\mathcal{L} = 32$.

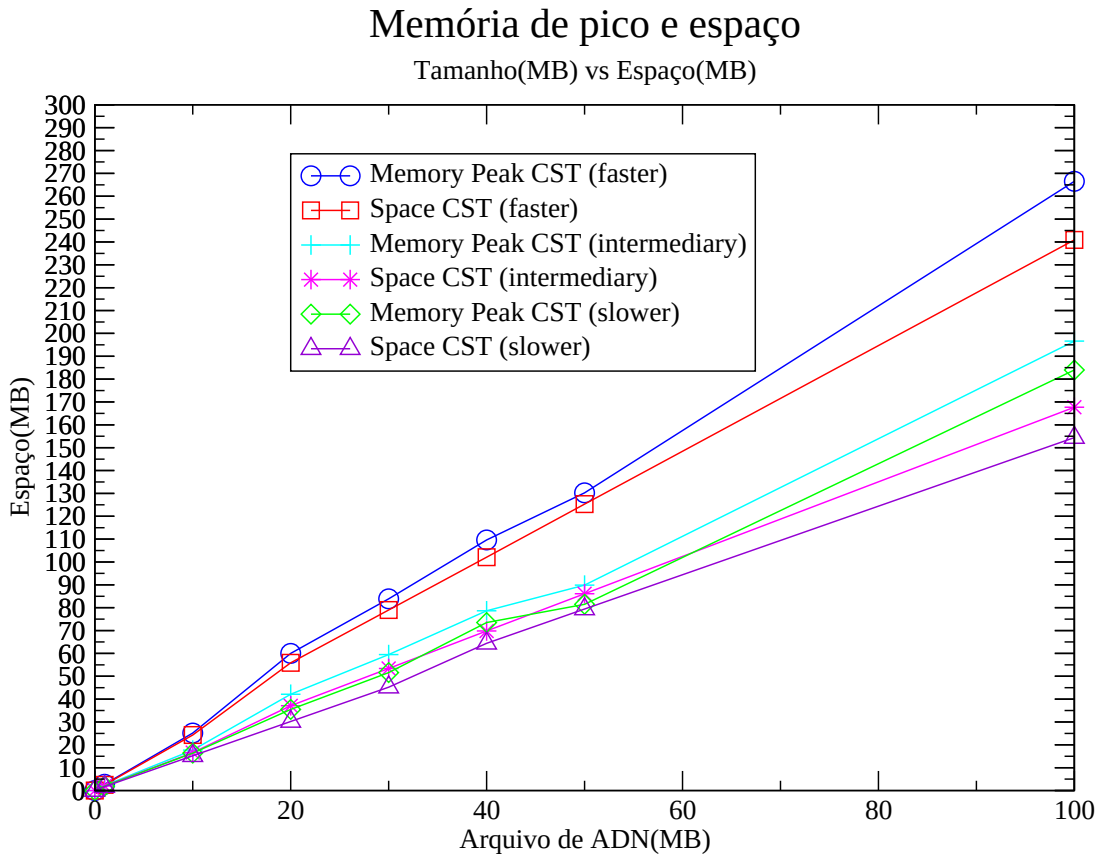


Figura 5.7: Espaço e pico de memória para os cenários.

5.3 Custo de operações

O custo de operações foi medido para realizar uma comparação entre a implementação proposta e o índice não-comprimido, visto que ambos baseiam-se em operações de RMQ, PSV e NSV. Além dessas operações, foram analisados o tempo de acesso à $\mathcal{A}$, LCP, e à consulta de ancestral comum mais baixo (LCA) entre duas folhas.

O gráfico da Figura 5.8 expõe o cenário pior espaço-eficiente e mais rápido. O gráfico da Figura 5.9 ilustra o cenário intermediário. Finalmente, o gráfico Figura 5.10 refere-se ao cenário melhor espaço-eficiente.

Nos três cenários fica evidente a troca de espaço por tempo. Quanto maior o fator de amostragem, menor o espaço, mas o tempo de computação aumenta para recuperação das entradas não amostradas. Isso influencia diretamente no custo de operações, que baseiam-se em acessos a Ψ, $\mathcal{A}$ e LCP.

