Monografia do Projeto de Formatura
O problema do Menor Ancestral Comum: Revisão, Melhorias e Implementação

Apoio: FAPESP **Processo: 05/50796-0**

Aluno: Daniel Ribeiro
Orientador: Alair Pereira do Lago

Sumário

Capítulo 1

Resumo

Algoritmos sobre textos é uma classe muito ampla de algoritmos com aplicações de grande relevância para diversas áreas como Biologia Molecular Computacional e o desenvolvimento de ferramentas de buscas na Internet. Devido ao volume gigantesco de dados envolvidos, soluções lineares são muito procuradas, pois podem tornar solúvel um problema antes intratável.

Duas das estruturas de dados mais avançadas que permitem algoritmos extremamente eficientes são as Árvores dos Sufixos e as Tabelas que permitem a resolução do problema do Menor Ancestral Comum em tempo constante após um pré-processamento linear.

O projeto se concentra em estudar o problema do Menor Ancestral Comum, algumas de suas soluções algorítmicas, propor também melhorias (tanto no quesito de utilização de memória quanto no de tempo de processamento) a estas soluções, formalizar e escrever estas melhorias, implementá-las e realizar *benchmarks*. Ademais, deseja-se estudar algumas de suas aplicações, em particular, envolvendo árvores dos sufixos.

Esse projeto está inserido no Projeto *Fundamentos da Ciência da Computação: Algoritmos Combinatórios e Estruturas Discretas* – Projeto Temático ProNEx - FAPESP/CNPq Proc. No. 2003/09925-5. A. P. do Lago é um dos pesquisadores desse projeto. (Veja em `http://pronex-focos.incubadora.fapesp.br/portal`)

Capítulo 2

Introdução

A Combinatória das Palavras [Lot83] tem ganhado bastante importância recentemente com o desenvolvimento de algoritmos [CR94] que possibilitem a análise, processamento e extração de informações em seqüências cada vez mais compridas, sejam elas textos disponíveis na internet [cit, goo], bibliotecas digitais [Les97, NMWP99] ou seqüências genômicas [Gus97]. Face às dimensões cada vez maiores destas seqüências, algoritmos eficientes (lineares e até sublineares) têm sido cada vez mais requisitados.

Diversas formas de comparações de seqüências acabam por recorrer à solução de problemas como alinhamento de seqüências ou obtenção de uma subseqüência mais comprida que é comum às seqüências comparadas. Estes algoritmos são a grosso modo quadráticos, o que impossibilita sua aplicação a seqüências muito compridas, como as do tamanho de um genoma completo. Por conta disto, no campo da biologia computacional por exemplo, várias heurísticas [AGM$^+$90, TM99, AMS$^+$97, PL88, Pea00] têm sido introduzidas de forma a que se possa eliminar estas limitações.

Nem todo problema admite uma solução eficiente, mas aqueles que admitem acabam por se tornarem centrais. Tanto que novos modelos matemáticos e boas heurísticas para problemas antes intratáveis sempre acabam por recorrer a estes problemas, na busca de soluções eficientes ainda que às vezes aproximadas. Num processo normal de desenvolvimento de algoritmos aplicados à Biologia, propõem-se modelos matemáticos o mais realistas possíveis, para uma posterior implementação dos algoritmos. De fato, muitas vezes a eficiência de técnicas como a construção da árvores dos sufixos [dLS03] e a obtenção do *Menor Ancestral Comum* [dLS03] interferem na elaboração destes modelos de forma que os algoritmos e heurísticas resultantes possam valer-se delas. Este é o caso de diversas implementações de alinhadores de seqüências [DKF$^+$99, BPM$^+$00] bem como ferramentas que se prestam ao estudo das repetições numa seqüência genômica [KCO$^+$01].

De fato, as *árvores dos sufixos* e as tabelas necessárias à resolução do problema do *Menor Ancestral Comum(MAC)* são duas das estruturas de dados mais avançadas e importantes à elab-

oração de algoritmos eficientes que possam ser utilizados no estudo de seqüências extremamente compridas. Recentemente foi escrito [dLS03] um livro que visita novamente os dois problemas e foram também apresentados alguns algoritmos eficientes que constroem e fazem uso eficiente destas estruturas.

Entre os problemas que são resolvidos de forma eficiente pelas duas estruturas de dados estão: encontrar maior fator comum de duas palavras, encontrar todos os palíndromos maximais/repetições encadeadas (ou palavras que são palíndromos/repetições encadeadas a menos de uma certa quantidade de erros) de um texto, buscas de padrões em textos admitindo erros num dos dois ou nos dois, e o de encontrar textos nos quais um padrão aparece ao menos uma vez (que é o mesmo problema dos sites de busca, tais como o Google [goo]).

O problema do Menor Ancestral Comum em árvores tem uma longa história de melhoras. A primeira solução não trivial aparece em 1973 e é publicada em [AHU76]. A primeira solução de consultas em tempo constante após pré-processamento linear é de 1984 e aparece em Harel e Tarjan [HT84]. Estes algoritmos são muito complexos e foram simplificados em 1988 por Schieber e Vishkin em [SV88] e em 2000 por Bender e Farach-Colton em [BFC00]. O problema do Mínimo de um Vetor aparece em 1984, num trabalho de Gabow, Bentley e Tarjan em [GBT84]. Existe uma descrição detalhada da variante de Shieber e Vishkin no livro de Gusfield [Gus97], que é um dos raros casos em que o algoritmo é descrito com alguns detalhes num livro. Existe uma ampla literatura sobre o problema, seus usos e suas generalizações. O problema é discutido nos trabalhos [BPSS01, AGKR02, BFC02].

2.1 Menor Ancestral Comum (MAC)

2.1.1 Grafos e Árvores

Omitiremos as poucas definições e propriedades elementares sobre grafos e árvores que serão necessárias a este projeto. Um leitor poderá encontrá-las[1] na página 4 de [dLS03] ou em qualquer livro de teoria de grafos.

2.1.2 Alfabetos e Palavras

Seja A um *alfabeto finito*. Qualquer seqüência finita de letras de A é também chamada de *palavra em* A, ou simplesmente *palavra* se o alfabeto for claro. Outros sinônimos muitas vezes encontrados para as palavras incluem cadeias, strings e seqüências (o termo palavra será preferido ao longo do projeto). O *comprimento* de uma palavra w é o comprimento da seqüência finita e é denotado por $|w|$. O conjunto das palavras com letras em A (inclusive a palavra vazia[2] 1) é denotado por A^*. A *concatenação* de duas palavras u e v é definida de forma natural e é

[1] http://palavras.incubadora.fapesp.br/portal/livro/livro.pdf

[2] O símbolo λ também é comumente usado para representar a palavra vazia.

denotada por $u \cdot v$, ou simplesmente uv. Observe que $|uv| = |u| + |v|$. O elemento neutro da concatenação é a palavra vazia 1. Dado um inteiro qualquer $k \geq 0$ e uma palavra w, definimos w^k a potência de w a k como sendo a concatenação de k cópias de w. Dizemos $x \in A^*$ é *prefixo*, *sufixo* ou *fator* de $w \in A^*$ se $w \in xA^*$, $w \in A^*x$, ou $w \in A^*xA^*$, respectivamente. Dado $0 \leq k \leq |w|$, observe que há um único prefixo de w de comprimento k e um único sufixo de w de comprimento k.

