UMA SOLUÇÃO SIMPLIFICADA PARA O PROBLEMA DO ANCESTRAL COMUM MAIS PROFUNDO (LOWEST COMMON ANCESTOR - LCA) EM ÁRVORES ENRAIZADAS

TRABALHO DE CONCLUSÃO DE CURSO

CIÊNCIA DA COMPUTAÇÃO

Daniel Cândido Cauás
Orientador: Paulo Gustavo Soares Fonseca

Universidade Federal de Pernambuco
Centro de Informática
Graduação em Ciência da Computação

DANIEL CÂNDIDO CAUÁS

UMA SOLUÇÃO SIMPLIFICADA PARA O PROBLEMA DO ANCESTRAL COMUM MAIS PROFUNDO (LOWEST COMMON ANCESTOR - LCA) EM ÁRVORES ENRAIZADAS

Monografia apresentada como requisito parcial para obtenção de diploma de Bacharel em Ciência da Computação pela Universidade Federal de Pernambuco.

Recife, junho de 2019

Dedico este trabalho, primeiramente, a Deus, por ser a base de minhas conquistas.

Aos meus pais Pierre Cândido Cauás e Márcia Maria Cauás, que são e sempre serão meus exemplos morais e a quem terei eterna gratidão e amor.

Ao meu professor orientador Paulo Gustavo Soares Fonseca, por sua dedicação e seus ensinamentos ao longo de todos esses anos.

Ao professor Domingos Aguiar, que me introduziu às ciências exatas e a quem serei sempre grato.

Por fim, a todos os amigos e professores que contribuíram com a minha evolução como estudante e como pessoa.

Resumo

Um problema algorítmico recorrente em áreas diversas como Engenharia de Software ou Biologia Computacional é o de encontrar o ancestral comum mais profundo entre dois ou mais nós em árvores, conhecido como *Lowest Common Ancestor* – LCA. As principais soluções existentes consistem em pré-processar a árvore, de forma a criar uma estrutura que agilize uma consulta de LCA e, portanto, essas soluções são avaliadas em termos do tempo e espaço de pré-processamento e de consulta.

O objetivo deste trabalho é propor uma solução simples para consultas de LCAs sucessivas e avaliar o seu desempenho em termos teóricos e práticos com respeito ao tempo de pré-processamento, consulta e uso de memória, em relação a algumas soluções consagradas encontradas na literatura.

Palavras-chave: Ancestral Comum Mais Profundo; Consulta de Mínimo em Intervalo; Árvores; Estruturas de Dados; Algoritmos.

Abstract

A recurrent algorithmic problem in several areas, such as Software Engineering and Computational Biology, is to find the Lowest Common Ancestor (LCA) between two or more nodes in a tree. The main existing solutions consist in preprocessing the tree in order to create a structure that speeds up an LCA query and thus, they are evaluated in terms of time and space of preprocessing and query.

The objective of this work is to propose a simple solution for a large number of successive LCA queries and to evaluate its performance in theoretical and practical terms with respect to the time of preprocessing, querying and memory usage, compared to some well-established solutions found in the literature.

Keywords: Lowest Common Ancestor; Range Minimum Query; Trees; Data Structures; Algorithms.

Sumário

1. Introdução **6**

2. Lowest Common Ancestor **7**

2.1 LCA - consulta única 7

2.2 LCA - múltiplas consultas 8

2.3 Valor Mínimo em Intervalo - RMQ 14

2.4 Sparse Table - RMQ eficiente 14

2.5 LCA ótimo - abordagem ⟨O(n), O(1)⟩ 17

3. LCA simplificado ⟨O(n), O(lg(n))⟩ **23**

4. Análise experimental **28**

4.1 Tempo de pré-processamento 28

4.2 Tempo de consulta 29

4.3 Uso de memória 29

5. Conclusão **31**

Bibliografia **32**

1. Introdução

Em Ciência da Computação, as árvores são estruturas amplamente conhecidas e utilizadas nas mais diversas aplicações, devido à sua versatilidade. As árvores formam a base de implementação de inúmeras estruturas de dados de várias linguagens de programação, como as estruturas “conjunto” (set) e “mapa” (map) da linguagem C++, que utilizam árvores balanceadas devido à sua propriedade de encontrar elementos em tempo logarítmico em função do número de nós[1]. Na área de Engenharia de Software, por exemplo, são usadas para modelar hierarquias entre classes em linguagens orientadas a objetos[2]. Na área de Biologia Computacional, são usadas para representar árvores evolutivas[3]. Nessas e em outras aplicações, um problema recorrente é o de determinar o ancestral comum mais profundo entre dois nós, problema conhecido como *Lowest Common Ancestor* – LCA. A figura 1.1 exemplifica operações de LCA entre pares de nós.

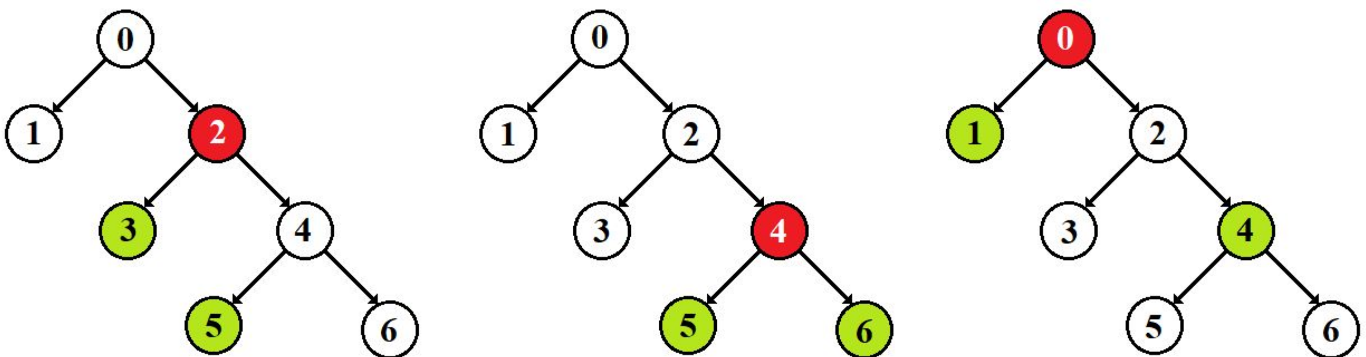

Figura 1.1. Três exemplos de LCAs numa mesma árvore, onde os nós de entrada estão em verde e seus LCAs, em vermelho.

As estratégias para resolver esse problema variam de acordo com o número de consultas de LCA que se quer realizar. Se apenas uma consulta é necessária, pode-se realizar uma abordagem mais direta, como será visto a seguir, mas se várias consultas serão feitas, será necessário realizar um pré-processamentos na árvore para agilizar o tempo de consulta.

Neste trabalho, o capítulo 2 abordará as soluções ótimas já consagradas pela literatura para o problema LCA. No capítulo 3, será proposta uma abordagem simplificada para esse problema. No capítulo 4, serão comparados os resultados empíricos da abordagem ótima e da abordagem simplificada. Finalmente, o capítulo 5 abordará as conclusões sobre a implementação e sobre os testes comparativos.

2. Lowest Common Ancestor

Neste capítulo, iremos falar das soluções consagradas da literatura para o problema do LCA. A solução para o LCA que será vista ao longo do capítulo está descrita em [7], cujo algoritmo será re-explicado com a finalidade de esclarecer os trabalhos já realizados acerca do LCA e de introduzir conceitos que serão usados na abordagem simplificada que será posteriormente introduzida. Para isso, vamos nos ater às árvores enraizadas.

Uma árvore enraizada não vazia é um grafo dirigido acíclico tal que todo nó tem no máximo um pai. Existirá um único nó fonte (sem arestas que apontam para ele) chamado raiz. Árvores enraizadas contemplam as propriedades 2.1 e 2.2.

Propriedade 2.1. Não existirão duas arestas $u \rightarrow w$ e $v \rightarrow w$ com $u \neq v$, ou seja, u terá, no máximo, um pai.

Propriedade 2.2. Para cada nó v, existirá apenas um caminho de r até v.

Vamos considerar as seguintes notações:

- T será a representação da árvore;
- r será a raiz de T;
- u, v, w serão nós quaisquer de T;
- w_i será o i-ésimo filho do nó w;
- n será o tamanho de T, ou seja, o número de nós de T;
- pai(u) será o pai do nó u;
- prof(u) é a profundidade do nó u, ou seja, o número de arestas no caminho de r a u;
- Uma folha é um nó que não tem filhos.

Vistas as notações e propriedades, podemos introduzir o LCA.

2.1 LCA - consulta única

Para responder uma consulta de LCA, não é necessário fazer pré-processamento da árvore. Em linguagens de programação, há diversas formas de se representar uma árvore, uma delas é como um vetor cuja i-ésima posição representa o pai do nó i. Por exemplo, uma árvore $T = (-1, 0, 0, 1, 1, 2, 2)$ pode ser representada como na figura 2.1.

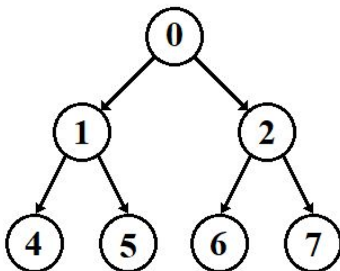

Figura 2.1. Árvore T representada pelo vetor (–1, 0, 0, 1, 1, 2, 2)

Sejam u e v dois nós de T. Pode-se descobrir o LCA de u e v pelo Algoritmo 2.3.

```
Algoritmo LCA-simples
Entrada: T (vetor de pais dos nós); u, v (nós)
Saída: LCA(u,v)
Início
	marcado = (0, 0, … , 0)		/* vetor de tamanho n */
	enquanto u ≠ –1:
		marcado[u] = 1
		u = T[u]
	fim-enquanto
	enquanto marcado[v] ≠ 1
		v = T[v]
	fim-enquanto
	retorna v
Fim
```

Algoritmo 2.3. LCA sem pré-processamento.

O algoritmo baseia-se em escolher um dos nós, marcar todos os ancestrais desse nó (incluindo ele próprio) até a raiz da árvore. Após isso, partindo do outro nó, observa-se todos os ancestrais dele até que se encontre alguém marcado, que será o ancestral mais profundo de ambos os nós e, portanto, o LCA.

Apesar da simplicidade, o Algoritmo 2.3 tem complexidades de espaço e de tempo lineares sobre o tamanho da árvore, pois é necessário criar-se um vetor do tamanho da árvore para que se possa marcar o caminho dos nós até a raiz. Como a árvore pode ter qualquer formato (como o de uma lista encadeada), um LCA entre uma folha e a raiz da árvore, no pior caso, se daria em tempo $O(n)$.

No caso de uma consulta única, tem-se então uma abordagem em tempo linear como solução, o que não é um problema, tendo em vista que o algoritmo deve, inicialmente, ler a árvore inteira para poder melhor representá-la, o que consome um tempo também linear, tornando esse algoritmo assintoticamente ótimo, pois, no pior caso, é preciso ao menos ler toda a árvore.