O custo de operações mais simples na implementação comprimida, como acessos a $\mathcal{A}$ e a LCP, gasta apenas alguns microssegundos, a mesma ordem do que operações mais complexas no índice não-comprimido, como a operação de RMQ e LCA. Já as operações mais complexas na implementação proposta estão na ordem de centenas de microssegundos (como consultas de

PSV e NSV) e na ordem de milissegundos para as operações mais complexas RMQ e LCA. O preço da economia do espaço reflete-se em um custo de operação básica maior.

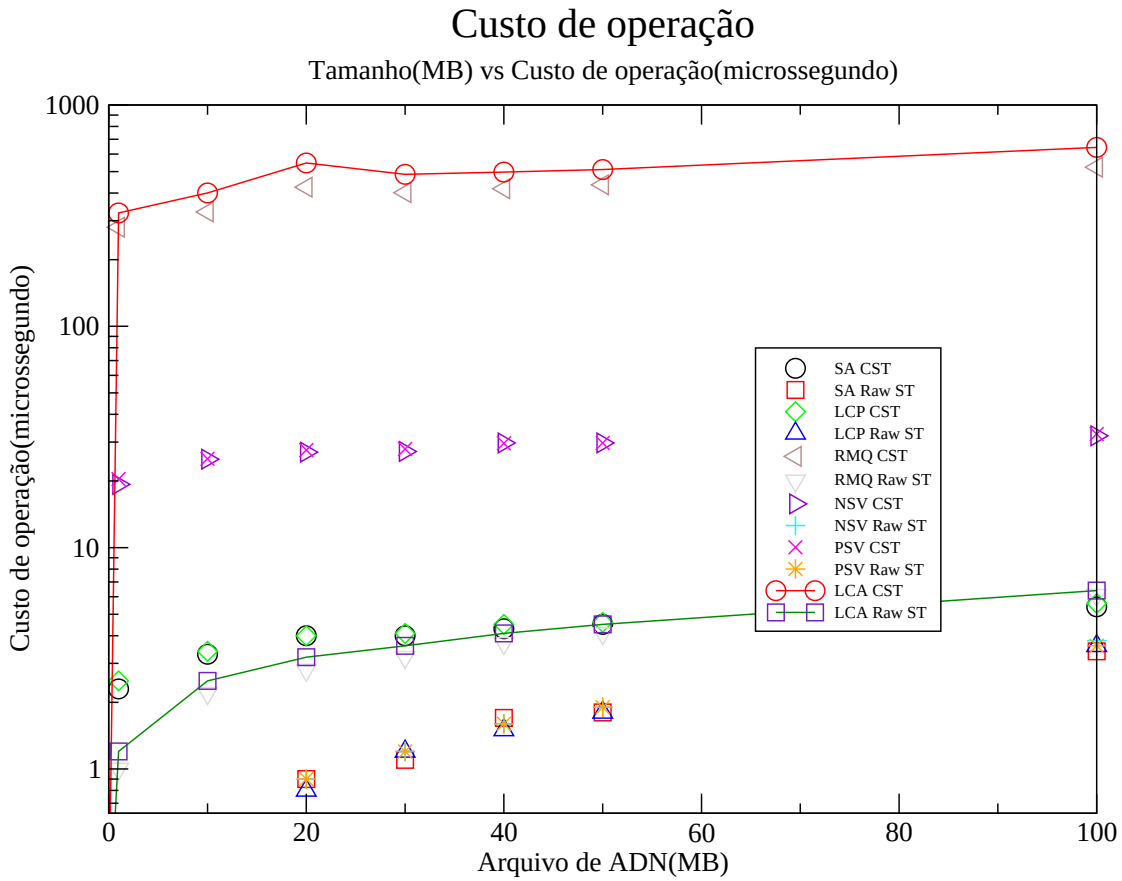


Figura 5.8: Custo de operação para $\mathcal{K} = 10$ e $\mathcal{L} = 8$. ≈ 19 bits por símbolo.