2.1.3 Árvores dos Sufixos

Dada uma árvore rotulada por palavras nas arestas, associamos a um vértice v a palavra obtida pela concatenação seqüencial dos rótulos das arestas pertencentes ao único caminho que vai da raiz até o vértice v. Dada uma palavra w, sua *árvore dos sufixos* é a árvore com menor número de vértices onde as arestas são rotuladas com palavras não vazias e as seguintes propriedades são satisfeitas:

- palavras que rotulam arestas distintas que partem de um mesmo vértice v começam com letras distintas;
- existe um conjunto de vértices, ditos *finais*, tais que o conjunto das palavras associadas a eles é o conjunto dos sufixos de w.

Uma propriedade fundamental da árvore dos sufixos de uma palavra w é que ela é uma estrutura compacta capaz de armazenar todos os fatores de w e que pode ser consultada de forma eficiente.

2.1.4 O problema do MAC

Dado um vértice v de uma árvore $\mathcal{T}$ de raiz r, existe um único caminho de r até v. Qualquer vértice neste caminho será dito um *ancestral* de v. Dados dois vértices em $\mathcal{T}$, seu *menor ancestral comum*[3](MAC) é o ancestral de ambos mais distante da raiz. Usualmente, deseja-se pré-processar eficientemente $\mathcal{T}$ de forma que qualquer consulta sobre o MAC de dois vértices quaisquer em $\mathcal{T}$ possa ser resolvida em tempo constante.

Em princípio, não é necessário que a árvore sobre a qual se resolve o problema do MAC seja uma árvore dos sufixos, contudo esta é uma aplicação bastante útil e freqüente no estudo de seqüências.

[3]Em inglês: Least(ou Lower) Common Ancestral: LCA.

Capítulo 3

Justificativa

O projeto tem como objetivo estudar vários tópicos, que, apesar de sua relevância para muitos problemas, não são abordados pelas matérias oferecidas tanto à graduação quanto à pós em Ciência da Computação no IME. O assunto é relevante o suficiente para ser visto em cursos de doutorado em universidades no exterior, como é o caso do MIT. Os algoritmos a serem estudados são aplicações eficientes do MAC, resolvido em tempo constante depois de um pré-processamento linear sobre Árvores de Sufixos, buscando soluções de preferência lineares para os problemas descritos abaixo.

3.1 Aplicações do MAC sobre Árvores dos Sufixos

3.1.1 Problema da extensão comum mais longa

São dadas as palavras s e t, de comprimentos m e n respectivamente, e uma seqüência de pares de posições $(i_k, j_k) \in \{1, \ldots, m\} \times \{1, \ldots, n\}$. Para cada um dos pares (i_k, j_k) queremos calcular o comprimento do fator comum mais longo de s e de t que ocorre na posição i_k em s e j_k em t.

3.1.2 Palíndromos maximais

Seja f a função inversão de uma palavra, ou seja, se w é uma palavra de comprimento m a i-ésima letra de $f(w)$ é a $m - i + 1$-ésima letra de w. Uma palavra w é dita *palíndromo* se $w = f(w)$. Um fator $w[i..j]$ de uma palavra w é um *palíndromo maximal em* w se $w[i..j]$ for palíndromo e $w[i-1..j+1]$ não é definida ou não é palíndromo. O problema consiste em encontrar todos os palíndromos maximais de uma palavra w.

3.1.3 Busca de padrão exata com coringas

Uma busca de padrão exata com coringas de um padrão p num texto t pode ser feita admitindo coringas em p e/ou t. Cada ocorrência do coringa pode ser vista como uma letra adicional ao alfabeto que admite um casamento perfeito com qualquer letra do alfabeto durante a comparação das letras de p com as de t. Este problema tem uma solução linear ao utilizar uma solução eficiente para o problema do MAC.

3.1.4 Busca de padrão com erros

São dados uma palavra s de comprimento m, também chamada padrão, uma outra palavra t de comprimento n, também chamada texto e um inteiro $k \geq 0$ fixo. Queremos saber se existe fator de t com o mesmo comprimento de s e cujas letras diferem das de s em no máximo k posições. Tipicamente, k é muito menor que m, que por sua vez é menor que n, i.e.,

$$k \ll m < n.$$

3.1.5 Palíndromos aproximados e repetições

Dado um inteiro $k > 0$, um *palíndromo com k erros* é uma palavra s tal que exista um palíndromo t, com o mesmo comprimento de s, e cujas letras diferem das de s em no máximo k posições.

Uma *repetição encadeada*[1] é uma palavra v tal que existam um inteiro $k > 1$ e um prefixo w de v tais que w^k é prefixo de v, que por sua vez é prefixo de w^{k+1}.

Dado um inteiro $k > 0$, uma *repetição encadeada com k erros*, é uma palavra s tal que exista uma repetição encadeada t, com o mesmo comprimento de s, e cujas letras diferem das de s em no máximo k posições.

O problema em questão é o de detectar todos os palíndromos com k erros, todas as repetições, e todas repetições com k erros numa palavra. Este problema pode ser resolvido em tempo polinomial como uma aplicação do MAC sobre árvores dos sufixos.

3.1.6 k-fator comum

O problema do k-fator comum pode ser resolvido em tempo linear ao utilizar o MAC. O problema consiste em, dadas K palavras *distintas*, determinar, para todo $k: \ 2 \leq k \leq K$ o maior fator comum em pelo menos k das K palavras.

[1] Tradução de *tandem repeat*.

3.1.7 Busca de Documentos (Document Retrival)

Dados k textos, queremos pré-processá-los de modo a, dado um padrão, sermos capazes de determinar rapidamente em quais destes textos o padrão aparece ao menos uma vez. Este problema é resolvido por um site de buscas na Internet como o Google [goo].

Capítulo 4

Objetivos

O objetivo principal do projeto é estudar o problema do Menor Ancestral Comum e algumas soluções já propostas, bem como propor melhorias a estas soluções que levem a uma solução mais eficiente. Para tanto, serão feitas comparações com estas outras soluções, tanto teoricamente quanto através de *benchmarks*. Uma implementação também será necessária, para tanto.

Também se pretende estudar, e eventualmente implementar, algumas das aplicações do MAC discutidas anteriormente.

Sendo-se feitas melhorias em relação aos algoritmos apresentados no livro [dLS03], como o mesmo se encontra disponível[1] para alterações cooperativas no projeto da incubadora FAPESP, pretende-se propor alterações ao livro de forma a incluir estas eventuais melhorias. Eventualmente, considerar-se-á a possibilidade de que seja escrito algum relatório técnico e/ou artigo.

O aluno considera a hipótese de dar continuidade a este trabalho numa pós-graduação.