Contudo, se o contexto exige que sejam feitas $k > 1$ consultas de LCAs, o algoritmo as responderia em tempo $O(kn)$. Como veremos a seguir, é possível realizar certos pré-processamentos com a árvore para agilizar o tempo dessas consultas.

2.2 LCA - múltiplas consultas

Para agilizar o tempo de múltiplas consultas sobre a mesma árvore, a ideia é pré-processar essa árvore e montar uma estrutura que agilize cada consulta[7]. Para criar essa estrutura, vamos definir, inicialmente, uma forma de se fazer um percurso na árvore:

Definição 2.4. O Percurso em Profundidade (DFP) de um nó u pode ser descrito por:

1. $\text{DFP}(\varnothing) = \varnothing$;
2. $\text{DFP}(u) = u$ se u for uma folha, ou seja, se não tiver filhos;
3. $\text{DFP}(u) = u \rightarrow \text{DFP}(u_1) \rightarrow u \rightarrow \dots \rightarrow u \rightarrow \text{DFP}(u_k) \rightarrow u$ se u tem $k > 0$ filhos, sendo u_i o i-ésimo filho de u.

A figura 2.5 exemplifica um DFP partindo da raiz de uma árvore. Seu percurso é similar ao de uma Busca em Profundidade[4], com o diferencial de que sempre que se termina o percurso numa subárvore de um nó, esse nó entra no percurso novamente.

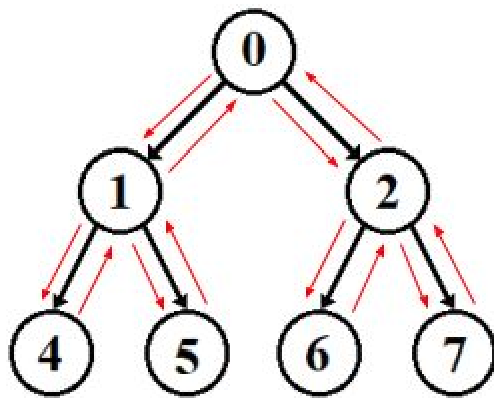

Figura 2.5. Caminho decorrente do DFP(0).

O DFP na raiz dessa árvore é $0 \rightarrow 1 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 1 \rightarrow 0 \rightarrow 2 \rightarrow 5 \rightarrow 2 \rightarrow 6 \rightarrow 2 \rightarrow 0$.

Proposição 2.6. Seja T uma árvore com n nós. O DFP da raiz de T visita exatamente $2n - 1$ nós.

Prova: Indução em n .

Caso base: $n = 1$.

Se existe apenas um nó, o caminho que o DFP faz visita apenas ele, pelo caso (2) da definição 2.4.

$2 \times 1 - 1 = 1$ [OK]

Hipótese indutiva: Se T tem n nós, o DFP da raiz visita exatamente $2n - 1$ nós.

Passo indutivo: Se T tem $n + 1$ nós, é preciso provar que o DFS da raiz de T visita $2(n + 1) - 1$ nós.

Seja u um nó qualquer de T. Se adicionarmos um novo nó v em u, garantindo que T tenha $n + 1$ nós, saberemos, pelo caso (3) da definição 2.4, que o DFP de u visitará v antes de encerrar, ou seja, supondo que u tenha $k \geq 0$ filhos além de v, temos agora que:

$\text{DFP}(u) = u \rightarrow \text{DFP}(u_1) \rightarrow u \rightarrow \ldots \rightarrow u \rightarrow \text{DFP}(u_k) \rightarrow u \rightarrow \text{DFP}(v) \rightarrow u$.

Tem-se que dois novos fatores foram incluídos no DFP: u e DFP(v). Como v é uma folha, pelo caso (2) da definição 2.4, DFP(v) = v, logo:

$\text{DFP}(u) = u \rightarrow \text{DFP}(u_1) \rightarrow u \rightarrow \ldots \rightarrow u \rightarrow \text{DFP}(u_k) \rightarrow u \rightarrow v \rightarrow u$.

Conclui-se que o caminho visitado aumenta em dois nós. Temos, pela hipótese indutiva, que uma árvore de n nós gera um caminho de $2n - 1$ nós, logo:

$(2n - 1) + 2 = 2(n + 1) - 1 \quad \Rightarrow \quad 2n + 1 = 2n + 1$ [OK].

Os primeiros pré-processamentos a serem feitos são os vetores correspondentes ao DFP da árvore, à profundidade e à primeira ocorrência de cada nó do DFP. Como provado anteriormente, o DFP terá tamanho $2n - 1$, logo o vetor de profundidades também terá tamanho $2n - 1$, enquanto o vetor de primeira ocorrência terá tamanho n, já que há apenas uma primeira ocorrência para cada um dos n nós.

Exemplo 2.7. Seja T a árvore ilustrada na figura 2.8. Inicialmente, vamos pré-processar os vetores D, P e E, que representam os nós visitados no DFP, as profundidades relativas aos nós visitados no DFP e os índices da primeira ocorrência de cada nó no DFP, respectivamente.

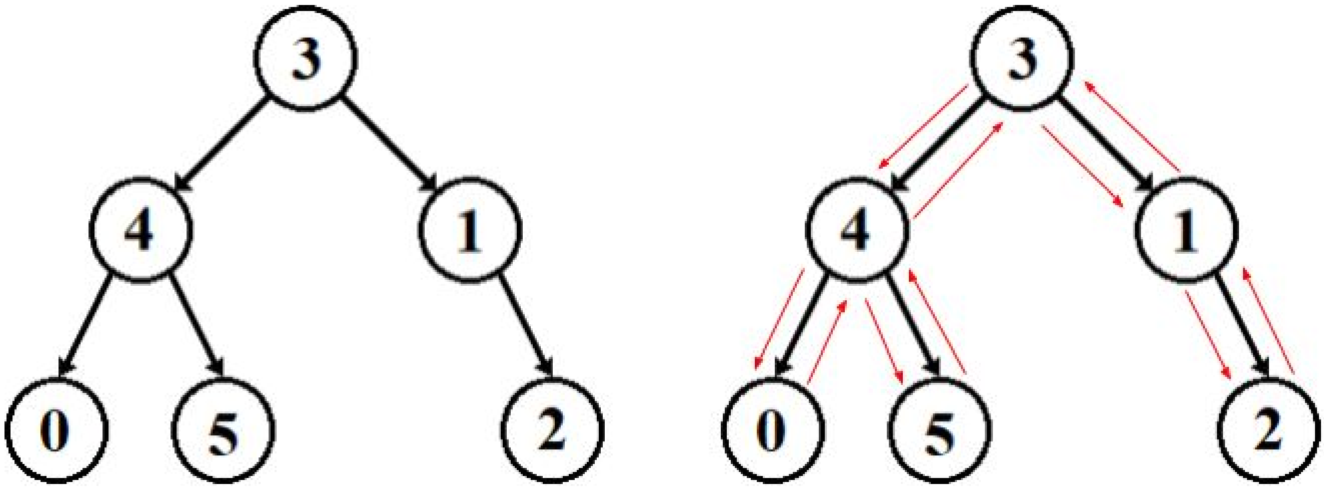

Figura 2.8. Árvore T de tamanho 6 e seu respectivo DFP.

O DFP da raiz da árvore é $3 \rightarrow 4 \rightarrow 0 \rightarrow 4 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 3$, logo:

$D =$ (3, 4, 0, 4, 5, 4, 3, 1, 2, 1, 3)
$P =$ (0, 1, 2, 1, 2, 1, 0, 1, 2, 1, 0)
$E =$ (2, 7, 8, 0, 1, 4)

Com essas informações, é possível propor os seguintes lemas:

Lema 2.9. Seja T uma árvore enraizada em r e w um nó qualquer de T. O percurso entre a primeira e a última ocorrência de w em DFP(r) corresponde a DFP(w).

Prova: Por contradição.

Sabemos que em algum momento, DFP(r) visitará w, pois DFP(r) cobre todos os nós de T. Nesse momento, DFP(w) será integrado no percurso corrente. Sabemos, pelos casos (2) e (3) da definição 2.4, que DFP(w) começa e termina visitando w. Se houver outra visita a w após DFP(w), significa que w tem mais de um pai, pois, pelo caso (3) da definição 2.4, um nó não olha para um mesmo filho duas vezes, logo algum outro nó teria que chamar o DFP(w) novamente. Pela propriedade 2.1, temos que w não pode ter mais de um pai, logo w não ocorre novamente depois de DFP(w) ser integrado em DFP(r).

Lema 2.10. Todas as ocorrências de u e v estarão entre a primeira ocorrência e a última ocorrência de LCA(u, v).

Prova: Por contradição.

Seja w o LCA de u e v. Sabemos que u e v ocorrem em DFP(w), pois, por definição, w é raiz de uma árvore que contém u e v (pois é o LCA de u e v) e DFP(w) percorre toda a árvore enraizada em w.

Sabemos, pelo lema 2.11, que a primeira e a última ocorrência de w demarcam o DFP de w. Dito isto, sem perda de generalidade, se u aparecer antes da primeira ocorrência de w, significa que existe um caminho da raiz da árvore até um deles que não passa por w, pois u foi alcançado sem que w tenha sido visitado.

Como w é o LCA de u e v, por definição, existe um caminho de w até u e de w até v. Pela propriedade 2.2, não pode haver mais de um caminho entre a raiz e um nó, portanto é impossível que u ou v ocorram antes de w no DFP da árvore.

Quanto ao caso de u ocorrer depois de DFP(w), sem perda de generalidade, a prova é análoga, já que pelo caso (3) da definição 2.4, sabemos que DFP(w) só será chamado uma vez, então se u ou v ocorrerem após DFP(w), significa que vieram por outro caminho que não passa por w, quebrando a propriedade 2.2, tornando essa possibilidade igualmente impossível.

Lema 2.11. O LCA de dois nós u e v sempre estará entre uma ocorrência de u e uma ocorrência de v no DFP.

Prova: Há 2 casos possíveis:

1. Se LCA(u, v) = u ou LCA(u, v) = v, segue que o LCA está dentro do intervalo, pois ele será um dos limites.
2. Caso contrário, seja w o LCA de u e v. Pelo caso (3) da definição 2.4, temos $\text{DFP}(w) = w \rightarrow \text{DFP}(w_1) \rightarrow w \rightarrow \text{DFP}(w_2) \rightarrow w \rightarrow \ldots \rightarrow w \rightarrow \text{DFP}(w_k) \rightarrow w$.
   É possível notar que u e v estão contidos em duas subárvores enraizadas em w_i e w_j, onde $i \neq j$, pois se $i = j$, o LCA não seria w mas sim w_i (pois w_i é mais profundo que w e seria ancestral de u e v). Daí, segue-se que w estará no meio do percurso entre qualquer par de DFPs de filhos de w, que é onde u e v ocorrem pela primeira vez, vide lema 2.10.

