Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Pedro Vítor Bortolli Santos

Ancestral comum mais próximo entre dois vértices de uma árvore

São Paulo
Novembro de 2019

Ancestral comum mais próximo entre dois vértices de uma árvore

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Carlos Eduardo Ferreira

São Paulo
Novembro de 2019

Resumo

Neste trabalho estudamos o problema de encontrar o ancestral comum mais próximo entre dois vértices de uma árvore. Este problema, que tem aplicações importantes em teoria dos grafos, também exige o conhecimento de diversas técnicas importantes de Ciência da Computação, como programação dinâmica e algoritmos de busca em grafos. O foco é apresentar os algoritmos de forma didática a fim deste material servir como um texto para pessoas interessadas no tema - em especial aquelas envolvidas em programação competitiva.

Palavras-chave: ancestral comum mais próximo, grafos, árvores.

Abstract

In this monograph we will study the problem of finding the lowest common ancestor between two vertices of a tree. This problem, which has important applications on graph theory, also requires knowledge on many important techniques of Computer Science, such as dynamic programming and graph search algorithms. The focus is to present algorithms in an educational way so that this paper serves as a guide for those interested on this subject - especially those involved in competitive programming.

Keywords: lowest common ancestor, graphs, trees.

Sumário

1 Introdução **1**

1.1 Organização do texto . . . 1

1.2 Complexidade dos algoritmos estudados . . . 2

2 Conceitos importantes **3**

2.1 Grafo . . . 3

2.2 Caminho . . . 3

2.3 Descendente e ancestral . . . 3

2.4 Árvore . . . 3

2.5 Subárvore . . . 4

2.6 Profundidade . . . 4

2.7 Altura . . . 4

2.8 Arestas direcionadas . . . 4

2.9 Complexidade de um algoritmo . . . 4

2.10 Dicionário . . . 4

3 Visão geral **5**

3.1 Descrição . . . 5

3.2 Unicidade . . . 6

3.3 Algoritmo simples . . . 6

3.3.1 Corretude . . . 7

3.3.2 Complexidade . . . 7

4 Decomposição em Baldes **9**

4.1 Introdução . . . 9

4.2 Otimização . . . 9

4.2.1 Detalhando o algoritmo . . . 9

4.3 Código . . . 11

4.3.1 Corretude . . . 12

4.3.2 Complexidade . . . 12

5 Programação Dinâmica **15**
5.1 Introdução 15
5.2 Descrição 15
5.3 Algoritmo 18
5.4 Corretude 19
5.4.1 Pré-processamento 19
5.4.2 LCA 19
5.5 Complexidade 20

6 Passeio de Euler **21**
6.1 Descrição 21
6.2 Obtendo todos os vértices de qualquer caminho 22
6.2.1 Passeio de Euler 22
6.3 Otimização com árvore de segmentos 24
6.4 Código 24
6.5 Corretude 26
6.6 Complexidade 27

7 Resolução de problemas **29**
7.1 ANTS10 - Colônia de Formigas (SPOJ Brasil) 29
7.2 QTREE2 - Query on a tree II (SPOJ) 34

8 Conclusões **39**

Referências Bibliográficas **41**

Capítulo 1

Introdução

Neste trabalho será abordado um importante tópico da Teoria de Grafos: o ancestral comum mais próximo - ou simplesmente LCA (derivado do inglês *Lowest Common Ancestor*).

A motivação para o estudo este assunto vem majoritariamente da Maratona de Programação. Nestas competições os participantes são expostos a desafios lógicos que requerem a escrita de códigos que, para uma entrada de um problema, são capazes de retornar sua resposta esperada. Um ótimo material (porém em Inglês) sobre LCA utilizado por muitos competidores para aprender mais sobre este problema é o tutorial do website TopCoder (1).

Entender o problema de encontrar o LCA entre dois vértices de uma árvore não é uma tarefa complicada. Ademais, escrever um algoritmo simples para tal também não possui um grau de complexidade elevado. Entretanto, nessas competições sempre procuramos resolver os problemas de forma eficiente, visto que o tamanho da entrada pode ser grande o suficiente para exigir um algoritmo rápido (e, geralmente, mais complexo) que termine sua execução dentro do tempo limite proposto.

Diversos algoritmos são conhecidos para encontrar o LCA - uns mais eficientes do que outros. Ao longo deste trabalho me aprofundarei em algumas soluções, partindo de algoritmos mais simples porém ineficientes até uma solução eficiente e mais complexa.