[1] http://palavras.incubadora.fapesp.br/portal/livro/livro.pdf

Capítulo 5

Síntese da Bibliografia fundamental

- do Lago e Simon [dLS03]: até onde saibamos, este é primeiro livro em português que descreve uma construção em tempo linear das árvores dos sufixos (desenvolvendo a solução de McCreight) bem como uma solução em tempo constante do problema do Menor Ancestral Comum (desenvolvendo e melhorando uma solução de Bender e Farach-Colton [BFC00]);

- Gusfield [Gus97]: este é um livro de Biologia Computacional, que trata de vários problemas na área, enfocando reconhecimento exato e inexato de padrões de um texto. Inclui capítulos específicos para tratar de algoritmos de construção de árvores dos sufixos (baseado em artigo de Ukkonen), e uma solução em tempo constante do Menor Ancestral Comum após um pré-processamento linear (baseado em artigo de Schieber e Vishkin [SV88]) a apresenta inúmeras aplicações à Biologia Computacional;

- Bender e Farach-Colton [BFC00]: este é um artigo que propõe uma versão seqüencial obtida a partir de uma simplificação de uma solução paralela [BGSV89] para o problema do Menor Ancestral Comum e que possui a grande virtude de ser uma solução bem mais simples se comparada à solução original de Harel de Tarjan [HT84] ou mesmo à solução já mais simples de Schieber e Vishkin [SV88];

- Muthukrishan'02 [Mut02]: este é um artigo com uma solução em tempo ótimo (linear no tamanho da saída) para o problema de Busca de Documentos (Document Retrival) após um pré-processamento linear.

- Michael A. Bender [BPSS01]: artigo que trata do problema do Menor Ancestral comum em Grafos Dirigidos Acíclicos (DAGs), comentando algumas de suas aplicações mais relevantes. O artigo mostra um algoritmo $\Omega(n^3)$ para preprocessar um DAG de modo a responder, em tempo constante, qual é o menor ancestral comum de dois vértices do DAG. Além disso o artigo faz estudos de performance, além de demonstrações acerca da dificuldade computacional do problema.

Capítulo 6

Resumo das Atividades Realizadas

1. Dos artigos presentes na síntese bibliográfica do projeto original, consideramos nesta fase aqueles que tratam de soluções para o problema do MAC: [BFC00], [dLS03] e [SV88]. As resenhas enfocaram as soluções para o problema do Menor Ancestral Comum presentes nos trabalhos, e uma análise crítica do consumo de espaço. O artigo de Schieber e Vishkin [SV88] foi escolhido ao invés do capítulo 8 do Gusfield [Gus97] pois durante o estudo achamos que o primeiro é mais claro, além de ser o artigo original.

2. O algoritmo descrito por Lago e Simon [dLS03] foi implementado. Foram estudadas e aplicadas as seguinte melhorias:

   (a) eliminação das tabelas de logaritmos e exponencial, com ganho constante de eficiência em casos de aplicação reais;

   (b) redução de uma tabela de tamanho $O(n)$ à sua metade;

   (c) diminuição da memória auxiliar utilizada no pré-processamento.

3. Foram realizados vários testes com árvores aleatórias e com árvores de sufixos geradas a partir de textos reais. Os testes incluíram a comparação com um algoritmo de tempo de consulta de ordem da altura da árvore.

4. O algoritmo descrito em [BFC00] foi implementado, e comparado com a implentação descrita por Lago e Simon [dLS03], e com uma implementação do algoritmo descrito em [SV88], da coleção de algoritmos Strmat[1].

[1] http://www.cs.ucdavis.edu/ gusfield/strmat.html

Capítulo 7

Detalhamento dos Progressos Realizados

7.1 Resenhas: Visão Geral

Este projeto parte do capítulo 4 do livro Lago e Simon [dLS03], que por sua vez desenvolve o trabalho de Bender e Farach-Colton [BFC00]. Suas resenhas se encontram a seguir.

Este trabalho, por sua vez, é por sua vez uma simplificações do algoritmo paralelo presente no trabalho de Berkman et al. [BGSV89], que não faz parte do conjunto de artigos estudados por ser paralelo.

Apresentamos um pequeno resumo de Schieber e Vishkin [SV88]. Por causa da complexidade dos seus detalhes, a versão integral da resenha deste artigo pode ser encontrada no apêndice.

É relevante notar que embora o artigo do Schieber e Vishkin seja baseado no de Harel e Tarjan [HT84], este não foi selecionado para estudo por ser sensivelmente mais complexo do que aquele, apesar de ter a importância histórica de ter sido o primeiro artigo apresentando uma solução com consultas em tempo constante após um pré-processamento linear.

7.2 Schieber e Vishkin

O algoritmo se baseia fortemente em dois fatos:

- Se a árvore sobre a qual se deseja responder o MAC é um caminho, então a profundidade de cada vértice, obtível com tempo e memória lineares, é suficiente para resolver o MAC.
- Se a árvore fosse uma árvore binária completa, uma enumeração dada por uma busca inordem na árvore é suficiente para resolver consultas do MAC em tempo constante.

  Para os detalhes do algoritmo, veja o apêndice do relatório.

7.3 Bender e Farach-Colton

7.3.1 Problema Range Minimum Query (RMQ)

Definição

O problema do Range Minimum Query (RMQ), recebe um vetor de números $A[1, \ldots, n]$ como entrada. Então, para índices i e j entre 1 e n, quer-se saber qual é o elemento de menor valor no subvetor $A[1, \ldots, n]$.

Uma versão mais restrita para o problema (chamado de RMQ$\pm$1) é aquela onde exigimos que A seja um vetor de inteiros, e todos os elementos adjacentes diferirem entre si de 1 (esta sendo a *restrição* ± 1), temos o problema RMQ$\pm$1.

RMQ e MAC

Ao longo do texto, T indica uma árvore enraizada dada para o problema do MAC, e que possui n vértices.

Lema 1 *$MAC(u, v)$ é o vértice menos profundo encontrado nas visitas de u e v em uma busca em profundidade em T.*

O problema do MAC se reduz ao problema RMQ$\pm$1 da seguinte forma:

Pré-processamento:

1. Seja E um vetor que contém todos os vértices que são visitados por um passeio Euleriano em T, começando pela raiz. Então E possui $m := 2n - 1$ elementos, e $E[i]$ é o símbolo que representa o i-ésimo nó visitado no passeio Euleriano.

2. Seja a *profundidade* de um nó, a sua distância à raíz. Computa-se o vetor de profundidades $L[1, \ldots, n]$

3. Seja o *representante* de um nó no passeio Euleriano o índice da primeira ocorrência do nó no passeio. Obtém-se um vetor $R[1, \ldots, n]$ tal que $R[i]$ é o índice do representante do nó i.

Portanto, para se obter o $MAC(u, v)$, basta obter $E[_{RMQ_L}(R[u], R[v])$, segundo o lema 1. NOTA: RMQ_L é o RMQ resolvido para o vetor L.

7.3.2 Resolução Eficiente do RMQ$\pm$1

Apresentamos então a resolução em tempo $O(m)$ para um pré-processamento, e tempo $O(1)$ para as consultas. Além disto, toda a computação dispõe de memória linear.

Supomos que temos um vetor de inteiros $A[1,\ldots,m]$ com a restrição ± 1. Denotamos por $\lg(x)$ por $\log_2^x$. Particionamos o vetor A em blocos de tamanho $\lg(m)/2$ (a menos do último bloco, que pode ter menos). Definimos o vetor $A'[1,\ldots,2m/\lg(m)]$, onde $A'[i]$ é o valor mínimo do i-ésimo bloco. Definimos o vetor $B'[1,\ldots,2m/\lg(m)]$ tal que $B[i]$ é a menor posição dentro do i-ésimo bloco que tal valor ocorre.

Para responder o MAC dentro dos blocos, introduzimos o conceito de vetor normalizado: *Um vetor é normalizado se seu primeiro elemento é 0.*

Lema 2 *Existem* $2^{(\lg(m)/2)-1} = O(\sqrt{m})$ *tipos de blocos normalizados que satisfazem à restrição* ± 1.