Na figura 2.8, tomemos por exemplo os nós 4 e 2. A primeira ocorrência do 4 é na posição $E[4] = 1$ de D e a primeira ocorrência do 2 é na posição $E[2] = 8$ de D. Entre as posições 1 e 8 de D, ocorrem os nós (4, 0, 4, 5, 4, 3, 1, 2). Como o LCA de 4 e 2 é 3, examinemos o 3:

DFP(3) = 3 → DFP(4) → 3 → DFP(1) → 3.

Sabemos que DFP(4), pelos casos (2) e (3) da definição 2.4, contém pelo menos o próprio 4, logo DFP(3) = 3 → {4, …} → 3 → DFP(1) → 3.

Por último, temos que DFP(1) = 1 → DFP(2) → 1. Como 2 é uma folha, pelo caso (2) da definição 2.4, temos que DFP(1) = 1 → 2 → 1, logo:

DFP(3) = 3 → {4, …} → **3** → 1 → 2 → 1 → 3.

Temos, enfim, que o LCA estará, de fato, entre as primeiras ocorrências de cada nó.

Corolário 2.12. Se w é o LCA de u e v, então ou um desses nós é o próprio w ou ambos os nós estão em subárvores distintas de w.

Corolário 2.13. Todas as ocorrências de u e v estão dentro de DFP(LCA(u, v)).

Lema 2.14. Sejam T uma árvore de raiz r e w um nó de T. O nó menos profundo do DFP de w é o próprio w.

Prova: Por contradição.

Temos que DFP(w) = $w \rightarrow$ DFP(w_1) $\rightarrow w \rightarrow \ldots \rightarrow w \rightarrow$ DFP(w_k) $\rightarrow w$, logo, é possível notar que $\forall w_i$ (prof(w_i) = 1 + prof(w)). O mesmo ocorre para os filhos de cada w_i e assim, sucessivamente, o que indica que todos os nós visitados em DFP(w) tem, no mínimo, uma profundidade maior que a de w. Vamos supor que em DFP(w) exista um nó $\hat{w}$ menos profundo que w. Isto pode significar duas coisas:

(i) Existe um caminho de w até $\hat{w}$ que não forma um ciclo. Se $\hat{w}$ é menos profundo que w, significa que também existe um caminho $r \rightarrow \ldots \rightarrow \hat{w}$ que não passa por w, pois se passasse, não seria possível $\hat{w}$ ser menos profundo que w, portanto é notável que podemos chegar em $\hat{w}$ por este caminho e por um caminho $r \rightarrow \ldots \rightarrow w \rightarrow \ldots \rightarrow \hat{w}$ que eventualmente passa por w. Logo, há dois caminhos de r até $\hat{w}$, quebrando a propriedade 2.2, então, nesse caso, não é possível que $\hat{w}$ exista.

(ii) Existe um caminho de w até $\hat{w}$ que forma um ciclo. Como árvores são estruturas acíclicas, também é impossível que exista tal $\hat{w}$.

A figura 2.15 demonstra os dois casos, onde w = 3 e $\hat{w}$ é o nó que, se supostamente fosse encontrado durante o DFP de w, criaria um problema estrutural que, caso ocorra, quebra as propriedades de uma árvore.

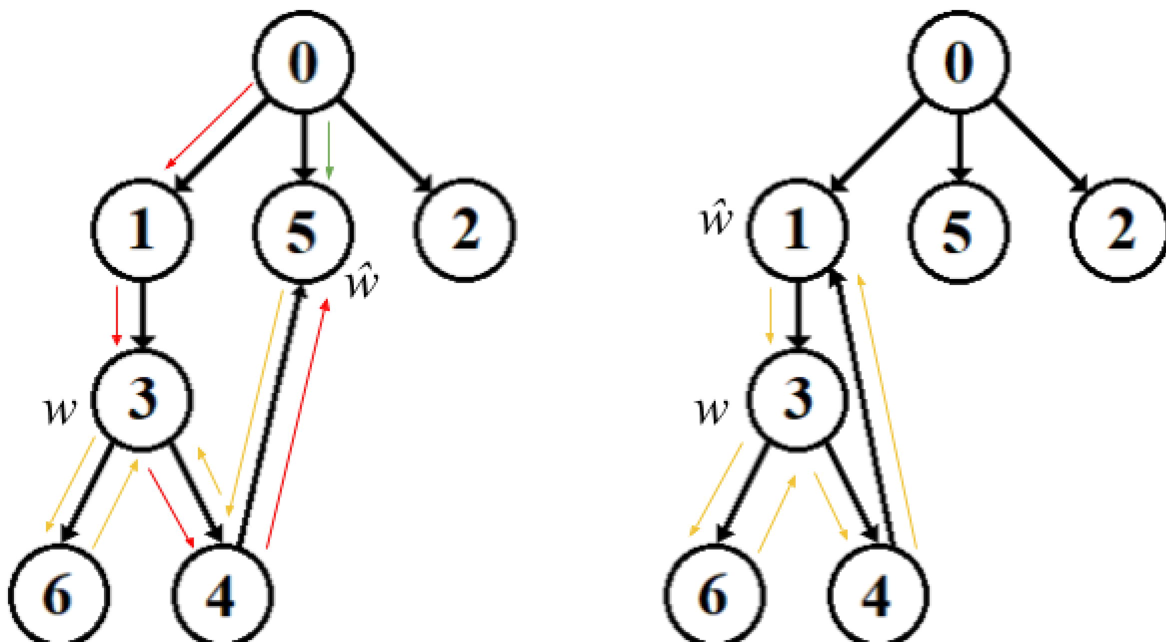

Figura 2.15. À esquerda, o caso (i), onde existem dois caminhos entre r e $\hat{w}$, ilustrados em setas vermelhas e verdes, que são os caminhos 0 → 1 → 3 → 4 → 5 e 0 → 5. À direita, o caso (ii), onde DFP(w) cria o ciclo 3 → 6 → 3 → 4 → 1.

Como todas as possibilidades onde $\hat{w}$ existe quebram propriedades fundamentais da árvore, por contradição, não existe tal $\hat{w}$.

Com os lemas acima, podemos provar um dos teoremas fundamentais do LCA:

Teorema 2.16. O LCA de dois nós u e v quaisquer sempre será o nó de menor profundidade que se encontra entre entre $E[u]$ e $E[v]$.

Prova: Seja w o LCA de u e v. Pelo corolário 2.13 (que provém dos lemas 2.9 e 2.10), sabemos que todas as ocorrências de u e v se encontram em DFP(w).

Pelo lema 2.11, sabemos que o LCA de u e v sempre estará entre um par de ocorrências qualquer de u e v. Por comodidade, vamos considerar que esse par de ocorrências seja $E[u]$ e $E[v]$.

Do corolário 2.13, também temos que todos os elementos entre $E[u]$ e $E[v]$ estão contidos em DFP(w), pois são visitados após $E[u]$ e antes de $E[v]$ durante o DFP(w).

Pelo lema 2.14, temos que w é o nó de menor profundidade entre todos os nós de DFP(w).

Se w está entre $E[u]$ e $E[v]$ e todos os elementos entre $E[u]$ e $E[v]$ estão contidos em DFP(w), decorre (pelos lema 2.11 e 2.14) que w, de fato, será o nó de menor profundidade entre $E[u]$ e $E[v]$.

Em suma, é preciso pré-processar 3 vetores D, P e E como demonstra o exemplo 2.7. O espaço utilizado é de $2(2n - 1) + n = 5n - 2$, que totalizam complexidade de espaço $O(n)$. Podemos realizar um DFP que preencha todos os vetores em tempo $O(n)$, conforme exibido no Algoritmo 2.17. Nesse algoritmo, levamos em conta que algumas variáveis são globais[5], isto é, podem ser acessadas livremente em qualquer trecho do procedimento. É importante salientar que, neste algoritmo, a árvore é representada como uma lista de adjacências.

```
/* variáveis globais */
posição = 0
D, P = (0, … , 0)                    /* vetores de tamanho 2n – 1 */
E = (0, … , 0)                       /* vetor de tamanho n */

Algoritmo DFP
Entrada: u (nó); d (profundidade do nó); T (árvore)
Saída: vazia
Início
    D[posição] = u
    P[posição] = d
    E[u] = posição
    posição = posição + 1
    para cada filho v de u em T:
        DFP(v, d + 1, T)
        D[posição] = u
        posição = posição + 1
    fim-para
Fim
```

Algoritmo 2.17. DFP que preenche D, P e E numa única travessia na árvore. Como exibido no exemplo 2.7, é preciso passar a raiz da árvore T como parâmetro para o DFP, ou seja, é necessário chamar DFP(r, 0, T) para corretamente preencher os vetores.

Com esses vetores definidos, podemos reduzir o problema inicial de achar o LCA de dois nós a achar o nó de menor profundidade entre dois índices do DFP da raiz da árvore.

2.3 Valor Mínimo em Intervalo - RMQ

Outro problema recorrente em Ciência da Computação é o de encontrar o índice do menor valor dentro de um intervalo em um vetor, conhecido como *Range Minimum Query* – RMQ[6]. Como visto anteriormente, o problema de encontrar o LCA de dois nós foi reduzido a encontrar o nó de menor profundidade num intervalo dentro do vetor do DFP. Podemos reduzir um problema de LCA para um problema de RMQ da seguinte maneira:

Redução LCA→RMQ:

Sejam u e v dois nós quaisquer de uma árvore;
Seja $i = E[u]$ e $k = E[v]$;
Seja j o RMQ(i, k) no vetor P;
LCA(u, v) = $D[j]$.

Dado que temos essa redução, o objetivo é ser capaz de responder uma consulta de RMQ entre dois índices em tempo hábil, isto é, com uma complexidade inferior a O(n), caso contrário cairíamos no mesmo resultado de usar o algoritmo LCA-simples. Felizmente, a literatura já nos oferece soluções eficientes para o RMQ. Neste trabalho, a solução abordada será baseada em estruturas chamadas tabelas esparsas (*Sparse Tables*).

2.4 Sparse Table - RMQ eficiente

Seja Z um vetor de tamanho n. A *Sparse Table* consiste numa tabela que é capaz de responder uma consulta de RMQ em Z em tempo constante, isto é, O(1).

A ideia por trás da *Sparse Table* é calcular os mínimos entre intervalos de tamanhos potências de 2. A i-ésima coluna corresponde a RMQs de intervalos de tamanho 2^i, logo podemos concluir que haverá $\lfloor \lg(n) \rfloor + 1$ colunas, pois $2^{\lfloor \lg(n) \rfloor}$ é a maior potência de 2 menor ou igual a n (e não podemos ter RMQ de um intervalo maior que o próprio vetor).

Seja S uma *Sparse Table* de um vetor Z qualquer. Temos que $S[i][j]$ será o RMQ do intervalo que começa em $Z[i]$ e tem tamanho 2^j, ou seja, de $Z[i]$ até $Z[i + 2^j - 1]$. Tendo essa característica, é possível calcular os valores de cada elemento de uma certa coluna usando os elementos da coluna anterior, por exemplo, $S[0][1]$ = mínimo($S[0][0]$, $S[1][0]$), isto é, o RMQ do intervalo que inicia em 0 e tem tamanho 2^1 é o mínimo entre o intervalo que inicia em 0 e tem tamanho 2^0 e o que inicia em 1 e tem tamanho 2^0.