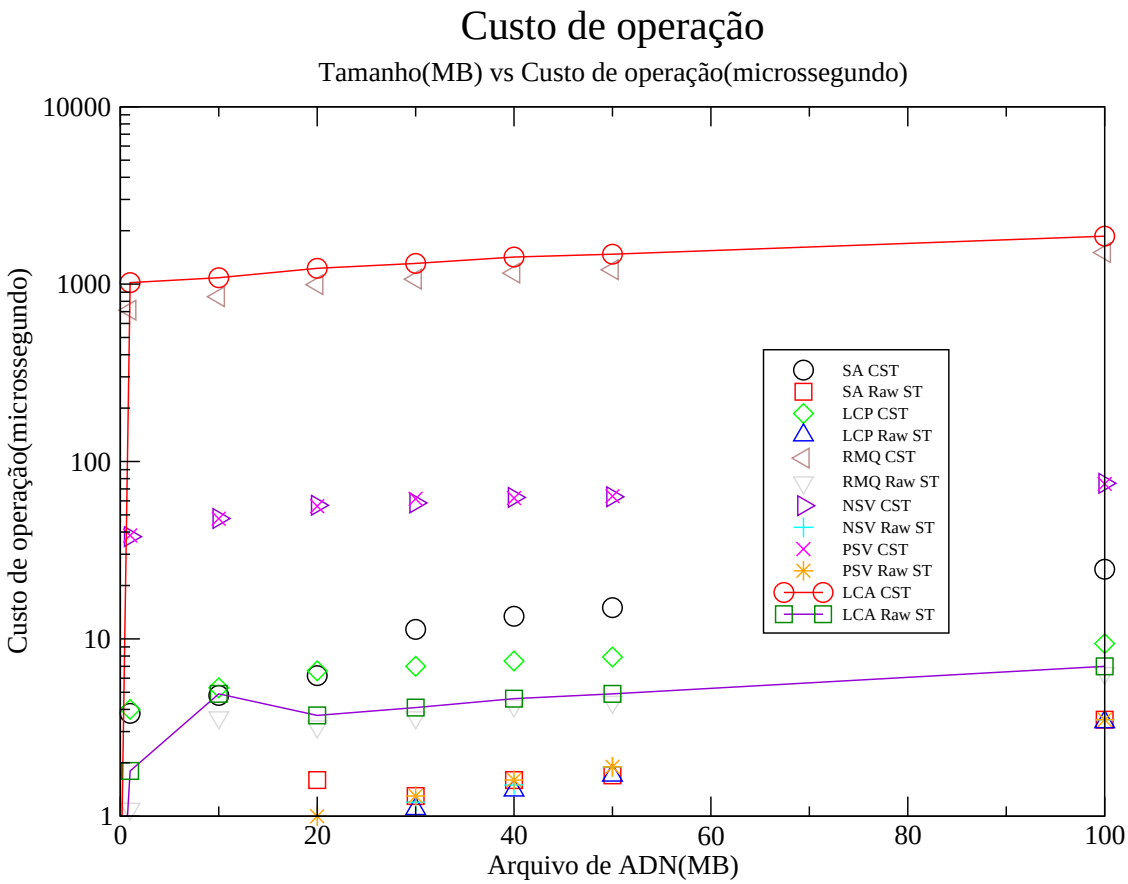


Figura 5.9: Custo de operação para $\mathcal{K} = 20$ e $\mathcal{L} = 16$. ≈ 14 bits por símbolo.