Lema 3 *Se dois vetores que satisfazem à restrição* ± 1 *diferem de uma constante* c *em cada posição, então a posição das consultas RMQ nos dois é a mesma, apenas diferindo o valor absoluto do mínimo.*

Então são pré-computadas $O(\sqrt{m})$ tabelas, uma para cada bloco normalizado possível. Cada tabela possui todas as $(\lg(m)/2)^2 = O(\lg^2(m))$ respostas de consultas RMQ possíveis naquele bloco. Todas as tabelas são trivialmente computadas em tempo $O(\sqrt{m}\lg^2(m)) = O(m)$, e gastam o mesmo espaço. O lema 3 garante que podemos tratar apenas de blocos normalizados.

Dados os mínimos dos blocos, obtém uma matriz M, onde $M[i,j] = argmin_{k=i\ldots i+2^j-1}A'[k]$. M é computada e ocupa espaço $O(\frac{2m}{\lg(m)}\lg(\frac{2m}{\lg(m)})) = O(m)$, usando programação dinâmica.

Feito todo o pré-processamento, respondemos ao $RMQ(i,j)$ da seguinte forma:

1. Obtém-se das tabelas o mínimo a partir de i até o fim do seu bloco.
2. Computa-se o mínimo de todos os blocos entre os de i e de j
3. Obtém-se o mínimo a partir começo do bloco de j até j.

Que, dadas as estruturas obtidas no pré-processamento, é feito em tempo $O(1)$

7.3.3 Consumo de Espaço

Uma vez feito o pré-processamento, o algoritmo de obtenção do MAC precisa armazenar apenas as seguintes estruturas, com os respectivos tamanhos de seus elementos, e com suas dimensões, lembrando que $m := 2n-1$.

Estrutura	Dimensão	Tamanho dos Elementos
Vetor dos Vértices do passeio Euleriano E	m	$\lg(m)$
Vetor de Profundidades L	m	$\lg(m)$
Vetor de Representantes R	n	$\lg(m)$
Tabelas dos mínimos dos blocos normalizados	$2^{(\lg(m)/2)-1}(\frac{\lg(m)}{2})^2$	$\lg(\lg(m))$
Matriz de mínimo entre blocos M	$\frac{2m}{\lg(m)}\lg(\frac{2m}{\lg(m)})$	$\lg(m)$
Vetor que relaciona blocos com tabelas	$\frac{2m}{\lg(m)}$	$(\lg(m)/2)-1$

Nota 1 *O **vetor que relaciona blocos com tabelas** não é mencionado no artigo de Bender e Farach-Colton, mas é necessário relacionar um bloco do Vetor V com a respectiva tabela de mínimos internos. Como são $2^{(\lg(m)/2)-1}$ tabelas, o logaritmo deste valor é necessário para indexá-las.*

Uma implementação real que trate de valores de $n < 2^{31}$, e portanto $m < 2^{32}$, e que trate com unidades de 8 bits, precisa de 4 bytes para armazenar m e de 1 byte para armazenar $\lg(m)$, e portanto tem um consumo de espaço *por vértice de* T, quando m é aproximadamente 2^{32}, e por estrutura:

Estrutura	Aproximação do Consumo de espaço em bytes por Vértice
Vetor dos Vértices do passeio Euleriano E	8
Vetor de Profundidades L	8
Vetor de Representantes R	4
Tabelas dos mínimos dos blocos normalizados	0.2
Matriz de mínimo entre blocos M	14.55
Vetor que relaciona blocos com tabelas	0.27

Logo o custo total por vértice numa implementação real, sob tais hipóteses, é de aproximadamente 35 bytes.

7.4 Lago e Simon

7.4.1 Introdução

A algoritmo de resolução do MAC proposto em [dLS03] utiliza a mesma estratégia de redução ao problema RMQ±1 presente em [BFC00]. As principais diferenças entre os trabalhos encontram-se na resolução das consultas internas aos blocos e parte do pré-processamento do RMQ±1.

Definição 1 *Um vetor de inteiros é denominado* contínuo *se ele possui a* restrição ± 1.

Portanto podemos chamar o problema RMQ$\pm$1 de *Mínimo de um Vetor Contínuo(MVC).*

Definição 2 *Um vetor de inteiros é dito normalizado se seu primeiro elemento é zero.*

7.4.2 Resolução Eficiente do MVC

Apresentamos então a resolução que gasta tempo $O(m)$ para um pré-processamento, e tempo $O(1)$ para as consultas. Além disto, toda a computação dispõe de memória linear.

Definição 3 *O* ***Vetor de Diferenças*** V' *de um vetor contínuo* $V[1, \ldots, m]$ *é vazio se* $m = 0$*, ou é definido da seguinte forma:* $V'[i] = V[i+1] - V[i]$*, para todo* i *tal que* $1 \leq i < m$*. Note que como* V *é um vetor contínuo, os elementos de* V' *são* 1 *ou* -1*.*

Definição 4 *Seja* V *um vetor contínuo e* V' *seu vetor de diferenças. Seja* $p(V)$ *a palavra em* $\{-1, 1\}^*$ *formada pela concatenção dos elementos de* V' *(definimos* p *como a palavra vazia* λ *caso* V' *seja vazio). Então definimos* $f(V)$ *como a palavra em* $\{0, 1\}^*$ *formada substituindo as letras* -1 *por* 0 *em* $p(V)$*.*

A definição acima é equivalente a dizer que:

$$f(V[1,\ldots,m]) = \begin{cases} \lambda & \text{se } m = 1 \\ f(V[1,\ldots,m-1]) \cdot 0 & \text{se } m > 1 \text{ e } V[m] = V[m-1] - 1 \\ f(V[1,\ldots,m-1]) \cdot 1 & \text{se } m > 1 \text{ e } V[m] = V[m-1] + 1 \end{cases}$$

Observação: A representação binária com k bits, $k \geq m$, de $f(V)$ não é única, pois ela inclui zeros mais significativos.

Exemplo: Sejam $V = [0, -1, -2, -1]$, e $S = [0, -1, -2, -1]$

$f(V) = 001$

$f(S) = 01$

Contudo, as reprsentações com 3 bits tanto de $f(V)$ quanto de $f(S)$ são iguais a 001. É importante notar que não haveria falta de unicidade se todos os vetores contínuos com que trabalhássemos tivessem o mesmo comprimento.

Introduzindo unicidade: Para eliminar o problema de falta de unicidade na representação binária, representamos $1 \cdot f(V)$ ao invés de $f(V)$. Denominamos **Cracha**(**V**) o número cuja representação binária é $1 \cdot f(V)$.

Observação 1 *Se dois vetores contínuos de mesmo tamanho diferem de uma constante elemento a elemento, então ambos possuem o mesmo crachá. Contudo, dado um crachá* c*, só existe um vetor contínuo normalizado que possui* c *como crachá.*

Pré-processamento:

Os passos do pré-processamento que diferem dos citados em [BFC00] são:

1. O vetor V é particionado em blocos de tamanho[1] $b := \lg(m)$ e não $\frac{\lg(m)}{2}$. Esta alteração não altera a complexidade deste algoritmo devido à técnica dos russos. Visando garantir a linearidade, b não pode exceder ou ser igual a $\lg(m)/2$.