Suponhamos $Z = (1, 5, 3, 6, 2, 4, 7)$. A *Sparse Table* S correspondente a Z é exibida na figura 2.18.

0	1	2
1	1	1
5	3	2
3	3	2
6	2	2
2	2	–
4	4	–
7	–	–

Figura 2.18. *Sparse Table* do vetor $Z = (1, 5, 3, 6, 2, 4, 7)$.

Como Z tem tamanho 7, S terá 7 linhas e $\lfloor \lg(7) \rfloor + 1 = 3$ colunas. A primeira coluna representa os mínimos entre intervalos de tamanho $2^0 = 1$ em Z, que é o próprio Z.

Para realizar uma consulta de RMQ nessa tabela, é necessário saber qual a maior potência de 2 menor ou igual ao tamanho do intervalo a ser consultado. Se quisermos, por exemplo, o RMQ(2, 6) em Z, ou seja, o índice do valor mínimo entre os índices 2 e 6 em Z, precisamos calcular o tamanho desse intervalo, que é $(6 – 2) + 1 = 5$. A maior potência de 2 menor ou igual a 5 é $\lfloor \lg(5) \rfloor = 2$, que corresponde a $2^2 = 4$, logo a coluna que nos responderá essa consulta será a coluna 2. Sabemos que $S[2][2]$ corresponde ao mínimo do intervalo que começa em 2 e tem tamanho $2^2 = 4$, ou seja, o mínimo entre $\{3, 6, 2, 4\}$. Como o intervalo desejado tem tamanho 5, precisamos levar mais um valor em conta na resposta, que é o $Z[6] = 7$, no entanto, não é preciso voltar para outras colunas para incluí-lo. Podemos representar o mínimo do intervalo [2, 6] como o mínimo entre os intervalos [2, 5] e [3, 6], que são dois intervalos de tamanho 4 (e, portanto, já foram calculados pela *Sparse Table*).

Dessa maneira, é notável que uma *Sparse Table* consegue responder RMQs em tempo $O(1)$, pois não é necessário fazer nenhum tipo de iteração.

Contudo, como a tabela tem tamanho $n \lg(n)$, a complexidade para preenchê-la será de $O(n \lg(n))$, mas esse preço não é tão alto, visto que após a tabela ser preenchida, o que ocorre uma única vez, k consultas de RMQ são realizadas em tempo $O(k)$.

Para o LCA, é importante ter em mente que os nós da árvore são comparáveis por sua profundidade, ou seja, para compararmos u e v, comparamos $P[E[u]]$ com $P[E[v]]$. O LCA de u e v será o nó w tal que $\forall i \ (E[u] \leq i \leq E[v] \rightarrow P[E[w]] \leq P[i])$.

O procedimento de comparação entre dois nós é demonstrado no Algoritmo 2.19, enquanto a construção da *Sparse Table* é demonstrada no Algoritmo 2.20 e a consulta de RMQ é demonstrada no Algoritmo 2.20.

D, P e E são referenciados nesses algoritmos, mas como são variáveis globais (vide Algoritmo 2.17), não é necessário passá-los como parâmetro.

```
Algoritmo Nó-Menos-Profundo
Entrada: u, v (nós)
Saída: nó menos profundo entre u e v
Início
	se P[E[u]] < P[E[v]]
		retorna u
	fim-se
	retorna v
Fim
```

Algoritmo 2.19. Retorna o nó menos profundo entre *u* e *v*.

```
/* variáveis globais */
S = matriz em branco			/* variável declarada mas ainda não inicializada */

Algoritmo Montar-Sparse-Table
Entrada: P (vetor); n (tamanho de P)
Saída: Sparse Table correspondente a P
Início
	S = matriz n por ⌊lg(n)⌋ + 1
	para i = 0, … , n – 1:
		S[i][0] = P[i]
	fim-para
	para j = 1, … , ⌊lg(n)⌋:
		limite = n – 2^j
		para i = 0, … , limite:
			S[i][j] = Nó-Menos-Profundo( S[i][j – 1] , S[i + 2^(j – 1)][j – 1] )
		fim-para
	fim-para
Fim
```

Algoritmo 2.20. Preenche a *Sparse Table S* do vetor *P*.

```
/* variáveis globais */
S = matriz n por ⌊lg(n)⌋ + 1		/* variável já preenchida pelo Algoritmo 2.20 */

Algoritmo RMQ-Sparse-Table
Entrada: l, r (índices que delimitam o intervalo)
Saída: RMQ(l, r)
Início
	tamanho = r – l + 1
	coluna = ⌊lg(tamanho)⌋
	retorna Nó-Menos-Profundo( S[l][coluna] , S[l + tamanho – 2^coluna][coluna] )
Fim
```

Algoritmo 2.21. Realiza a consulta do RMQ entre os índices *l* e *r*.

Dada a implementação da *Sparse Table*, sabe-se que a complexidade de tempo e espaço de sua construção é de O(n lg(n)). Nos tópicos anteriores, observamos que o caminho que a DFS percorre na árvore tem tamanho O(n), então o pré-processamento desse caminho seria, também, O(n lg(n)), com tempo de consulta constante, garantido pela *Sparse Table* e pela Redução LCA→RMQ.

Como a avaliação das soluções para múltiplas consultas de LCA envolvem sempre medir as complexidades de tempo/espaço de pré-processamento da árvore e o tempo de consulta, a partir daqui adota-se a sintaxe ⟨P, C⟩, onde P é a complexidade do pré-processamento e C é a complexidade da consulta.

Até agora, sabemos construir uma abordagem ⟨O(n lg(n)), O(1)⟩, com o uso da *Sparse Table*.

A seguir, vamos apresentar a abordagem ótima ⟨O(n), O(1)⟩.

2.5 LCA ótimo - abordagem ⟨O(n), O(1)⟩

Para que seja possível fazer um pré-processamento de complexidade O(n), é necessário otimizar, de alguma maneira, os procedimentos da *Sparse Table* vistos acima.

Até o momento, temos uma complexidade de espaço O(n lg(n)). Para que essa complexidade seja reduzida, é necessário que a *Sparse Table* seja construída sobre um vetor com tamanho menor que n, caso contrário sua complexidade não será afetada.

A abordagem proposta em [7] é a de dividir o vetor do percurso da DFS em blocos de tamanho $\frac{\lceil lg(n) \rceil}{2}$.

Tomemos, por exemplo, a árvore da figura 2.22.

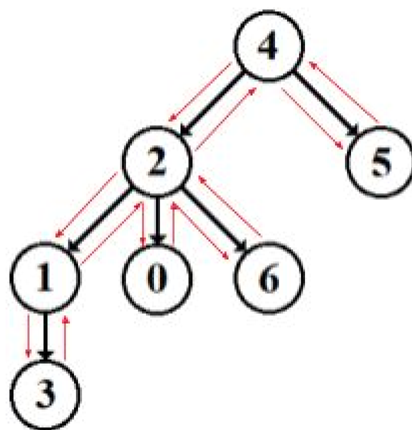

Figura 2.22. Árvore de 7 nós enraizada no nó 4 e seu DFP.

D = (4, 2, 1, 3, 1, 2, 0, 2, 6, 2, 4, 5, 4)
P = (0, 1, 2, 3, 2, 1, 2, 1, 2, 1, 0, 1, 0)
E = (6, 2, 1, 3, 0, 11, 8)

Como essa árvore tem 7 nós, o percurso da DFS terá 13 nós, o tamanho de um bloco será de $\frac{\lceil lg(13) \rceil}{2}$ = 2. Quebramos o DFP em blocos de tamanho 2 como na figura 2.23.

D	4 | 2	1 | 3	1 | 2	0 | 2	6 | 2	4 | 5	4 | –
P	0 | 1	2 | 3	2 | 1	2 | 1	2 | 1	0 | 1	0 | –

Figura 2.23. Decomposição de *D* e *P* em blocos.

A ideia para reduzir a complexidade da *Sparse Table* é criar um novo vetor *R*, onde cada elemento será o representante de um bloco. Esse representante será o nó com menor profundidade dentro do bloco, como ilustra a figura 2.24.

D	4 | 2	1 | 3	1 | 2	0 | 2	6 | 2	4 | 5	4 | –
R	4	1	2	2	2	4	4

Figura 2.24. Criação do vetor de *R* de representantes de cada bloco.

O vetor *R* é, claramente, menor que *D*. Como *D* tem tamanho $2n - 1$ e um bloco terá tamanho $\frac{\lceil lg(2n-1) \rceil}{2}$, teremos

$$\frac{2n-1}{\left(\frac{\lceil lg(2n-1) \rceil}{2}\right)} = \frac{4n-2}{\lceil lg(2n-1) \rceil}$$

blocos, sendo esse valor o tamanho do vetor *R*.

Para mostrar que pré-processar uma *Sparse Table* sobre *R* é um procedimento linear, basta lembrar que o número de colunas dela corresponde a $\lfloor \lg(n) \rfloor + 1$, onde *n* é o tamanho do vetor a ser processado, que nesse caso é o vetor *R*, de tamanho $2n - 1$. O tamanho da *Sparse Table* será:

$$\frac{4n-2}{\lceil lg(2n-1) \rceil} \times \left(lg\left(\frac{4n-2}{\lceil lg(2n-1) \rceil}\right) + 1 \right)$$

Sabemos que $O\left(\frac{4n-2}{\lceil lg(2n-1) \rceil}\right) = O\left(\frac{n}{lg(n)}\right)$.

Temos então $O\left(\frac{n}{lg(n)}\right) \times O\left(lg\left(\frac{n}{lg(n)}\right)\right)$

Que podemos reescrever na forma $O(n) \times O\left(\frac{lg\left(\frac{n}{lg(n)}\right)}{lg(n)}\right)$.

Sabemos que $\frac{n}{lg(n)} \leq n$, então $\frac{lg\left(\frac{n}{lg(n)}\right)}{lg(n)} \leq 1$, logo

$O(n) \times O(x \leq 1) \leq O(n)$.

Tendo provado que a construção da *Sparse Table* é linear em tempo e espaço, vamos discutir os requisitos para calcular o LCA após dividir o D em blocos.

Para calcular o LCA de dois nós u e v, temos que saber os blocos onde eles aparecem pela primeira vez, isto é, os blocos que representam as posições $E[u]$ e $E[v]$. O bloco de um nó w qualquer é dado por $b_w = \frac{2 \cdot E[w]}{\lceil lg(2n-1) \rceil}$. Chamemos os blocos de u e v de b_u e b_v, respectivamente. Temos duas possibilidades:

1. Se $b_u \neq b_v$, ou seja, quando u e v caem em blocos diferentes, o LCA de u e v será o nó de menor profundidade entre 3 nós:
   a. O RMQ de $E[u]$ até o fim de b_u.
   b. O RMQ entre os blocos $b_u + 1$, $b_u + 2$, … , $b_v - 1$.
   c. O RMQ do início de b_v até $E[v]$.
2. Se $b_u = b_v$, ou seja, quando u e v caem no mesmo bloco, o LCA de u e v será o nó de menor profundidade entre os índices $E[u]$ e $E[v]$ dentro de b_u.