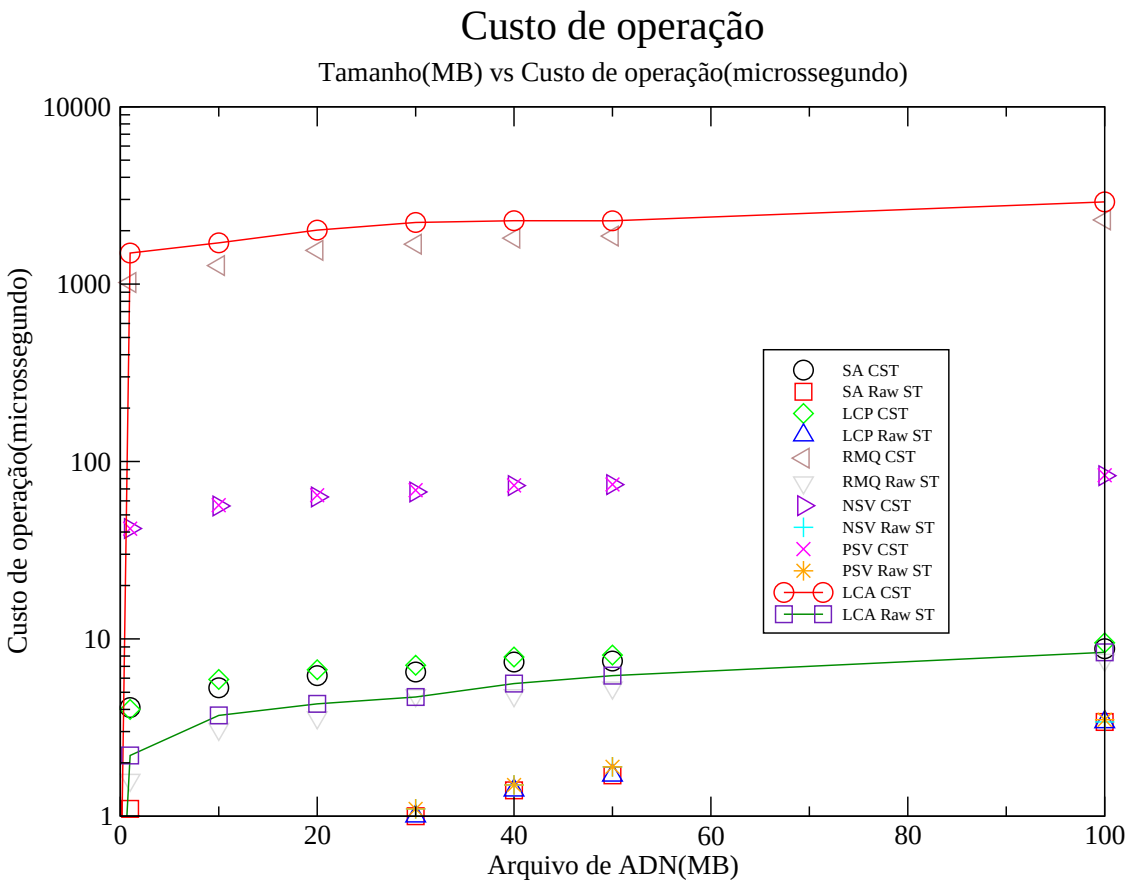


Figura 5.10: Custo de operação para $\mathcal{K} = 20$ e $\mathcal{L} = 32$. ≈ 13 bits por símbolo.

No entanto, estes cenários não refletem a importância dos índices comprimidos quando a memória é escassa. O Gráfico 5.11 ilustra o cenário melhor espaço-eficiente quando a memória disponível não é suficiente para armazenar o índice não-comprimido inteiramente em memória.

No experimento, a memória foi definida para $580MB$ para simular uma situação em que pouca memória estivesse disponível. Como o índice não-comprimido não cabe em memória, muitas páginas tem que ser recuperadas do disco e portanto, muitos acessos a disco começam a ser realizados em vez de acessos à memória. Este fenômeno causa uma degradação do custo de operação do índice não-comprimido, conforme o tamanho de entrada aumenta, o custo de operações fica excessivo, até que o custo das operações do índice não-comprimido seja superior ao custo de operação da implementação proposta.