2. Para responder as consultas do MVC dentro de um bloco, não utilizamos as $O(\sqrt{m})$ tabelas, mas sim a técnica dos quatro russos. É então utilizado um vetor $RUSSO$, tal que $RUSSO[i]$ é a posição do menor elemento do vetor contínuo normalizado que possui i como crachá, para i entre 1 e $2^{b-1} = \lg(m)$, onde b é o tamanho dos blocos de V.

3. Existe um vetor auxiliar que identifica cada bloco com o seu crachá.

4. São montadas tabelas com as pré-computações dos logaritmos na base dois de todos os números entre 1 e m. Também são pré-computados os valores de 2^k, para k entre 0 e $\lfloor \log_2^k \rfloor$. Em [BFC00] estas pré-computações não são descritas, apenas consideradas possíveis e desacopladas do algoritmo de resolução do problema do MVC.

Consultas MVC dentro de blocos:

As consultas se baseiam no fato de que, dado o crachá c de um vetor contínuo normalizado v, podemos obter em tempo $O(1)$ os crachás dos subvetores contínuos normalizados (ou possivelmente vazios) x e y, sendo que $v = x \cdot y$. Os passos da consulta $MVC(i, j)$são:

1. Sejam g_i e g_j os blocos de i e j respectivamente. Devolva erro se $g_j \neq g_i$, ou se $i > j$.

2. Seja p_i a posição relativa do elemento i dentro do bloco.

3. Seja c_1 o crachá do subvetor $x_1 := g_i[p_i \ldots \lg(m)]$.

4. Seja c_2 o crachá do subvetor $x_2 := x_1[1 \ldots j - i + 1]$.

5. Devolva $i - 1 + RUSSO[c_2]$.

7.4.3 Consumo de Espaço

Uma vez feito o pré-processamento, o algoritmo de obtenção do MAC precisa armazenar apenas as seguintes estruturas, com os respectivos tamanhos de seus elementos, e com suas dimensões, tomando $m := 2n - 1$.

[1] $\lg(m)$ é notação para $\log_2^m$

Estrutura	Dimensão	Tamanho dos Elementos
Vetor dos Vértices do passeio Euleriano	m	$\lg(m)$
Vetor de Profundidades	m	$\lg(m)$
Vetor de Representantes	n	$\lg(m)$
Vetor dos Russos	m	$\lg(\lg(m))$
Vetor dos Crachás	$\frac{m}{\lg(m)}$	$\lg(m)$
Matriz de mínimo entre blocos	$\frac{m}{\lg(m)} \lg(\frac{m}{\lg(m)})$	$\lg(m)$

Nota 2 *Não consideramos as tabelas de logaritmos e exponenciais pois os algoritmos descritos em [BFC00] e [SV88] não fazem tal consideração, e pois ambas operações podem ser executadas em tempo constante em números de comprimento* $\lg(m)$.

Uma implementação real que trate de valores de $n < 2^{32}$, e portanto $m < 2^{31}$, e que trate com unidades de 8 bits, precisa de 4 bytes para armazenar m e de 1 byte para armazenar $\lg(m)$, e portanto tem um consumo de espaço *por vértice de* T, quando m é aproximadamente 2^{32}, e por estrutura:

Estrutura	Aproximação do Consumo de espaço em bytes por Vértice
Vetor dos Vértices do passeio Euleriano	8
Vetor de Profundidades	8
Vetor de Representantes	4
Vetor dos Russos	2
Vetor dos Crachás	0.27
Matriz de mínimo entre blocos	6.96

Logo o custo total por vértice numa implementação real, sob tais hipóteses, é de aproximadamente 30 bytes.

7.5 Melhorias

Aqui detalhamos as melhorias feitas em relação ao algoritmo descrito por Lago e Simon [dLS03].

7.5.1 Eliminação das Tabelas de Logaritmos e Exponenciais

Os vetores que possuem logaritmos e exponenciais, que são usados nas consultas do MVC são vetores cuja quantidade de elementos depende de n. Contudo, os exponenciais podem ser feitos em uma instrução de máquina utilizando a operação de bits *SHIFTLEFT*.

A obtenção dos logaritmos na base dois podem ser feitos também em tempo constante, utilizando *Mascaras de bits* que sempre dividem os bits ao meio. Para números de até 2^{32} bits, que foi o enfoque da implementação, são necessárias apenas $\lg(\lg(2^{32})) = 5$ operações de máscaras e shifts, utilizando esta técnica.

Contudo, segundo experimentos, esta técnica é cerca de *duas* vezes menos eficiente, se a tabela de logaritmos consegue ficar no cache da máquina. Como o algoritmo implementado utiliza muitas outras estruturas mais volumosas e utilizadas mais freqüentemente, esta suposição de que a tabela de logaritmos permaneça no cache não é realista. Segundo os testes, quando a tabela de logaritmos é grande demais para permanecer no cache, a utilização da tabela é cerca de *sete* vezes menos eficiente do que o cálculo do logaritmo por máscaras e shifts.

Seguem os resultados obtidos numa máquina $1466.755MHz$ de clock, e $256KB$ de cache (foram utilizados amostras com 3 milhões de números):

Tamanho do Tabela de Logaritmos	Tempo da Computação utilizando a Tabela (segundos)	Tempo da Computação por Shifts e Máscaras (segundos)
10	0.090000	0.130000
50	0.090000	0.130000
100	0.090000	0.140000
500	0.090000	0.150000
1000	0.090000	0.140000
5000	0.090000	0.140000
10000	0.100000	0.140000
50000	0.100000	0.140000
100000	0.090000	0.140000
500000	0.280000	0.150000
1000000	0.440000	0.130000
5000000	0.170000	0.140000
10000000	0.110000	0.140000

Portanto, podemos sem temer perda de eficiência, eliminar tais tabelas do algoritmo proposto em [dLS03].

7.5.2 Crachás de Vetores de Comprimento Fixo

Seja f como definida na resenha de Lago e Simon. Baseado no fato de que f não perde unicidade se todos os vetores com que trabalharmos possuem o mesmo número de elementos, definimos uma nova identificação dos vetores contínuos normalizados:

Definição 1 *O crachá β (denotado por* Cracha_β*) de um vetor contínuo normalizado V é o número cuja representação binária é $f(V)$.*

Observação 1 $\forall\ V$, *vetor contínuo,* $\text{Cracha(V)} = 1 \cdot \text{Cracha}_\beta\text{(V)}$, *quando vistos como palavras em* $\{0,1\}^*$.

Ao nos restringirmos a trabalhar *apenas* como crachás de contínuos normalizados de comprimento igual a b (que é o do tamanho do blocos que particionam o vetor contínuo sobre o qual se deseja fazer consultas MVC), podemos utilizar a representação binária com k bits, k maior ou igual ao tamanho dos blocos menos 1 sem perder unicidade.

Lema 1 *Dado um vetor contínuo normalizado V tal que:*

$\text{Cracha}_\beta\text{(V)} = \text{c} \in \{0,1\}^*,\ |\text{c}| = \text{m} - \text{b} + 1$

Seja w um fator de c. Então é possível em tempo $O(1)$*, obter a posição do menor elemento num vetor contínuo normalizado S, tal que* $\text{Cracha(S)} = 1 \cdot \text{w}$

Este lema implica que podemos obter em tempo $O(1)$ o MVC interno a um bloco (que é um vetor contínuo) a partir de seu Cracha_β. Logo a complexidade das consultas MVC não é afetada.

Logo, basta guardar a representação binária com $m-b+1$ do crachá β dos vetores contínuos normalizados.