A figura 2.25 ilustra a situação (1), enquanto a figura 2.26 ilustra a situação (2).

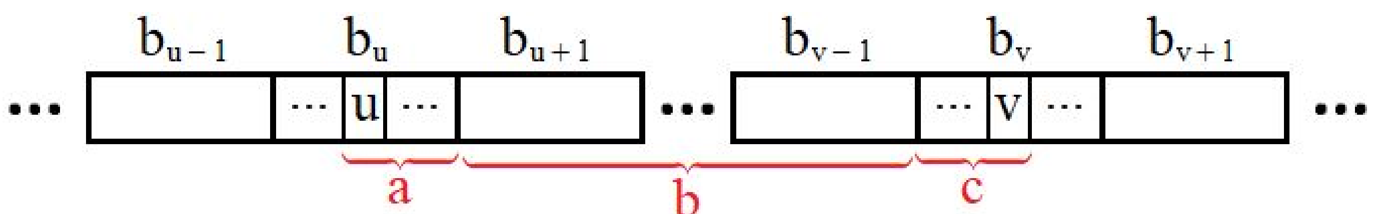

Figura 2.25. LCA onde os nós estão em blocos diferentes, como no caso (1).

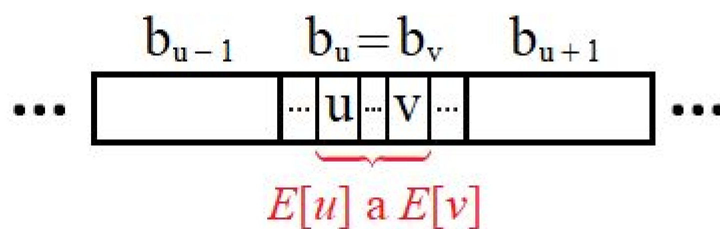

Figura 2.26. LCA onde os nós estão no mesmo bloco, como no caso (2).

No caso de blocos diferentes, já somos capazes de responder o RMQ 1.b, pois a *Sparse Table* do vetor R consegue nos retornar, em O(1), o RMQ de uma sequência qualquer de blocos (inteiros).

Para os passos 1 e 3 do primeiro caso e para o segundo caso, devemos ser capazes de responder RMQs dentro de blocos. Poderíamos construir uma *Sparse Table* para cada bloco individualmente, porém a complexidade de espaço deixaria de ser linear, pois como cada bloco tem tamanho O(lg(n)), suas respectivas tabelas teriam tamanho O(lg(n) × lg(lg(n))), e como temos O(n / lg(n)) blocos, teríamos espaço O(n × lg(lg(n))).

Apesar de ser uma complexidade competitiva, essa não é a solução ótima em termos assintóticos. Com efeito, podemos obter uma solução ainda melhor com base na seguinte observação:

Lema 2.27. O vetor P (isto é, o vetor de tamanho $2n - 1$ tal que $P[i]$ corresponde à profundidade do nó $D[i]$ na árvore) tem a seguinte propriedade: $P[i] - P[i + 1] = \pm 1$. Isso significa que, para quaisquer dois nós adjacentes no vetor do percurso em profundidade, a diferença de profundidade deles será sempre de 1.

Prova: Essa propriedade é trivial, dado que em qualquer nó u em que o DFP esteja, ou o caminho seguirá para um filho de u ou voltará para o pai de u.

Sendo assim, qualquer bloco de tamanho t pode ser representado por seu valor inicial e por $t - 1$ diferenças entre seus elementos sucessivos, que chamaremos de padrão, conforme a definição 2.28.

Definição 2.28. Seja b um bloco de tamanho t tal que a diferença entre dois elementos adjacentes de b seja sempre ± 1. Seja $b[i]$ o i-ésimo elemento de b. O padrão do bloco b é definido como um vetor B de tamanho $t - 1$ onde $B[i] = b[i + 1] - b[i]$.

Pela definição 2.28, podemos notar que, por um valor inicial s e um padrão B de tamanho $t - 1$, podemos reconstruir um bloco b de tamanho t. Basta, para isso, colocarmos s no início de b e, para cada i tal que $0 < i < t$, teremos $b[i] = b[i - 1] + B[i - 1]$.

Corolário 2.29. Seja b um bloco de tamanho t e padrão B. Seja s o valor inicial de b. Podemos afirmar que, para todo i tal que $0 < i < t$, $b[i] = s + \sum_{j=0}^{i-1} B[j]$.

Exemplo 2.30. Seja $b = (3, 4, 5, 4, 5, 4, 3, 2)$ um bloco de tamanho 8. O padrão de b é o vetor $B = (4 - 3, 5 - 4, 4 - 5, 5 - 4, 4 - 5, 3 - 4, 2 - 3) = (1, 1, -1, 1, -1, -1, -1)$.

Lema 2.31. Se dois blocos b_1 e b_2 de tamanho t têm o mesmo padrão B, então as diferenças entre todos os elementos de mesmo índice de b_1 e b_2 são iguais, isto é, $\forall\, i < t$ $(b_2[i] = b_1[i] + k)$.

Prova: Vamos supor, sem perda de generalidade, que $b_2[0] = b_1[0] + k$. Sabemos, pelo corolário 2.29, que para todo i tal que $0 < i < t$, $b_1[i] = b_1[0] + \sum_{j=0}^{i-1} B[j]$ e $b_2[i] = b_2[0] + \sum_{j=0}^{i-1} B[j]$. Como $b_2[0] = b_1[0] + k$, temos que

$$b_2[i] = (b_1[0] + k) + \sum_{j=0}^{i-1} B[j] \Rightarrow b_2[i] = \left(b_1[0] + \sum_{j=0}^{i-1} B[j] \right) + k \Rightarrow b_2[i] = b_1[i] + k$$

como queríamos demonstrar.

Lema 2.32. Dois blocos de padrões iguais têm os mesmos RMQs.

Prova: Se dois blocos b_1 e b_2 têm o mesmo padrão B, significa que as diferenças entre todos os elementos de mesmo índice de b_1 e b_2 são iguais, isto é, $\forall i\ (b_1[i] - b_2[i] = k)$, como visto no lema 2.31, ou seja, $b_2[i] = b_1[i] + k$. Sendo assim, essa prova resume-se a mostrar que, para uma sequência qualquer de números $a_1, a_2, \ldots, a_t$, se a_i é o menor valor nessa sequência, então $a_i + k$ será o menor valor na sequência $a_1 + k, a_2 + k, \ldots, a_t + k$, o que é trivial, visto que apenas adicionamos um valor constante todos os elementos da sequência, fato que não altera a diferença entre qualquer par de elementos dela, pois é fato que, se $a_i - a_j = (a_i + k) - (a_j + k) \Rightarrow a_i - a_j = a_i + k - a_j - k \Rightarrow a_i - a_j = a_i - a_j$.

Por fim, basta supormos que a sequência $a_1, a_2, \ldots, a_t$ é um intervalo qualquer de b_1 e $a_1 + k, a_2 + k, \ldots, a_t + k$ é o intervalo correspondente em b_1, logo teremos os mesmos RMQs para ambos os blocos.

Pela definição 2.28 e pelos lemas que a seguem, podemos introduzir outro teorema fundamental para essa abordagem:

Teorema 2.33: Existem apenas $O(\sqrt{n})$ vetores na forma $(\pm1, \ldots, \pm1)$ de tamanho $\frac{\lceil lg(n) \rceil}{2} - 1$, que é o tamanho do padrão de um bloco de tamanho $\frac{\lceil lg(n) \rceil}{2}$.

Prova: sabemos que o tamanho do vetor em questão é $\frac{\lceil lg(n) \rceil}{2} - 1$. Sabemos também que cada elemento do vetor pode assumir apenas dois valores: +1 ou –1. O número total de possibilidades de formar vetores de tamanho $\frac{\lceil lg(n) \rceil}{2} - 1$, onde cada elemento pode assumir

apenas dois valores é de $\underbrace{2 \cdot 2 \cdots 2}_{\frac{\lceil lg(n) \rceil}{2} - 1 \text{ vezes}} = 2^{\frac{\lceil lg(n) \rceil}{2} - 1}$.

Temos, portanto, que

$$2^{\frac{\lceil lg(n) \rceil}{2} - 1} = 2^{\frac{\lceil lg(n) \rceil - 2}{2}} = \sqrt{2^{\lceil lg(n) \rceil - 2}} = \sqrt{\frac{2^{\lceil lg(n) \rceil}}{2^2}} = \sqrt{\frac{n}{4}} = \frac{\sqrt{n}}{2} = O(\sqrt{n})$$

como queríamos demonstrar.

Se o tamanho do padrão é $\frac{\lceil lg(n) \rceil}{2} - 1$, uma *Sparse Table* desse vetor teria tamanho

$$\left(\frac{\lceil lg(n) \rceil}{2} - 1\right) \times lg\left(\frac{\lceil lg(n) \rceil}{2} - 1\right) = O(lg(n) \cdot lg(lg(n)))$$, logo, se

pré-processarmos todas as possibilidades de padrões na forma $(\pm1, \pm1, \ldots, \pm1)$ de tamanho $\frac{\lceil lg(n) \rceil}{2} - 1$ possíveis, teremos que a complexidade desse processamento será a complexidade de multiplicar-se o tamanho da *Sparse Table* de um bloco pelo total de blocos, isto é, $O(\sqrt{n} \cdot lg(n) \cdot lg(lg(n)))$, que é menor que $O(n)$.

Para provar formalmente que $O(\sqrt{n} \cdot lg(n) \cdot lg(lg(n)))$ é menor que $O(n)$, basta fazer a simplificação $\cancel{O(\sqrt{n})} \cdot O(lg(n) \cdot lg(lg(n)))$ e $\cancel{O(\sqrt{n})} \cdot O(\sqrt{n})$, de forma que, usando a regra de L'Hospital, teremos $\lim\limits_{n \to \infty} \dfrac{O(\sqrt{n})}{lg(n) \cdot lg(lg(n))} = \infty$.

Temos, finalmente, $O(\sqrt{n})$ *Sparse Tables*. Para cada bloco b do vetor P da DFP, é necessário saber qual dessas *Sparse Tables* é responsável por responder os RMQs de intervalos em b. Sabemos que b é representado por seu valor inicial e um vetor B de ±1s, que é representado por uma das $O(\sqrt{n})$ *Sparse Tables*. Podemos ler B como um número binário, onde –1 é interpretado como 0 e +1 como 1, de forma que a representação binária de B = (–1, +1, +1, –1) é 0110, que corresponde ao decimal 6, ou seja, ele pode ser representado pela 6ª *Sparse Table*.