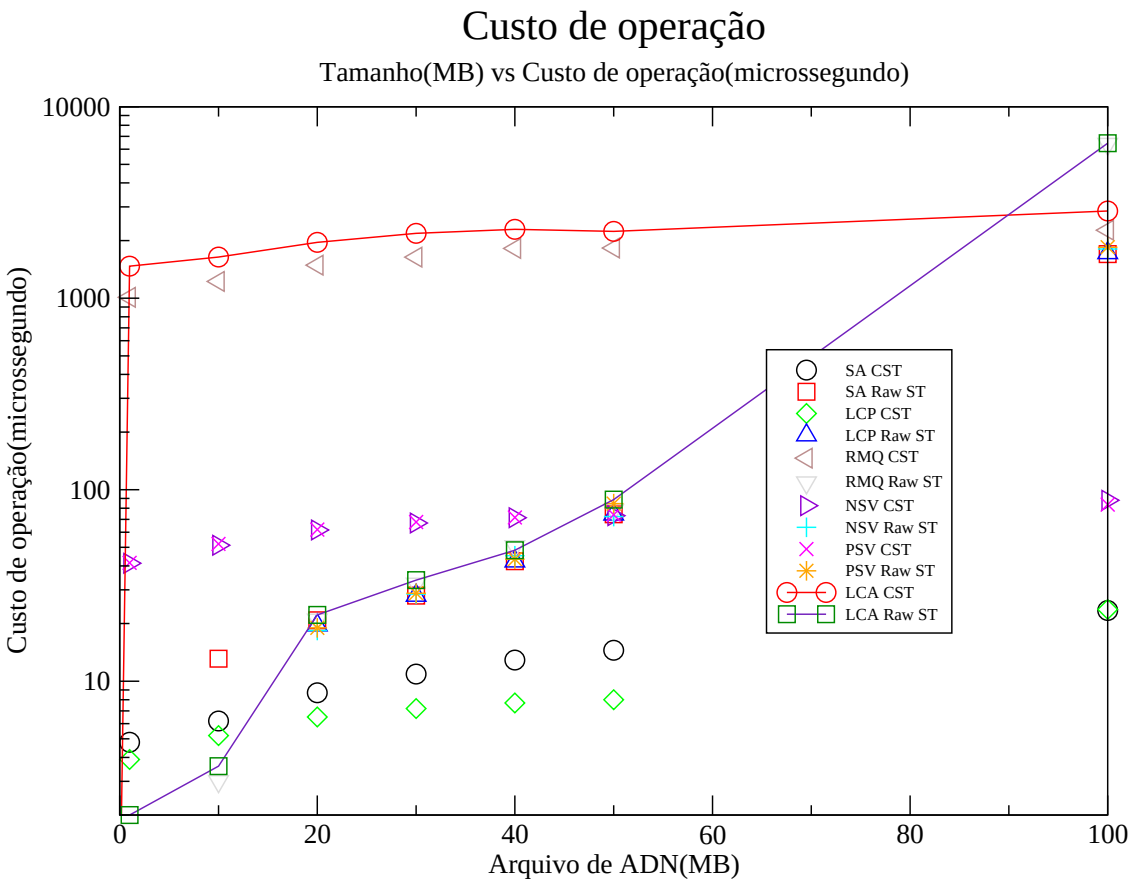


Figura 5.11: Custo de operação para $\mathcal{K} = 20$ e $\mathcal{L} = 32$ com $580MB$ disponíveis de memória RAM. ≈ 13 bits por símbolo.

5.4 Análise

Conforme visto, devido ao algoritmo incremental proposto por Hon *et al.* [HLS+07], a implementação proposta mostra-se muito eficiente quanto à memória de pico utilizada durante a construção, requerendo pouco mais que o espaço final para representação da estrutura. Em comparação ao índice SuDS, a implementação neste trabalho mostra-se muito mais eficiente caso haja pouca memória disponível, visto que este índice requer um fator maior que $4\times$ de memória em relação ao tamanho da entrada para sua construção. Já o índice não-comprimido usa um fator de $12\times$ de pico de memória em relação ao tamanho da entrada, o que pode tornar o seu uso inviável quando pouca memória está disponível.

O tempo de construção da estrutura na implementação proposta é o maior em todos os cenários, tendo um fator de $\approx 2\times$ sobre o índice SuDS. Por sua vez, o índice não-comprimido possui um tempo de construção substancialmente menor que os demais em todos os cenários. No entanto, o tempo de construção não é crítico para algumas aplicações, conforme mencionado anteriormente, uma vez que a estrutura necessita ser construída apenas uma vez. Já o espaço

pode ser crucial para algumas aplicações de modo que é essencial que a estrutura caiba em memória principal durante o seu uso e durante a sua construção.