7.5.3 Diminuição da Memória Auxiliar

A utilização de crachás de vetores contínuos normalizados com $\lg(m)$ elementos permite que os vetores auxiliares (que armazenam as posições dos mínimos, os valores destes, e a diferença entre o valor do último elemento do vetor e o primeiro) não precisem armazenar os valores para todos os vetores contínuos normalizados com menos do que $\lg(m)$.

A melhoria, consiste armazenar os valores para vetores contínuos normalizados com menos do que $\lg(m)$ cujos crachás, quando vistos como palavras em $\{0,1\}^*$, são prefixos do crachá do vetor com $\lg(m)$ elementos cuja posição do mínimo queremos obter.

Exemplo: Para obter a posição do mínimo do vetor contínuo cujo crachá é 0010, basta obtermos a posição do mínimo dos vetores contínuos normalizados cujos crachás são: 0, 00 e 001.

Logo o consumo de memória auxiliar para a computação do vetor $RUSSO$ cai de $O(m)$ para $O(\lg(m))$.

7.6 Implementação, testes e experimentos

Todos os algoritmos e testes implementados foram escritos na linguagem *C*. Além da implementação de uma versão melhorada do algoritmo descrito em [dLS03], e do algoritmo descrito em [BFC00]foram implementados:

- um algoritmo *ingênuo* que resolve o MAC após pré-processamento linear e consultas em tempo proporcional à soma dos comprimentos dos caminhos até a raiz;
- um gerador de testes de consultas do MAC para árvores geradas aleatóriamente e para árvores de sufixos geradas a partir de textos variados.

O gerador de árvores aleatórias é um simples gerador que começa com uma árvore com um nó, e a cada passo acrescenta um novo vértice de modo que a probabilidade dele ser filho de cada um dos nós já presentes na árvore é uniforme.

Para obter as árvores de sufixos a partir de textos foi utilizada uma biblioteca de criada por Stefan Kurtz. Contudo foi necessário converter para a representação utilizada pelos algoritmos de testes e de resolução do MAC a representação utilizada pela biblioteca, uma vez estas eram incompatíveis. A biblioteca pode ser encontrada *http:/www.genomics.jhu.edu/MUMmer*, como parte do fonte do programa *MUMmer 3.0*.

Nos testes realizados foram feitas variações do tamanho dos blocos utilizados no pré-processamento do MVC, comparações dos algoritmos descritos em [dLS03], [BFC00] e [SV88], e do algoritmo ingênuo, e experimentações com cálculo de logaritmo (calculado de forma direta comparado com o pré-processado). Os testes foram realizados em várias máquinas diferentes, com grandes diferenças de configurações.

Uma análise inicial comprovou nossa intuição, que o algoritmo descrito em [SV88], por realizar poucas operações bastantes simples e ocupar menos memória (não exigindo tanto da memória cache do sistema), tem um consumo de tempo menor por consulta que os outros algoritmos. Seguindo nossas previsões, o algoritmo ingênuo se mostrou bem menos eficiente para as instâncias maiores, e as consultas utilizando o algoritmo descrito em [BFC00] se mostrou pouco menos eficiente do que o algoritmo descrito em [dLS03] para instâncias maiores devido a sua relativa maior necessidade de memória (embora sua consulta exiga pouco menos cálculo).

7.7 Dificuldades

Uma das grandes dificuldades foi encontradar implementações em C dos algoritmos de Bender e Farach-Colton [BFC00] e de Schieber e Vishkin [SV88], que fossem bem documentadas e de fácil adaptação às nossas necessidades. Estas são importantes para a comparação dos algoritmos em aplicações reais que pretendemos realizar. Ao fim optamos por implementar o primeiro, e encontramos uma implementação adequada do segundo.

Uma das maiores surpresas contudo foi a relevância da influência da memória cache no testes. A utilização do cache se mostrou determinante para garantir que a tabela de logaritmos, devido à pouca freqüência que é utilizada nas consultas do MVC, se mostrasse menos eficiente do que a obtenção direita do logaritmo na base 2. Os testes realizados em máquinas variadas mostraram que a utilização de uma tabela pequena (menos de 1000 entradas) é duas vezes mais eficiente

do que o cálculo direto, enquanto a utilizaçãode tabelas com cerca de $3M$ entradas é cerca de 7 vezes meno eficiente do que o cálculo direto.

Capítulo 8

Parte Subjetiva

8.1 A Pesquisa e o BCC

A iniciação científica que deu origem à esta pesquisa começou em Novembro de 2004 e tem prazo para terminar em Janeiro de 2006. A escolha do tema de pesquisa foi feita baseada na sua proximidade com a área de Grafos, uma matéria que tinha acabado de cursar, e com que senti grande afinidade. No começo eu tinha um pouco de receio de trabalhar com alguns assuntos dos quais eu não tinha nenhum conhecimento (tais como Linguagens Formais e Biologia Computacional), mas depois de ter lido alguns textos introdutórios indicados pelo meu orientador, ganhei confiança em avançar com a pesquisa.

Depois de passado os estudos introdutórios, eu e meu orientador optamos por concorrer a uma bolsa FAPESP, apesar de estarmos cientes que isso tornaria a iniciação bem mais exigente para nós dois (como de fato se mostrou na época da elaboração do relatório parcial).

Durante a pesquisa senti muitas vezes a frustração do fato de que alguns processos para serem adequadamente realizados exigem muito mais tempo do que eu imaginava. Isso aconteceu principalmente na realização das leituras dos artigos e livros, muitas vezes mal escritos, com notações distintas, ou até mesmo incompletos. Também senti bastante dificuldade na confecção das resenhas, que muitas vezes exigiam não só um entendimento mais profundo do texto, como também a confecção de um texto mais claro que o original.

Ao longo da pesquisa, foi interessante perceber como vários dos conceitos aprendidos ao longo de todo o curso eram utilizados, mesmo que sutilmente (seja no estudo de crescimento de funções, entendimento de memória virtual de um sistema operacional, ou em propriedades de Linguagens).

Apesar de não estar inclinado a seguir estudando Biologia Computacional no meu mestrado, e tampouco seguir carreira acadêmica, acredito que a experiência foi bastante proveitosa, pois tive de aprender como pesquisar, trabalhar em projetos de longo prazo, redigir textos científicos claros, e trabalhar com outras pessoas. Também tive de aprender a dar palestras, uma vez que

tive reuniões semanais com meu orientador ao longo de todo o projeto nas quais eu apresentava o que tinha feito ao longo da semana.

8.2 Disciplinas Mais relevantes

MAC0328 Algoritmos em Grafos Pois muitos conceitos envolvendo árvores e buscas em grafos foram utilizadas diretamente ao longo da iniciação, além de ter sido muito importante para a compreensão de outras matérias relevantes para a pesquisa.

MAC0338 Análise de Algoritmos Não só por ter sido fundamental na análise dos algoritmos, como também por ter me ensinado técnicas muito utilizadas nos algoritmos vistos (tais como programação dinâmica).

MAC0414 Linguagens Formais e Autômatos Os conceitos formais de palavras e linguagens são bastante fundamentais para as aplicações dos algoritmos.

MAT0213 Álgebra II Embora não tenha sido utilizada diretamente, essa matéria foi bastante relevante por seu enfoque em demonstrações formais (que foram muitas vezes necessárias para garantir propriedades não demonstradas pelos artigos e livros).