O trabalho será abordado com viés didático, visando o aprendizado do leitor. Para isso, ao apresentar algoritmos sempre procuro fornecer códigos e, no final da monografia, resolver alguns problemas interessantes de juízes *online* que exploram o tema estudado.

1.1 Organização do texto

O capítulo 2 aborda conceitos importantes para o entendimento do trabalho.

O capítulo 3 descreve o problema do LCA e apresenta um algoritmo simples (porém ineficiente) para resolvê-lo.

O capítulo 4 apresenta uma otimização do algoritmo inicial mostrado.

O capítulo 5 aborda o problema usando a famosa técnica chamada de Programação

Dinâmica. A complexidade obtida neste algoritmo é a desejável para este trabalho.

O capítulo 6 mostra como o problema pode ser resolvido quando reduzido ao problema de achar o elemento mínimo em um intervalo. Além da complexidade ótima obtida, esta solução também possibilita atualizar os valores dos vértices da árvore.

No capítulo 7 alguns problemas de juízes online selecionados são resolvidos, para demonstrar a aplicação prática dos algoritmos estudados neste trabalho.

No capítulo 8 é feita a conclusão, apresentando os resultados obtidos com o trabalho.

1.2 Complexidade dos algoritmos estudados

Durante o trabalho serão estudados quatro algoritmos. A complexidade de tempo de cada um pode ser visualizada na tabela abaixo:

Algoritmo	**Complexidade**
Algoritmo simples	$O(n)$
Descomposição em Baldes	$O(\sqrt{n})$
Programação dinâmica	$O(log\ n)$
Passeio de Euler	$O(log\ n)$

Capítulo 2

Conceitos importantes

Neste capítulo serão apresentados brevemente alguns conceitos de suma importância para a compreensão do trabalho. Entretanto, já é suposto que o leitor tenha um conhecimento básico sobre os temas listados a seguir. O leitor que não estiver familiarizado com tais assuntos deve buscar um texto básico, como o livro introdutório a algoritmos escrito por Cormen(2).

2.1 Grafo

Conjunto de vértices (também chamados de nós) conectados por arestas.

2.2 Caminho

Sequência de arestas que ligam vértices de um grafo. Em um caminho é possível partir de um vértice a e chegar em um vértice b, usando em cada passo uma aresta cujas pontas são os vértices vizinhos pertencentes ao caminho.

2.3 Descendente e ancestral

Um vértice v é descendente de u se ao percorrer o caminho de v até a raiz da árvore passamos pelo vértice u. Neste caso também dizemos que u é ancestral de v.

2.4 Árvore

Caso especial de um grafo, onde para quaisquer 2 vértices existe apenas um único caminho que os liga no grafo. Dizemos que uma árvore é enraizada se escolhemos um vértice para ser a raiz e todas as arestas partirem dele para seus descendentes (neste sentido).

2.5 Subárvore

Define-se como subárvore de um vértice v o conjunto de vértices e arestas que são seus descendentes (incluindo o próprio vértice v).

2.6 Profundidade

A profundidade de um vértice em uma árvore enraizada é dada pelo tamanho de seu caminho até a raiz - isto é, quantidade de arestas pertencente a este caminho . A profundidade da raiz é sempre 1.

2.7 Altura

A altura de uma árvore equivale à maior profundidade de algum vértice dela. Ou seja, quantos vértices estão no caminho da raiz até alguma folha da árvore.

2.8 Arestas direcionadas

Um grafo pode ter as suas arestas direcionadas ou não. Neste texto, todas as árvores apresentadas são não direcionadas. Entretanto, pode-se sempre fixar um vértice para ser a raiz e criar um direcionamento a partir dele. Assim, todos os algoritmos apresentados neste trabalho funcionam para grafos direcionados e não direcionados.

2.9 Complexidade de um algoritmo

Existem algumas definições diferentes para medir a complexidade de um algoritmo, acompanhadas de notações distintas. Por exemplo, pode-se avaliar um algoritmo por seu melhor caso, médio ou pior. O que mais nos interessa neste trabalho é a análise de pior caso, e para isso usa-se a notação *"Big O"*.

2.10 Dicionário

Conjunto de pares de chaves e valores. Para cada chave, existe apenas um valor associado a ele. Podem ser de qualquer tipo, embora neste trabalho sempre utilizaremos um mapeamento de inteiros para inteiros.