Por simplicidade, os padrões que serão usados na construção de *Sparse Tables* não precisam estar no formato (±1, ±1, … , ±1). Pode-se utilizar um vetor na forma do bloco original, porém subtraindo-se todas as suas casas pelo seu valor inicial. Tomemos por exemplo o bloco b = (2, 1, 2, 3, 2). Para processar a *Sparse Table* desse bloco, podemos usar como base o vetor (0, –1, 0, 1, 0), que nada mais é do que b, subtraindo-se todos os seus elementos do elemento inicial $b[0] = 2$. Assim, o processo de construir a *Sparse Table* é facilitado, sem que nada seja alterado em termos de complexidade.

Com essas informações em mãos, podemos pré-processar todas as *Sparse Tables* possíveis em tempo linear, de maneira que uma consulta de RMQ dentro de um bloco possa ser realizada em tempo constante pela *Sparse Table* correspondente a ele.

Como forma de otimização, ao invés de pré-processarmos todas as $O(\sqrt{n})$ *Sparse Tables* possíveis, podemos pré-processar apenas as que possivelmente serão usadas.

Exemplo 2.34. Seja b = (2, 3, 4, 3, 2) um bloco qualquer do vetor P. Sabemos que b pode ser representado pelo valor inicial 2 e pelo vetor B = (1, 1, –1, –1).

Para associar o bloco b à sua *Sparse Table*, basta calcular a representação binária de B, que é $1100_2 = 12_{10}$, logo b será representado pela 12ª *Sparse Table*. Sabendo disso, podemos criar um conjunto que guarde quais *Sparse Tables* serão usadas, e no momento em que elas estiverem sendo criadas, será possível pular a construção de uma *Sparse Table* que sabidamente não representa nenhum bloco, ou seja, que não está nesse conjunto. Esse conjunto pode ser uma Tabela de Dispersão (*Hash Table*)[8], que consegue dizer se um valor está contido nele em tempo O(1). Temos, por fim, uma solução ⟨O(n), O(1)⟩ para o problema do LCA. Essa solução envolve uma série de observações e contém procedimentos que tornam sua implementação mais extensa do que a de outras abordagens mais custosas, como a abordagem ⟨O(n lg(n)), O(1)⟩ vista anteriormente.

3. LCA simplificado ⟨O(n), O(lg(n))⟩

Neste capítulo será proposta uma abordagem simplificada que é capaz de responder a maioria dos casos em tempo constante, tendo um pré-processamento da árvore mais rápido e mais econômico em espaço, além de ter uma implementação menor e respostas tão rápidas quanto as da abordagem ótima. Essa abordagem, no entanto, é $\langle O(n), O(\lg(n))\rangle$, ou seja, é logarítmica no pior caso da consulta, mas o pior caso terá baixa probabilidade de ocorrer, como será empiricamente demonstrado posteriormente, de maneira que quanto maior for a árvore, menor a chance de ocorrência do cenário de pior caso da consulta.

Tomemos a árvore da figura 3.1 como exemplo.

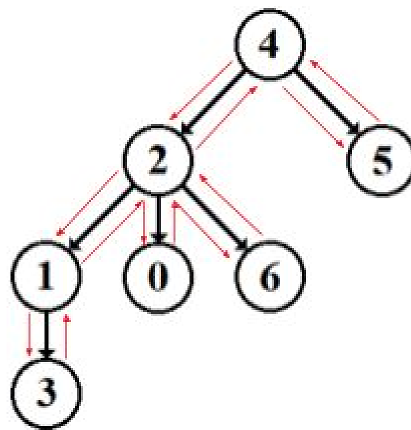

Figura 3.1. Árvore de 7 nós enraizada no nó 4 e seu DFP.

$D =$ (4, 2, 1, 3, 1, 2, 0, 2, 6, 2, 4, 5, 4)
$P =$ (0, 1, 2, 3, 2, 1, 2, 1, 2, 1, 0, 1, 0)
$E =$ (6, 2, 1, 3, 0, 11, 8)

Vamos seguir o mesmo caminho de dividir os vetores em blocos, mas dessa vez, por simplicidade, em blocos de tamanho $\lfloor \lg(2n-1) \rfloor$. A árvore acima tem 7 nós, logo o vetor caminho terá 13 nós, portanto $\lfloor \lg(13) \rfloor = 3$. Temos então a divisão como na figura 3.2:

D	4 2 1	3 1 2	0 2 6	2 4 5	4 − −
P	0 1 2	3 2 1	2 1 2	1 0 1	0 − −

Figura 3.2. Divisão de *D* e *P* em blocos de tamanho $\lfloor \lg(13) \rfloor = 3$.

Calcula-se, então, o vetor de representantes *R*, tal que *R*[i] é o nó com menor profundidade dentro do *i*-ésimo bloco. O procedimento para construir esse vetor é exibido nos Algoritmo 3.3 e 3.4.

```
Algoritmo Mínimo-no-Bloco
Entrada: l, r (índices de início e fim de um bloco)
Saída: u (nó de menor profundidade no bloco)
Início
        se l = r:
                retorna D[r]
        fim-se
        retorna Nó-Menos-Profundo(D[l], Mínimo-no-Bloco(l + 1, r))
Fim
```

Algoritmo 3.3. Retorna o representante do bloco delimitado por *l* e *r*, ou seja, o nó de menor profundidade entre *l* e *r*.

```
 /* variáveis globais */
R = vetor em branco

Algoritmo Construir-Representantes
Entrada: t (tamanho de um bloco)
Saída: vazia
Início
        R = (0, 0, … , 0)          /* vetor de tamanho ⌈(2n – 1) ÷ t⌉ */
        para i = 0, … , |R| – 1:
                R[i] = Mínimo-no-Bloco(i × t, mínimo(i × t + t – 1, 2n – 2))
        fim-para
Fim
```

Algoritmo 3.3. Retorna o representante do bloco delimitado por *l* e *r*, ou seja, o nó de menor profundidade entre *l* e *r*.

Tendo o vetor de R calculado, o próximo passo do procedimento é montar uma *Sparse Table* tendo-o por base. A construção dessa *Sparse Table* foi mostrada no Algoritmo 2.20.

Com essa *Sparse Table*, é possível saber o RMQ de qualquer sequência de blocos desejada, desde de que os blocos estejam inteiros.

Para calcularmos o LCA entre dois nós quaisquer u e v, o primeiro procedimento é encontrar em quais blocos estão suas primeiras ocorrências. O bloco b_u onde se encontra o nó u pode ser calculado por $b_u = \lfloor(E[u] \div \lfloor\lg(2n - 1)\rfloor)\rfloor$, assim como o bloco b_v onde se encontra o nó v pode ser calculado por $b_v = \lfloor(E[v] \div \lfloor\lg(2n - 1)\rfloor)\rfloor$. Como exemplo, vamos calcular os blocos dos nós $u = 2$ e $v = 5$ da árvore da figura 3.1.

Temos que o tamanho do bloco é $\lfloor\lg(13)\rfloor = 3$. Sabemos que a primeira ocorrência de u é na posição $E[2] = 1$ e a primeira ocorrência de v é na posição Ocorrência[5] = 11. Temos então

- $b_u = \lfloor(E[u] \div \lfloor\lg(2n - 1)\rfloor)\rfloor = \lfloor 1 \div 3\rfloor = 0$.
- $b_v = \lfloor(E[v] \div \lfloor\lg(2n - 1)\rfloor)\rfloor = \lfloor 11 \div 3\rfloor = 3$.

Tendo os blocos calculados, para calcular o LCA de u e v, há duas possibilidades, análogas à da abordagem ótima, que repetimos abaixo:

1. Se $b_u \neq b_v$, ou seja, quando u e v caem em blocos diferentes, o LCA de u e v será o nó de menor profundidade entre 3 nós:
    a. O RMQ de $E[u]$ até o fim de b_u.
    b. O RMQ entre os blocos $b_u + 1$, $b_u + 2$, … , $b_v - 1$.
    c. O RMQ do início de b_v até $E[v]$.
2. Se $b_u = b_v$, ou seja, quando u e v caem no mesmo bloco, o LCA de u e v será o nó de menor profundidade entre os índices $E[u]$ e $E[v]$ dentro de b_u.

Vamos tratar do caso $b_u \neq b_v$. É notável que, até o momento, não definimos como calcular o RMQ dentro de um único bloco. No geral, precisamos de três informações:

a. O RMQ de $E[u]$ até o fim de b_u.
b. O RMQ entre os blocos $b_u + 1$, $b_u + 2$, … , $b_v - 1$.
c. O RMQ do início de b_v até $E[v]$.

A informação (b) já é fornecida pela *Sparse Table* de R, logo nos falta as informações (a) e (c).

Para termos essas informações, vamos construir dois vetores de tamanho $2n - 1$, chamados M_p e M_a , cujos significados são:

- $M_p[u]$ é o nó de menor profundidade entre u e o fim de b_u .
- $M_a[u]$ é o nó de menor profundidade entre o início de b_u e u.

Considerando M_p e M_a como variáveis globais, suas construções são exibidas nos Algoritmos 3.4, 3.5 e 3.6.

```
Algoritmo Construir-Ma
Entrada: l, r (índices de início e fim de um bloco)
Saída: vazia
Início
    Ma[l] = D[l]
    para i = l + 1, … , r:
        Ma[i] = Nó-Menos-Profundo(Ma[i – 1], D[i])
    fim-para
Fim
```

Algoritmo 3.4. Construção do vetor de mínimos anteriores M_a .

```
Algoritmo Construir-Mp
Entrada: l, r (índices de início e fim de um bloco)
Saída: vazia
Início
    Mp[r] = D[r]
    para i = r – 1, … , l:
        Mp[i] = Nó-Menos-Profundo(Mp[i + 1], D[i])
    fim-para
Fim
```

Algoritmo 3.5. Construção do vetor de mínimos posteriores M_p .

```
/* variáveis globais */
M_a , M_p = (0, … , 0)      /* vetores de tamanho 2n – 1 */

Algoritmo Construir-Mínimos
Entrada: t (tamanho de um bloco); n (número de nós da árvore);
Saída: vazia
Início
      para i = 0, … , |R| – 1:          /* |R| é o número de blocos */
            Construir-M_a(i × t, mínimo(i × t + t – 1, 2n – 2))
            Construir-M_p(i × t, mínimo(i × t + t – 1, 2n – 2))
      fim-para
Fim
```

Algoritmo 3.6. Algoritmo que calcula os limites de cada bloco para invocar as construções de M_a e M_p .

Com esses dois vetores calculados, o LCA de u e v será o nó de menor profundidade entre os nós $M_p[u]$, RMQ-Sparse-Table($b_u + 1$, $b_v - 1$) e $M_a[v]$.

Já para o caso $b_u = b_v$, não haverá nenhum procedimento especial. Como o tamanho do bloco é de $\lfloor \lg(2n - 1) \rfloor$, uma iteração entre E[u] e E[v] está limitada superiormente por O(lg(n)) visitas.