MAC0422 Sistemas Operacionais Pois as implementações são utilizadas em sistemas operacionais, que podem afetar e de fato afetam os testes com suas mudanças de contexto, sua administração de memória, etc

MAC0465 Biologia Computacional As aplicações dos algoritmos vistos utilizam muito estruturas de dados vistas nessa matéria.

Apêndice A

Schieber e Vishkin - Resenha

A.1 Definições e Considerações

1. Seja T uma árvore enraizada. Denotamos por V o conjunto de nós de T, e por E o conjunto de arestas da árvore. Notaremos: $n := |V|$.

2. $PREORDER(v) := i \Leftrightarrow$ o vértice v é o $i-$ésimo vértice visitado por uma busca em pré-ordem em T.

3. $SIZE(v)$ é o número de vértices na subárvore enraizada em v.

4. Todos os números inteiros serão utilizados na sua representação binária.

5. B é a menor árvore binária completa com no mínimo n vértices. Os vértices de B são denotados $V(B)$, e suas arestas $E(B)$. B é isomorfa à uma árvore $(V', E(B))$, para algum $V' \subset \mathbb{Z}$, tal que cada nó de $V(B)$ é identificado com o seu índice fornecido por uma busca em inordem em B. Utilizando tal isomorfismo, identificamos os vértices de B por vértices de V', que são números inteiros.

6. Denotamos $\log_2^x$ por $\lg(x)$.

7. Denotamos por $l := \lg(n)$.

8. Convencionamos que todo vértice x de uma árvore é ancestral e é descendente de x.

A.2 Pré processamento

A.2.1 Passo 1

Para todo v em T computamos $PREORDER(v)$ e $SIZE(v)$. Repare que os valores de $PREORDER$ dos vértices da sub-árvore enraizada em v possuem valores no *intervalo de* v,

sendo que definimos:

intervalo de $v := \{x \in \mathbb{Z} : PREORDER(v) \leq x \leq PREORDER\ + SIZE(v) - 1\}$.

Lema 1: Qualquer classe de equivalência de $INLABEL(INLABEL^{-1}(v))$ induz um caminho em T, de comprimento possivelmente nulo.

Lema 2: Se $INLABEL(v) = x$, que é um vértice de B, todos os descendentes de v possuem um valor de $INLABEL$ que é descendente de x em B.

Executando a computação: É feita com uma busca em profundidade em T.

A.2.2 Passo 2:

Definimos como $INLABEL(v)$ o inteiro que possui a maior quantidade de bits 0 à **direita**, e que está no intervalo de v. Este número está bem definido.

Executando a computação:

Passo 2.1: Seja $i := \lfloor \lg([(PREORDER(v) - 1)\ XOR\ (PREORDER(v) + SIZE(v) - 1)]) \rfloor$. Então i é o índice (começando de zero e contando da **direita**) do bit mais à **esquerda** (e portanto de maior índice) que não coincide em $PREORDER(v) - 1$ e $PREORDER(v) + SIZE(v) - 1$.

Lema 3: Os $l - i + 1$ bits mais à **esquerda** de $INLABEL(v)$ e de $PREORDER(v) + SIZE(v) - 1$ são iguais.

Lema 4: Os i bits mais à **direita** de $INLABEL(v)$ são todos 0.

Passo 2.2: Para obter $INLABEL(v)$ a partir dos lemas acima, computamos:

$INLABEL(v) \leftarrow 2^i \lfloor (PREORDER(v) + SIZE(v) - 1)/2^i \rfloor$

O que equivale, em termos de operações com $l + 1$ bits e de modo mais direto, a:

$INLABEL(v) \leftarrow SHIFTLEFT(SHIFTRIGHT((PREORDER(v) + SIZE(v) - 1), i), i)$

ou mesmo, a:

$INLABEL(v) \leftarrow (PREORDER(v) + SIZE(v) - 1)\ AND\ SHIFTLEFT(1, i)$

A.2.3 Passo 3:

Relacionamos o $INLABEL(v)$ com o $INLABEL$ de seus ascendentes, através de $ASCENDANT(v)$.

Lema 5: Seja i o bit 1 mais à **direita** em $INLABEL(v)$. Então, para todo x, ancestral de v, os bits mais à **esquerda** do respectivo i em $INLABEL(x)$ são todos iguais (lembrando que v é considerado ancestral de v).

Então, descrevemos $ASCENDANT(v)$ da seguinte forma:

$ASCENDANT(v) = a_l(v)a_{l-1}(v)\ldots a_1(v)a_0(v)$, tal que $a_i(v) = 1$ caso i seja o índice, (da **direita** para a **esquerda**) do bit 1 mais significativo de $INLABEL(x)$, para algum $x \in V$, ancestral de v.

Obtendo $ASCENDANT(v)$: É feita uma busca em profundidade em T a partir de sua raíz, denotada por r. Da descrição acima, podemos concluir que $ASCENDANT(r) = 2^l$. Então, seja $PAI(x)$ o vértice que é pai de x, para algum $x \in V$. Para os vértices internos de T, procedemos da seguinte forma:

1. Seja $x \in V$, tal que $ASCENDANT(PAI(x))$ já foi computado.
2. Caso $INLABEL(x) = INLABEL(PAI(x))$, então $ASCENDANT(x) \leftarrow ASCENDANT(PAI(x))$.
3. Caso contrário, $ASCENDANT(x) \leftarrow ASCENDANT(PAI(x)) + 2^i$, onde i é a posição do 1 mais significtivo em $INLABEL(x)$.

   Nota: i é exatamente igual a $(INLABEL(x) - [INLABEL(x)\ AND\ (INLABEL(x) - 1)]$.

A.2.4 Passo 4:

Obtemos $LEVEL(v)$, para todo v em V, que é a profundidade de v em T, ou seja, é o número de nós no caminho da raíz até v. Isto pode ser feito com uma simples busca em profundidade.

A.2.5 Passo 5:

$HEAD(k)$ é o vértice mais próximo à raiz de T que está na mesma classe de equivalência k. A obtenção da tabela que contém $HEAD(x)$, $\forall x \in V$, é feita através de uma busca em T, em que para cada vértice v, se $INLABEL(v) \neq INLABEL(PAI(v))$, $HEAD(INLABEL(v)) \leftarrow v$.

A.3 Obtendo o MAC

Para obter $MAC(x, y)$, dividimos em dois casos:

Caso 1. $INLABEL(x) = INLABEL(y)$. Então x e y estão no caminho que contém os elementos da classe de equivalência de $INLABEL(x)$. Logo $MAC(x, y) = x$ se $LEVEL(x) \leq LEVEL(y)$, e y caso contrário.

Caso 2. $INLABEL(x) \neq INLABEL(y)$. Seja $z = MAC(x, y)$. Obtemos z nos seguintes passos:

Passo I. seja b o $MAC(INLABEL(x), INLABEL(y))$. Seja i a posição do bit 1 mais significativo em $INLABEL(x)$ XOR $INLABEL(y)$. Então b é o número inteiro tal que seus $l-i$ bits mais significativos são os mesmos que os de $INLABEL(x)$ (que são os mesmos de $INLABEL(y)$), seguido de um 1, seus i bits menos significativos são todos 0. Todas essas operações podem ser feitas através de manipulações de bits, e portanto levam tempo constante.