Capítulo 3

Visão geral

3.1 Descrição

Sejam a e b dois vértices (não necessariamente distintos) de uma árvore enraizada. Define-se como LCA entre eles o vértice mais profundo que contém ambos a e b em sua subárvore. Para ilustrar melhor o problema, uma árvore simples é apresentada na figura 3.1.

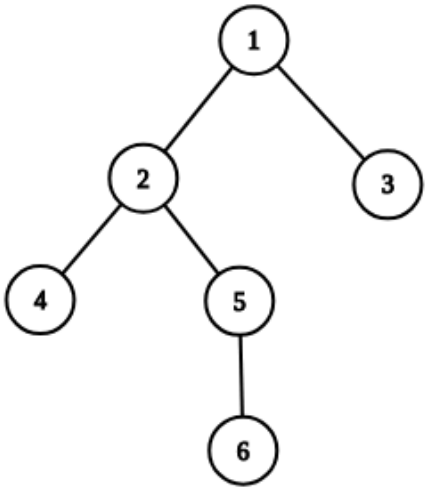

Figura 3.1: *Árvore simples.*

A partir desta árvore, podemos notar que o vértice 2 é o LCA entre 4 e 6. Afinal, os únicos dois candidatos ao LCA destes nós são 1 e 2, já que estes são os dois vértices que contêm tanto 4 quanto 6 em suas subárvores. Entretanto, da definição vem que o LCA é sempre o vértice com maior profundidade na árvore, e neste caso é o 2 (vértice 1 possui profundidade 1, vértice 2 possui profundidade 2).

Um caso que merece destaque é o LCA entre 1 e 3, que é 1. Note que pode ocorrer do LCA entre dois vértices ser um deles.

3.2 Unicidade

Propriedade 3.2.1. Em uma árvore enraizada, o LCA entre dois vértices é sempre único.

Prova: Suponha por absurdo que existam dois vértices x e y que sejam simultaneamente LCA de dois vértices a e b. Da definição de LCA deve valer que a profundidade de x seja igual à profundidade de y. Logo, existem dois caminhos diferentes para chegar na raiz partindo de a: um que passa por x, e outro que passa por y (de forma análoga o mesmo vale para os caminhos que partem de b). Conclui-se que chegamos em uma contradição, pois em uma árvore existe apenas um único caminho da raiz até cada vértice. Portanto, o LCA entre dois vértices em uma árvore é sempre único. A figura 3.2 ilustra um possível grafo com esse cenário.

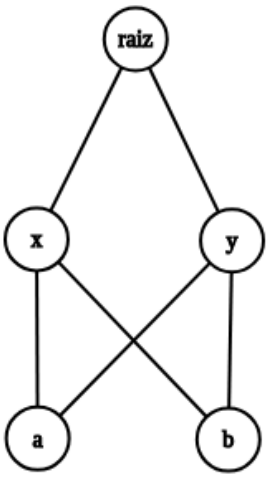

Figura 3.2: *Grafo gerado através da suposição absurda.*

3.3 Algoritmo simples

Agora que sabemos que para todo par de vértices em uma árvore enraizada sempre existe um único LCA vamos introduzir um algoritmo simples inicial para calculá-lo.

Inicialmente, usaremos um dicionário para guardar a profundidade de cada vértice em relação à raiz. Para preencher este dicionário (que será global e chamado de $profundidade$), basta executar uma busca em profundidade (*DFS - Depth First Search*) a partir da raiz:

Algoritmo 1 Cálculo da profundidade de cada vértice usando uma *DFS*

```
1: função CalculaProfundidade(vertice, nivel)
2:     profundidade[vertice] ← nivel
3:     para cada filho em filhos[vertice] faça
4:         CalculaProfundidade(filho, nivel + 1)
```

Deve-se inicializar o dicionário global $profundidade$ com zeros (vamos usar este valor como um indicador de que a profundidade de um vértice ainda não foi calculada) e chamar a função com os argumentos $(raiz, 1)$.

A complexidade de tempo de uma busca em profundidade é $O(n + m)$, onde n é a quantidade de vértices e m a quantidade de arestas. A complexidade de espaço é $O(n)$.