Por fim, o índice não-comprimido possui um custo menor de operação que a implementação proposta devido à troca de espaço por tempo da implementação proposta. No entanto, quando pouca memória está disponível, a implementação proposta demonstra ser de grande valia em prática, pois ela cabe inteiramente em memória principal e evita acessos a disco, o que evita uma degradação do desempenho, ao contrário do que ocorre com o índice não-comprimido, que quando não cabe em memória principal, é totalmente degradado pelas operações em disco.

Capítulo 6

Considerações finais

Neste trabalho, propusemos uma implementação espaço-eficiente baseada em árvores de sufixos comprimidas.

Após experimentos, foi observado o valor prático da implementação proposta em comparação a outros índices existentes na literatura. Vários mecanismos podem ser incorporados ao índice para melhorar seu desempenho de tempo e espaço bem como a sua flexibilidade. Além disso, novas comparações deverão ser feitas para ratificar a eficiência da implementação proposta em trabalhos futuros.

6.1 Discussão

Os índices desenvolvidos se mostram extremamente úteis por serem capazes de manipular informação de maneira eficiente para a resolução de vários problemas essenciais tais como, mapeamento de fragmentos em Bioinformática, codificação Huffman em processamento de sinais ou até mesmo casamento de padrões em Mineração de Dados. Portanto, estas estruturas de dados dão origem a uma vasta gama de aplicações práticas que compreendem vários campos da Ciência da Computação

Como visto no Capítulo 5, a implementação proposta demonstra ser muito eficiente com respeito ao uso de memória de pico.

- Enquanto o índice SuDS requer um fator de mais de $4\times$ sobre o tamanho da entrada, a implementação proposta requer um fator menor que $2\times$ em um cenário mais espaço-econômico. A memória de pico do índice comprimido requer um fator de $\times 12$ em relação ao tamanho da entrada, muito maior que o fator dos índices comprimidos.
- Outro ponto notável da implementação é que a memória de pico não é muito maior do que o espaço final para representar a estrutura, ou seja, não é necessário uma quantia considerável de memória adicional durante a construção da estrutura de dados o que a diferencia do índice SuDS e do índice não-comprimido.

A implementação também demonstra ser espaço-eficiente quanto ao espaço final da estrutura.

- Em cenários menos espaço-econômicos, ela perde por pouco em relação ao índice SuDS.
- Em cenários mais espaço econômicos, a implementação supera o índice SuDS.

Em relação ao tempo de construção, a implementação possui um fator de $\approx 2\times$ em relação ao índice SuDS. O índice não-comprimido possui um tempo de construção desprezível em relação aos índices comprimidos, tanto ao índice SuDS quanto ao da implementação proposta. No entanto, para algumas aplicações, a estrutura final só precisa ser construída uma vez, tornando o tempo de construção em um fator não crucial.

Experimentos realizados com relação ao custo de operação demonstraram que a implementação proposta leva mais tempo para cada operação em relação ao índice não-comprimido, um preço que se paga pela compressão. No entanto, quando a máquina dispõe de pouca memória a implementação mostra-se extremamente útil. Neste caso, o índice não-comprimido não cabe em memória principal, observando-se que o desempenho do mesmo se degrada completamente e se torna inferior à da implementação proposta, que não sofre degradação, pois cabe em memória principal.

Sendo assim, o presente trabalho oferece uma alternativa espaço-econômica competitiva em relação aos índices comparados, tornando seu uso muito útil em prática.

6.2 Progresso e trabalhos futuros

Diversas características e mecanismos podem ser incorporados à implementação proposta para melhorar os recursos de tempo de construção, custo de operação e espaço utilizado e a manipulação de textos. Estas melhorias são:

Arranjos de sufixos comprimidos: A implementação constitui do arranjo de sufixos comprimido descrito por Grossi e Vitter [GV05] e construído de acordo com o algoritmo incremental de Hon *et al.* [HLS$^+$07]. Outras formas de arranjos de sufixos comprimidos, como a proposta em [MN05] por Mäkinen e Navarro, podem ser implementadas futuramente para possível melhoria de espaço e tempo. Além disso, testes realizados com o índice-FM como estrutura equivalente ao arranjo de sufixos comprimidos podem apresentar resultados mais rápidos e melhor espaço-eficientes, como sugerido em [HLS$^+$04].