Passo II. **Descobrindo o INLABEL(z)**: Pelo *lema 2*, $INLABEL(z)$ é ancestral comum de $INLABEL(x)$ e de $INLABEL(y)$. Além disto, sabemos que a profundidade de b é maior o igual à de $INLABEL(z)$.

Lema 6: $INLABEL(z)$ é o menor ancestral de b em B que que é $INLABEL$ de algum ancestral comum de x e y em T.

Seja j o índice do bit 1 menos significativo em $INLABEL(z)$. Então $INLABEL(z)$ é a concatenação de:

i. prefixo de comprimento $l-j$ de $INLABEL(x)$
ii. letra 1.
iii. palavra 0^j.

Justificativa: Como z é ancestral comum de x e y, $a_j(x) = a_j(y) = 1$, da definição de $ASCENDANT$. Como z é o *menor* ancestral comum de x e y, j é o índice do 1 menos significativo em $ASCENDANT(x)$ e $ASCENDANT(y)$. Então, pelo *lema 5*, os $l-j$ bits mais significativos de $INLABEL(x)$, de $INLABEL(y)$ são iguais e de $INLABEL(z)$ são iguais. Pela definição de j, os j bits menos significativos de $INLABEL(z)$ são todos 0. Portanto obtemos $INLABEL(z)$ a partir de $INLABEL(x)$ e $INLABEL(y)$.

Passo III. Definimos $\hat{x}$ da seguinte forma: $\hat{x} = x$, se $INLABEL(x) = INLABEL(z)$. Caso contrário, seja w o filho de $\hat{x}$, e seja k o índice do bit 1 menos significativo de $INLABEL(w)$. Por argumento semelhante ao anteriormente realizado, k é o índice do bit 1 menos significativo de $ASCENDANT(x)$, e $INLABEL(w)$ é a concatenação de:

i. prefixo de comprimento $l-k$ de $INLABEL(x)$
ii. letra 1.
iii. palavra 0^k.

w é $HEAD(INLABEL(w))$, pois $INLABEL(w) \neq INLABEL(\hat{x})$, e $\hat{x}$ que é pai de w. Portanto temos $INLABEL(w)$, e $\hat{x}$, pois $\hat{x} = PAI(HEAD(INLABEL(w)))$.

Assim temos que $\hat{x}$ é o ancestral de x, na mesma classe de equivalência de z, que possui maior profundidade em T.

Analogamente definimos $\hat{y}$, que é o ancestral de y, na mesma classe de equivalência de z, que possui maior profundidade em T.

Passo IV. Então $MAC(x,y) = \begin{cases} \hat{x} & \text{se } LEVEL(\hat{x}) \leq LEVEL(\hat{y}) \\ \hat{y} & \text{caso contrário} \end{cases}$

A.4 Consumo de Espaço

Uma vez feito o pré-processamento, o algoritmo de obtenção do MAC precisa armazenar apenas as seguintes funções: $PAI, LEVEL, INLABEL, ASCENDANT, HEAD$. Todas elas possuem valores que podem ser descritos com $\lg(n)$ bits. Portanto o consumo de espaço total do algoritmo *por vértice de* T é $5\lg(n)$).

Uma implementação real que trate de valores de $n < 2^{32}$, e que trate com unidades de 8 bits, precisa de 4 bytes para armazenar valores até n, (que é o caso de todas as funções descritas) e portanto tem um consumo de espaço total *por vértice de* T de $(5 \times 4) = 20$ bytes, quando n é aproximadamente 2^{32}.

Referências Bibliográficas

[AGKR02] S. Alstrup, C. Gavoille, H. Kaplan, and T. Raulie. Nearest common ancestors: A survey and a new distributed algorithm. In *Proceedings of the 14th Annual ACM Symposium on Parallel ALgorithms and Architectures (SPAA-02)*, pages 258–264, New York, August 10–13 2002. ACM Press.

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct 1990.

[AHU76] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.*, 5:115–132, 1976.

[AMS+97] Altschul, Madden, Schäffer, Zhang, Zhang, Miller, and Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, Sep 1997. Review.

[BFC00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, 2000.

[BFC02] Michael A. Bender and Martín Farach-Colton. The level ancestor problem simplified. In *LATIN 2002: Theoretical informatics (Cancun)*, volume 2286 of *Lecture Notes in Comput. Sci.*, pages 508–515. Springer, Berlin, 2002.

[BGSV89] O. Berkman, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 309–319. ACM Press, 1989.

[BPM+00] S. Batzoglou, L. Pachter, J. P. Mesirov, B. Berger, and E. S. Lander. Human and mouse gene structure: comparative analysis and application to exon prediction. *Genome Research*, 10(7):950–958, Jul 2000.

[BPSS01] Michael A. Bender, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Finding least common ancestors in directed acyclic graphs. In *Proceedings of the*

Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01), pages 845–854, New York, January 7–9 2001. ACM Press.

[cit] Citeseer. `See <http://www.citeseer.com>`.

[CR94] Maxime Crochemore and Wojciech Rytter. *Text algorithms*. The Clarendon Press Oxford University Press, New York, 1994. With a preface by Zvi Galil.

[DKF+99] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, Jun 1999.

[dLS03] Alair Pereira do Lago and Imre Simon. *Tópicos em Algoritmos sobre Seqüências*. IMPA, Rio de Janeiro, 2003. ISBN 85-244-0202-4.

[GBT84] Harold N. Gabow, Jon Louis Bentley, and Robert E. Tarjan. Scaling and related techniques for geometry problems. In ACM, editor, *Proceedings of the sixteenth annual ACM Symposium on Theory of Computing, Washington, DC, April 30–May 2, 1984*, pages 135–143, New York, NY, USA, 1984. ACM Press.

[goo] Google. `See <http://www.google.com>`.

[Gus97] Dan Gusfield. *Algorithms on strings, trees, and sequences*. Cambridge University Press, Cambridge, 1997. Computer science and computational biology.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[KCO+01] Kurtz, Choudhuri, Ohlebusch, Schleiermacher, Stoye, and Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, Nov 2001.

[Les97] Michael Lesk. *Practical Digital Libraries : Books, Bytes, and Bucks*. Morgan Kaufmann, 1997.

[Lot83] M. Lothaire. *Combinatorics on words*. Addison-Wesley Publishing Co., Reading, Mass., 1983. A collective work by Dominique Perrin, Jean Berstel, Christian Choffrut, Robert Cori, Dominique Foata, Jean Eric Pin, Guiseppe Pirillo, Christophe Reutenauer, Marcel-P. Schützenberger, Jacques Sakarovitch and Imre Simon, With a foreword by Roger Lyndon, Edited and with a preface by Perrin.

[Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 657–666. Society for Industrial and Applied Mathematics, 2002.

[NMWP99] C.G. Nevill-Manning, I.H. Witten, and G.W. Paynter. Lexically-generated subject hierarchies for browsing large collections. *International Journal of Digital Libraries*, pages 111–123, 1999.

[Pea00] Pearson. Flexible sequence similarity searching with the FASTA3 program package. *Methods in Molecular Biology*, 132:185–219, 2000.

[PL88] Pearson and Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of USA*, 85(8):2444–2448, Apr 1988.

[SV88] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput.*, 17:1253–1262, 1988.

[TM99] T. A. Tatusova and T. L. Madden. BLAST 2 Sequences, a new tool for comparing protein and nucleotide sequences. *FEMS Microbiology Letters*, 174(2):247–250, May 1999.