Agora, uma vez chamada a função $CalculaProfundidade$, considere o seguinte algoritmo que encontra o LCA entre dois vértices a e b:

Algoritmo 2 Determina o LCA entre dois vértices

```
1: função LCA_SIMPLES(a, b)
2:     se profundidade[a] < profundidade[b] então
3:         troca(a, b)
4:     enquanto profundidade[a] > profundidade[b]
5:         a ← pai[a]
6:     enquanto a != b
7:         a ← pai[a]
8:         b ← pai[b]
9:     devolve a
```

3.3.1 Corretude

Nosso algoritmo parte do pressuposto de que a e b estão na mesma árvore.

O código fixa o vértice a para ser mais (ou pelo menos igualmente) profundo do que b (linhas 2 e 3). A partir disto, sabemos que ao sair do primeiro laço teremos que o vértice atualizado a tem profundidade igual à de b. Afinal, no início de cada iteração sempre vale que $profundidade[a] - profundidade[b] \geq 0$. Além disso, $profundidade[b]$ não é alterada enquanto $profundidade[a]$ sempre decresce em 1. Assim, quando a execução do laço é interrompida temos que $profundidade[a] - profundidade[b] = 0$, o que implica que $profundidade[a] = profundidade[b]$.

A última etapa do algoritmo parte do princípio de que, como a e b estão na mesma árvore e possuem mesma profundidade, os caminhos para seus pais os levam até a raiz simultaneamente. Assim, como o invariante do segundo laço é $profundidade[a] = profundidade[b]$, e sabemos que a e b estão na mesma árvore, então pela definição encontramos o LCA entre eles por ter sido o primeiro vértice que está tanto no caminho de a quanto de b para a raiz.

3.3.2 Complexidade

Para analisar a complexidade de tempo do nosso algoritmo, vamos verificar quantas operações cada etapa do código executa no pior caso:

- **Linhas 2 e 3**: uma operação cada.

- **Primeiro laço**: sua condição de execução é $profundidade[a] - profundidade[b] > 0$. Para maximizar o valor desta função, é claro que $profundidade[a]$ deve assumir o maior valor possível enquanto $profundidade[b]$ deve assumir o menor. Se a árvore possui n vértices, podemos ter um vértice cuja profundidade é n (a folha da árvore) e outro que tenha profundidade 1 (raiz). Assim, este laço é executado n vezes no pior caso.

- **Segundo laço**: em um pior caso onde ambos a e b são folhas de uma árvore de n vértices, temos que este laço executará n operações para que os caminhos até a raiz sejam percorridos.

Assim, a complexidade de tempo do algoritmo 3 é $O(n)$, onde n é o número de vértices na árvore. Como não utilizamos memória adicional nesta etapa, a complexidade de memória é constante - ou seja - $O(1)$.

Capítulo 4

Decomposição em Baldes

4.1 Introdução

Na seção anterior desenvolvemos um algoritmo simples que percorre, no pior caso, todos os vértices de uma árvore. Neste capítulo apresentaremos uma otimização ao algoritmo anterior para que sua complexidade de tempo seja $O(\sqrt{n})$.

4.2 Otimização

A ideia do algoritmo que vamos desenvolver é a mesma descrita no Algoritmo 3. Isto é, para dois vértices a e b ainda vamos percorrer seus caminhos até a raiz para determinar o seu LCA.

A diferença é que agora não vamos potencialmente visitar todos os vértices do caminho até encontrar o LCA. Agora, vamos dividir os vértices da árvore em "baldes", para que inicialmente um pulo entre dois membros do caminho seja de um vértice de um balde para um vértice de outro balde, ao invés de ir para o seu pai direto. Assim, um pulo não tem mais tamanho 1, mas sim p, onde p é a altura de um balde.

4.2.1 Detalhando o algoritmo

Assumindo que as profundidades dos vértices já foram calculadas, vamos criar alguns baldes de forma que:

- O balde número 1 tenha todos os vértices de profundidade $[1...p]$;
- O balde número 2 tenha todos os vértices de profundidade $[p+1...2p]$;
- O balde número 3 tenha todos os vértices de profundidade $[2p+1...3p]$;
- etc.

O valor de p é essencial para que a otimização do algoritmo seja feita. Entraremos em mais detalhes em como obtê-lo na seção 4.3.2, onde analisaremos a complexidade do código que desenvolveremos.