Arquitetura de 64-bits: Como a implementação foi criada para uma arquitetura de 32-bits textos grandes da ordem de gigabytes não podem ser manipulados. Ao portá-la para uma arquitetura de 64-bits será possível manipular textos ainda maiores dando origem a uma gama maior de aplicações.

Consultas de $Rank$ e $Select$: De acordo com Navarro e Providel [NP12], o método proposto para as consultas mostra-se bem mais eficiente do que o trabalho de Gonzáles e Navarro [GGMN05], o qual nosso trabalho foi baseado. Uma implementação do método mais recente forneceria uma melhora significante no tempo de acesso à Ψ e à LCP. Pois se o acesso de LCP fosse mais eficiente, as consultas de RMQ, PSV e NSV também seriam, visto que dependem do acesso à LCP. O tempo de construção também seria reduzido graças à melhora dos tempos de acesso ao arranjo de sufixos comprimidos, à Ψ e a LCP.

Representação de Ψ: No trabalho de Hon *et al.* [HLS$^+$04], os autores mencionam que o uso do código γ de Elias para codificar a função Ψ gasta menos espaço em relação ao código de Rice, usado em nossa implementação. Desta forma, ao implementar esse código, tanto

a memória de pico quanto o espaço final da estrutura tem potencial para serem reduzidos ainda mais.

Construção de LCP: Os métodos apresentados em [KMP09], [Fis11] e [BGOS11], mostram-se mais rápidos na prática do que os métodos de Kasai *et al.* [KLA$^+$01] e Sadakane [Sad02], usados em nossa implementação, para construção de uma informação espaço-eficiente de LCP. Com tais métodos, o tempo de construção total diminuiria já que a construção da informação de LCP seria mais rápida.

Com a adição destas melhorias, a implementação proposta poderá obter resultados ainda mais eficientes e interessantes na prática, dando origem a uma variedade maior de aplicações.

Comparações foram realizadas com um índice baseado na biblioteca `libdivsufsort` [Mor07] e com o índice SuDS [V. 13]. Outros índices tais como os expostos em [CN10] e [OFG10] serão comparados em trabalho posterior com propósito de identificar as vantagens e desvantagens da implementação proposta em relação a eles. A figura 3.1 ilustra os trabalhos que podem contribuir para melhoria da implementação proposta.

Referências

[AKO02] M. Abouelhoda, S. Kurtz, and E. Ohlebusch. The Enhanced Suffix Array and Its Applications to Genome Analysis. In *Workshop on Algorithms in Bioinformatics*, volume 2452 of *Lecture Notes in Computer Science*, pages 449–463. Springer Verlag, 2002. 15, 18, 19

[AKO04] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004. 18, 23, 27, 28

[Apo85] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *Nato Asi Series F, Computer and System Sciences*, pages 85–96. Springer Verlag, 1985. 14

[ARC03] M. Ayala-Rincón and P. Conejo. A linear time lower bound on mccreight and general updating algorithms for suffix trees. *Algorithmica*, 37(3):233–241, 2003. 15

[BGOS11] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. In *String Processing and Information Retrieval*, volume 7024 of *Lecture Notes in Computer Science*, pages 197–208. Springer Verlag, 2011. 78

[BM77] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977. 2

[BW94] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994. 34, 38

[Cla97] D. Clark. *Compact Pat trees*. PhD thesis, University of Waterloo, Department of Computer Science, 1997. 31, 48

[CN10] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *Proceedings of the 9th international conference on Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 94–105. Springer Verlag, 2010. 28, 47, 55, 56, 61, 63, 78

[CR03] M. Crochemore and W. Rytter. *Jewels of Stringology: text algorithms*. World Scientific Publishing Co, 2003. 2

[Far97] M. Farach. Optimal suffix tree construction with large alphabets. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 137–143. IEEE, IEEE Computer Society, 1997. 12, 28

[Fis11] J. Fischer. Inducing the lcp-array. In *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer Verlag, 2011. 78

[FM05] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the Association for Computing Machinery*, 52(4):552–581, 2005. 7, 9, 28, 30, 34, 39, 40