Proposição 3.7. O pior caso do LCA simplificado, ou seja, quando os nós estão no mesmo bloco, torna-se mais difícil de ocorrer com o aumento do tamanho da árvore.

Prova: Seja Z um vetor de tamanho $x = 2n - 1$, dividido em blocos de tamanho $\lfloor \lg(t) \rfloor$. Escolhido um índice qualquer de Z, a probabilidade de se escolher outro índice de Z no mesmo bloco é de $\frac{\lceil lg(x) \rceil}{x}$. Quando n tende ao infinito, x também tende ao infinito, logo a chance de dois índices estarem no mesmo bloco é de

$$\lim_{x\to\infty} \frac{lg(x)}{x} = \lim_{x\to\infty} \frac{\left(\frac{1}{x\ \ln(2)}\right)}{1} = \lim_{x\to\infty} \frac{1}{x\ \ln(2)} = \frac{1}{\ln(2)} \cdot \lim_{x\to\infty} \frac{1}{x} = \frac{1}{\ln(2)} \cdot 0 = 0$$

Contudo, essa prova é empírica, visto que as árvores não têm a obrigação de garantir que a probabilidade de ocorrência de dois nós no mesmo bloco é, de fato, $\frac{\lceil lg(x) \rceil}{x}$.

Finalmente, temos tudo que é necessário para realizar o LCA simplificado, que será demonstrado no Algoritmo 3.8. É importante salientar que os vetores D, P, E, R, M_a , M_p e a tabela S (referente à *Sparse Table* de R) são todos variáveis globais, para simplificar as funções e evitar o excesso de parâmetros.

```
Algoritmo LCA-Simplificado
Entrada: u, v (nós)
Saída: LCA(u,v)
Início
    posição_u = mínimo(E[u], E[v])    /* garantir que u ocorre antes */
    posição_v = máximo(E[u], E[v])    /* garantir que v ocorre depois */
    bloco_u = posição_u / lg(2n – 1)                 /* n é o tamanho de árvore */
    bloco_v = posição_v / lg(2n – 1)
    se bloco_u = bloco_v:                            /* caso 1: no mesmo bloco */
        lca = D[posição_u]
        para i = posição_u + 1, … , posição_v:
            lca = Nó-Menos-Profundo(lca, D[i])
        fim-para
        retorna lca
    fim-se
    lca = Nó-Menos-Profundo(M_p[posição_u], M_a[posição_v])
    se bloco_u + 1 = bloco_v:                        /* caso 2: blocos adjacentes */
        retorna lca
    fim-se                                           /* caso 3: blocos distantes */
    retorna Nó-Menos-Profundo(lca, RMQ-Sparse-Table(bloco_u + 1, bloco_v – 1))
Fim
```

Algoritmo 3.8. LCA simplificado.

O caso de dois nós caírem em blocos adjacentes é tratado de forma isolada, pois não há blocos entre eles, o que impossibilita uma consulta de RMQ.

Teorema 3.9. Para calcular-se o LCA dos nós $\{x_1, x_2, \ldots, x_k\}$ para $k \geq 2$, basta computar LCA(x_a, x_b), onde x_a e x_b são tais que $E[x_a] \leq E[x_i]$ e $E[x_i] \leq E[x_b]$ $\forall i \in [1, k]$.

Prova. Seja w o LCA de x_a e x_b. Pelo teorema 2.16, temos que w será o nó de menor profundidade entre $E[x_a]$ e $E[x_b]$. Como $E[x_a]$ e $E[x_b]$ estão nas extremidades, isto é, como $E[x_a] \leq E[x_i]$ e $E[x_i] \leq E[x_b]$ $\forall i \in [1, k]$, significa que w será menos (ou igualmente) profundo que qualquer outro nó contido no intervalo ($E[x_a]$, $E[x_b]$), logo ele será o LCA de todos os nós (simultaneamente) nesse intervalo. Como todos os nós x_i estão dentro do intervalo ($E[x_a]$, $E[x_b]$), então w também será ancestral deles.

Por fim, será apresentada uma análise comparativa entre as abordagens vistas acima, ou seja, a ótima $\langle O(n), O(1)\rangle$ e a simplificada $\langle O(n), O(\lg(n))\rangle$.

4. Análise experimental

Neste capítulo, será realizada uma análise comparativa entre a abordagem ótima[9] e a abordagem simplificada[10] do LCA, com o intuito de medir os tempos de pré-processamento, consulta e consumo de memória da ambas.

Para essa análise, a metodologia de teste foi feita da seguinte maneira:

- Foram geradas várias árvores completas de alturas variando de 0 até 14, ou seja, entre $2^1 - 1$ e $2^{15} - 1$ (=32767) nós;
- Para cada árvore de altura h, foram executados os LCAs de todos os possíveis pares de nós, que são $2^h (2^{h+1} - 1)$, variando de 1 a 536.854.528 consultas de LCAs;
- Calcula-se o tempo total t de todas as $2^h (2^{h+1} - 1)$ consultas juntas;
- Divide-se t pelo número de consultas para obter-se o tempo médio por consulta.

Os testes foram realizados numa máquina com sistema operacional Windows 10, 64 bits, processador Intel(R) Core(TM) i5-3330S CPU @ 2.70GHz (4 CPUs), ~3.2GHz, 8GB de memória RAM.

4.1 Tempo de pré-processamento

O pré-processamento foi medido marcando-se o tempo consumido para a construção de todos os vetores e tabelas necessários em cada abordagem para cada árvore, cujo resultado está no gráfico da figura 4.1.

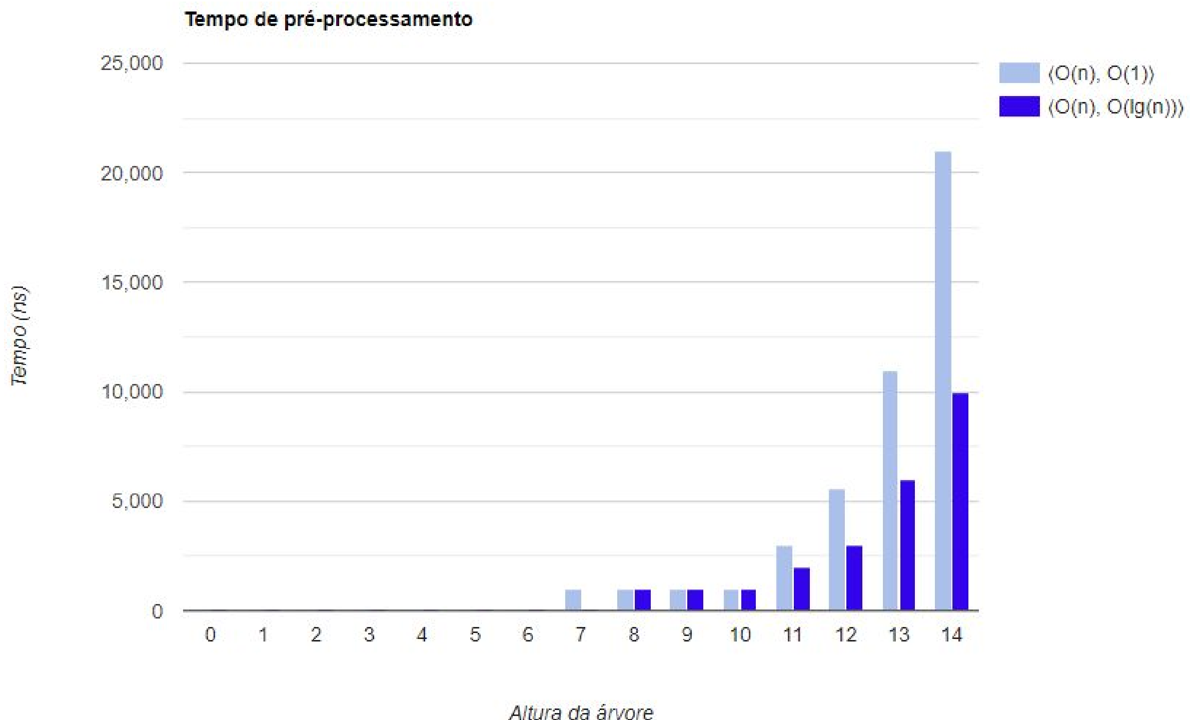

Figura 4.1. Gráfico da comparação entre os tempos de pré-processamento, em nanossegundos.

O tempo de pré-processamento da abordagem ótima é pior, devido ao fato de ter de pré-processar as *Sparse Tables* dos blocos, enquanto que a abordagem simplificada não

tem esse requisito. O pré-processamento simplificado é aproximadamente 1.8 vezes mais ágil.

4.2 Tempo de consulta

Para cada árvore de altura h, foram realizados todos os possíveis $2^h (2^{h+1} - 1)$ LCAs nessa árvore, cujo tempo total dividido pela quantidade de LCAs resulta no tempo médio por consulta, como exibido no gráfico da figura 4.2.