[FMN08] J. Fischer, V. Mäkinen, and G. Navarro. An (other) entropy-bounded compressed suffix tree. In *Combinatorial Pattern Matching*, volume 5029 of *Lecture Notes in Computer Science*, pages 152–165. Springer Verlag, 2008. 28, 43

[GGMN05] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05), 2005. 47, 61, 77

[Gus97] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, 1997. 4, 8, 12, 14, 15, 46

[GV05] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005. 28, 31, 32, 34, 77

[HLS+04] W.K. Hon, T.W. Lam, W.K. Sung, W.L. Tse, C.K. Wong, and S.M. Yiu. Practical aspects of compressed suffix arrays and FM-index in searching DNA sequences. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments*, pages 31–38. SIAM, 2004. 34, 77

[HLS+07] W.K. Hon, T.W. Lam, K. Sadakane, W.K. Sung, and S.M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007. 30, 33, 34, 47, 49, 52, 53, 55, 61, 74, 77

[HOK+09] S. Hoffmann, C. Otto, S. Kurtz, C.M. Sharma, P. Khaitovich, J. Vogel, P.F. Stadler, and J. Hackermüller. Fast mapping of short sequences with mismatches, insertions and deletions using index structures. *PLoS computational biology*, 5(9):e1000502, 2009. 4, 63

[J. 97] J. Setubal and J. Meidanis. *Introduction to computational molecular biology*. PWS Pub., 1997. 4

[KLA+01] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In Amihood Amir and Gad M. Landau, editors, *Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer Verlag, 2001. 20, 23, 47, 54, 55, 61, 78

[KMJP77] D.E. Knuth, J.H. Morris Jr, and V.R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977. 2

[KMP09] J. Kärkkäinen, G. Manzini, and S. Puglisi. Permuted longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching*, volume 5577 of *Combinatorial Pattern Matching*, pages 181–192. Springer, Springer Verlag, 2009. 78

[KSB06] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the Association for Computing Machinery*, 53(6):918–936, 2006. 18

[Kur99] S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999. 8, 15

[LD09] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. 4, 63

[LS07] N. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007. 50

[LTPS09] B. Langmead, C. Trapnell, M. Pop, and S.L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009. 4, 63

[McC76] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, 1976. 12, 13, 28

[MM90] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327. Society for Industrial and Applied Mathematics, Society for Industrial and Applied Mathematics, 1990. 8, 15, 18, 19, 28, 50

[MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12:40–66, 2005. 77

[Mor07] Y. Mori. Libdivsufsort: a lightweight suffix sorting library. Disponível em: `https://code.google.com/p/libdivsufsort/`, 2007. 9, 61, 78

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2–66, 2007. 1, 6

[NP12] G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Symposium on Experimental Algorithms*, volume 7276 of *Lecture Notes in Computer Science*, pages 295–306. Springer Verlag, 2012. 77

[OFG10] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Proceedings of the 17th international conference on String processing and information retrieval*, volume 6393 of *Lecture Notes in Computer Science*, pages 309–321. Springer Verlag, 2010. 28, 78

[Pag02] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2002. 31

[Sad02] K. Sadakane. Succinct representations of LCP information and improvements in the compressed suffix arrays. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 225–232. Society for Industrial and Applied Mathematics, Society for Industrial and Applied Mathematics, 2002. 28, 41, 47, 54, 61, 78

[Sad07] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007. 28, 61

[SAR12] D. N. N. Saad and M. Ayala-Rincón. A compressed suffix array based index with succinct longest common prefix information. In *Digital Proceedings of the 7th Brazilian Symposium on Bioinformatics*. Brazilian Symposium on Bioinformatics, 2012. 55

[TGH+02] J.D. Thompson, T. Gibson, D.G. Higgins, et al. Multiple sequence alignment using ClustalW and ClustalX. *Current protocols in bioinformatics*, pages 2–3, 2002. 5

[Tho68] K. Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968. 4

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. 12, 28

[V. 13] V. Mäkinen, N. Välimäki, J. Siren and S. Kazi. Site do grupo SuDS. Disponível em: `http://www.cs.helsinki.fi/group/suds/cst/`, 2013. 9, 61, 78

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, IEEE Computer Society, 1973. 12, 28