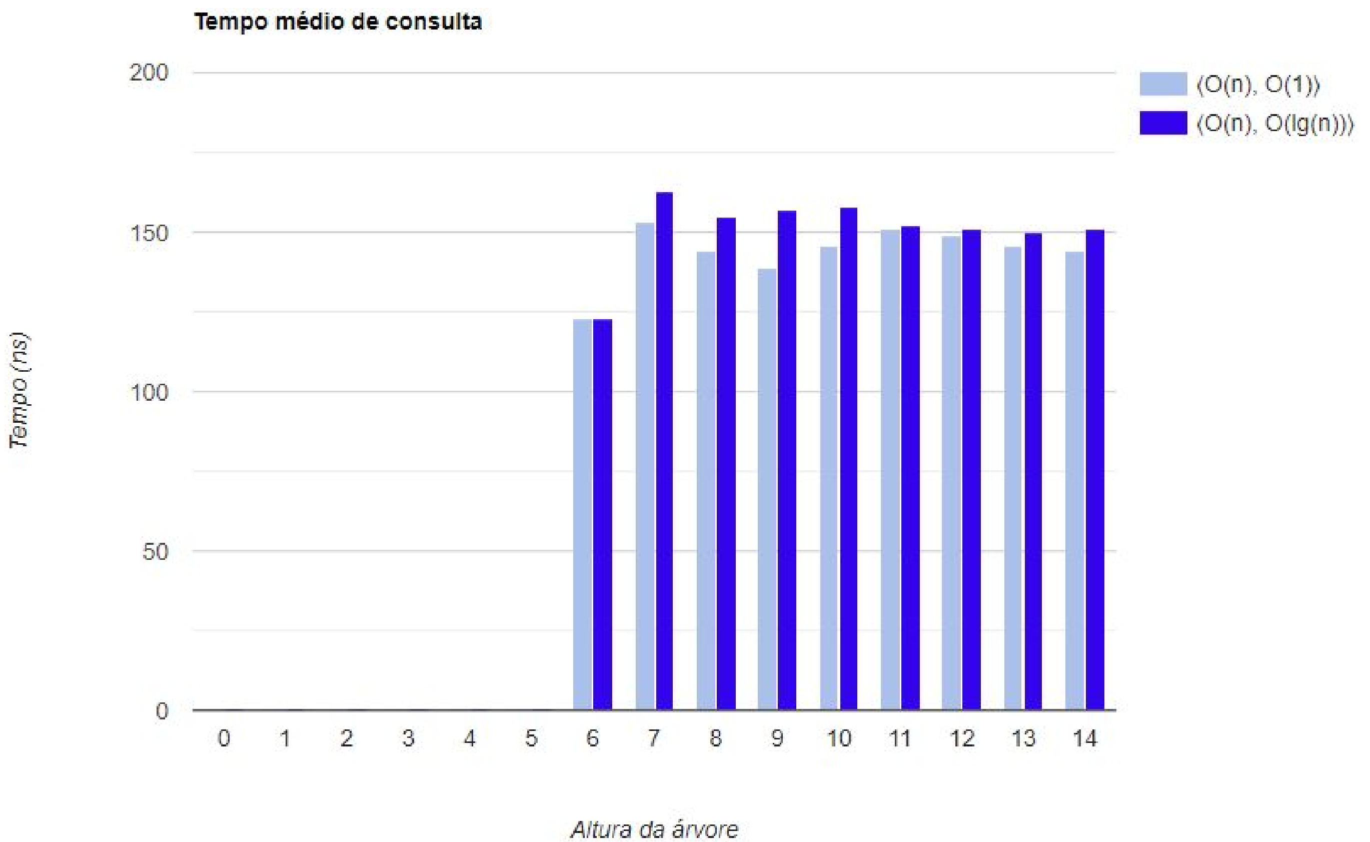

Figura 4.2. Gráfico de comparação entre tempos médios de consulta, em nanossegundos.

Os resultados indicam que o algoritmo simplificado responde os LCAs com tempo muito aproximado ao algoritmo ótimo. Isto se deve ao fato de que, para a grande maioria dos LCAs, ambos realizam o mesmo número de operações, que são as consultas do RMQ de $E[u]$ até o fim de b_u, do RMQ dos blocos $b_u + 1$ a $b_v - 1$ e o RMQ do início de b_v até $E[v]$.

Nos casos onde os nós estão em blocos separados, o algoritmo simplificado é ligeiramente mais rápido, pois evita algumas consultas (como quais tabelas respondem os blocos de cada um dos nós). Nos casos onde os nós estão no mesmo bloco, o algoritmo ótimo é ligeiramente mais rápido, pois não necessita fazer uma pesquisa sequencial de complexidade $O(\lg(n))$, diferentemente do algoritmo simplificado.

Tendo essas informações, é possível compreender o motivo pelo qual as abordagens apresentaram tempo tão semelhante em consulta, sendo o algoritmo ótimo 1.05 vezes mais rápido que o simplificado.

4.3 Uso de memória

Para essa análise, foi contada a quantidade de números inteiros (de 4 bytes cada) usada por cada abordagem.

Na abordagem ótima, foram contabilizados os vetores *D*, *P*, *E*, *R*, outras estruturas como o mapa que associa blocos às suas respectivas *Sparse Tables*, a tabela *S* e as $O(\sqrt{n})$ *Sparse Tables* de cada bloco possível.

Na abordagem simplificada, foram contabilizados os vetores *D*, *P*, *E*, *R*, M_a, M_p e a tabela *S*.

O resultado pode ser visualizado no gráfico da figura 4.3.

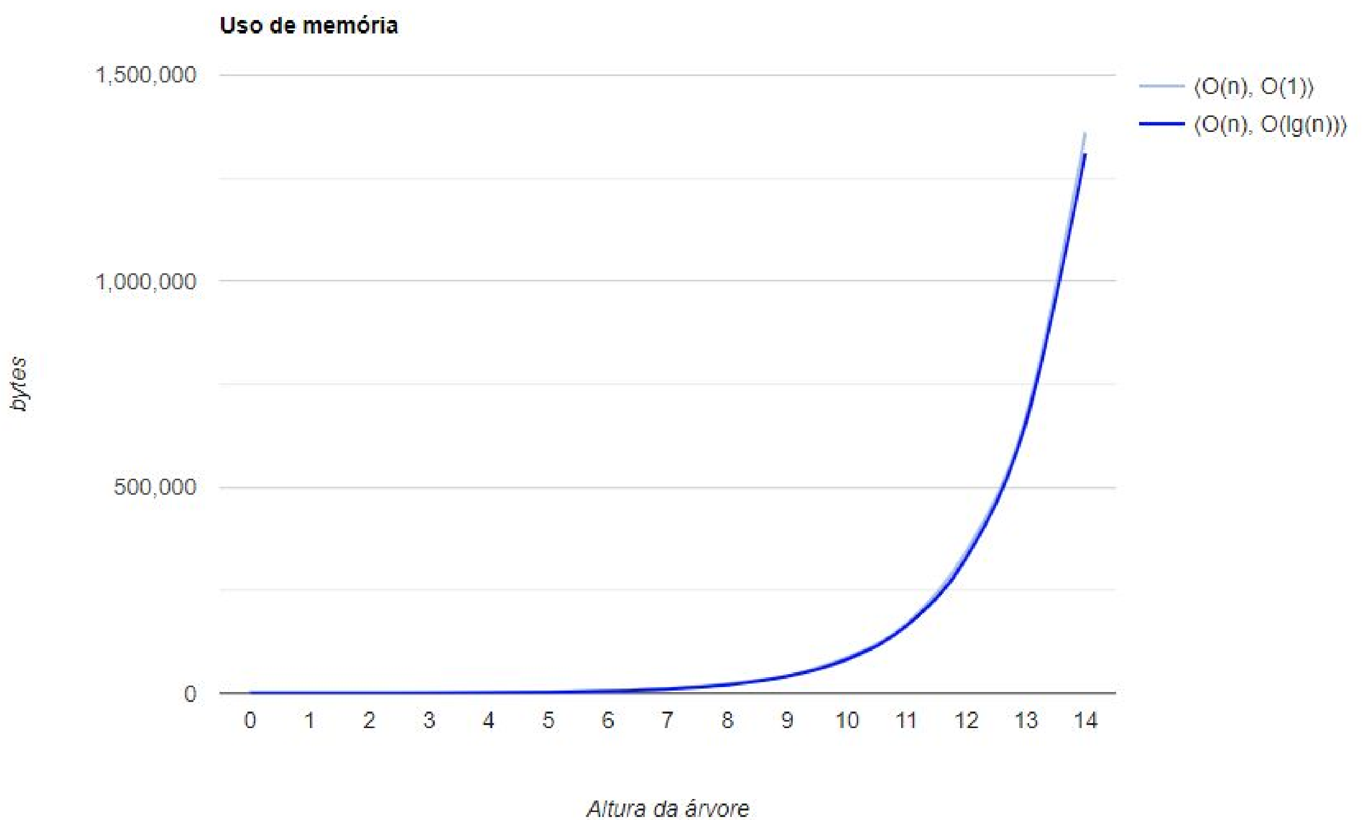

Figura 4.3. Gráfico de comparação de uso de memória.

Como ambas as abordagens são lineares em espaço, o gráfico da figura 4.3 ilustra a similaridade no consumo de memória. A abordagem ótima usa aproximadamente 1.03 vezes mais memória que a abordagem simplificada.

É possível notar uma certa dissonância entre o tempo de pré-processamento e o consumo de memória, pois, como os consumos são muito parecidos, é intuitivo crer que os tempos de pré-processamentos também o sejam, mas o diferencial é que, mesmo sendo linear, a abordagem ótima requer o pré-processamento de todos os blocos possíveis, além de ser necessário calcular qual tabela representa cada bloco, por isso a abordagem ótima demora mais para ser pré-processada.

5. Conclusão

Durante este trabalho, foram abordados a solução ótima para o LCA com apenas uma consulta, a redução do problema LCA para o problema RMQ, a solução ótima para o LCA e uma alternativa simplificada para resolvê-lo, sendo, finalmente, realizada a comparação empírica entre ambas.

Após os testes, é possível concluir que, até um número de nós razoavelmente grande, é vantajoso utilizar a abordagem simplificada, tanto pela implementação mais curta quanto pela melhor eficiência em todos os resultados. Teoricamente, a abordagem ótima será mais lenta até que a quantidade de nós seja grande o suficiente para que o logaritmo na base 2 dessa quantidade seja algo computacionalmente custoso. Vamos supor que, no pior caso, que é o cenário onde os dois nós de entrada do LCA estão no início e no final de um mesmo bloco, seja necessário percorrer 100 nós ao todo. Sabemos que um bloco tem tamanho $\lg(2n - 1)$, logo é fácil notar que se $\lg(2n - 1) = 100$, então o número n de nós da árvore será em torno de 2^{99}, isto é, algo próximo de 633.825.300.114.114.700.748.351.602.688, o que é um cenário escasso na vida real, ou seja, mesmo que a árvore tenha uma quantidade muito grande de nós, não será suficiente para tornar a abordagem simplificada lenta, além do fato de que ela se beneficia do aumento do número de nós, pois, como visto na proposição 3.7, quanto maior for esse número, menor a probabilidade de ocorrência do pior caso, onde, se evitado, tem-se uma consulta em tempo constante.

Como a abordagem simplificada é linear em tempo de pré-processamento, considerando que um número inteiro equivale a 4 bytes, uma árvore de aproximadamente 10^7 nós já seria suficiente para ocupar algo em torno de 1 GB de memória (supondo que os nós da árvore possam ser representados por números inteiros), que já consome uma boa parcela de memória e processamento de um computador comum.

Bibliografia

[1] Stroustrup, B. The C++ Programming Language, Fourth Edition. Addison-Wesley, June 2013.

[2] Tree (data structure). Disponível em: https://en.wikipedia.org/wiki/Tree_(data_structure). Acesso em: 21 jun. 2019.

[3] Phylogenetic tree. Disponível em: https://en.wikipedia.org/wiki/Phylogenetic_tree. Acesso em 21 jun. 2019.

[4] Ruohonen, K. Graph Theory. Página 57. Disponível em http://math.tut.fi/~ruohonen/GT_English.pdf. 2013.

[5] Global Variable. Disponível em: https://en.wikipedia.org/wiki/Global_variable. Acesso em 21 jun. 2019.

[6] Chen, B.; Paterson, M.; Zhang, G. Combinatorics, Algorithms, Probabilistic and Experimental Methodologies. Página 459. Springer, April 2007.

[7] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. Journal of Algorithms, 57(2):75–94, nov 2005.

[8] Hash table. Disponível em https://en.wikipedia.org/wiki/Hash_table. Acesso em 21 jun. 2019.

[9] Lowest Common Ancestor - Farach-Colton and Bender Algorithm. Disponível em http://e-maxx.ru/algo/lca_linear. Acesso em 21 jun. 2019.

[10] Lowest Common Ancestor simplificado. Disponível em https://github.com/DcCoO/TCC/blob/master/Codigo/%3CO(n),%20O(lg(n))%3E.cpp. Acesso em 21 jun. 2019.