Em nosso algoritmo precisamos guardar, para cada vértice de um balde, quem é o seu ancestral mais profundo do balde anterior. Para melhor ilustrar isso, apresentamos a figura 4.1:

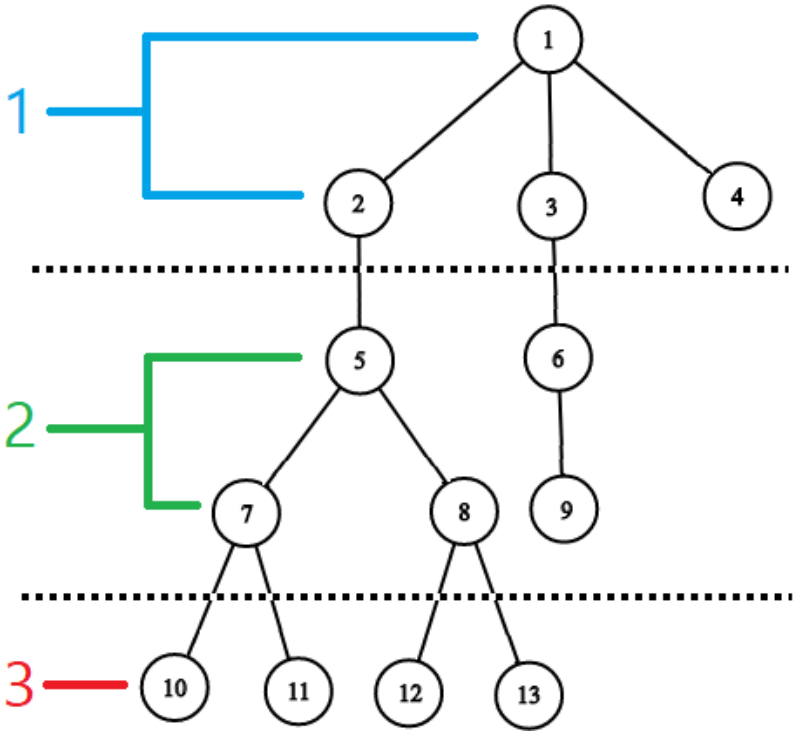

Figura 4.1: *Divisão dos vértices em baldes (1, 2 e 3).*

Neste exemplo temos 5 níveis de profundidades e escolhemos o valor de $p = 2$. Entretanto, como 5 não é divisível por 2, temos que os dois primeiros baldes possuam de fato $p = 2$ níveis de profundidade enquanto o terceiro tem o restante (ou seja, 1).

A partir desta árvore podemos construir tais ancestrais mais profundos do balde anterior do seguinte modo:

Vértice	1	2	3	4	5	6	7	8	9	10	11	12	13
Ancestral	1	1	1	1	2	3	2	2	3	7	7	8	8

Tabela 4.1: *Ancestrais mais profundos do balde anterior para cada vértice*

Note que o ancestral da raiz é naturalmente ela mesma.

Então, para encontrar o LCA entre dois vértices a e b basta verificar em qual balde o LCA está (isto é, percorrer os caminhos de a e b para a raiz até que os ancestrais do balde anterior sejam iguais), e daí trivialmente encontrá-lo.

Por exemplo, se quisermos encontrar o LCA entre os vértices **10** e **13**, faríamos o seguinte:

- Ancestral de 10 é o vértice **7**. Ancestral de 13 é o vértice **8**. Como são vértices distintos, atualizamos nossos vértices para seus ancestrais.

- Ancestral de 7 é o vértice **2**. Ancestral de 8 é o vértice **2**. Como seus ancestrais do balde anterior são o mesmo vértice, sabemos que o LCA está no balde atual (ou então é o próprio vértice 2, que já é comum aos dois).

Agora, basta percorrer os caminhos de **7** e **8** até a raiz naturalmente, dando pulos de tamanho 1 (isto é, indo aos seus pais diretos).

- Pai de 7 é o vértice **5**. Pai de 8 é o vértice **5**. Como seus pais são iguais, podemos afirmar que este é o LCA do nosso problema.

4.3 Código

Utilizaremos os algoritmos 2 e 3 demonstrados anteriormente. Veremos que precisamos de poucas adições neles para que fiquem otimizados.

Como dito anteriormente, além da profundidade, também guardaremos para cada vértice quem é o seu ancestral mais profundo do balde anterior utilizando os vetores globais $profundidade$ e $baldeAnterior$, respectivamente. Além disso, toda vez que o nível atual for múltiplo da altura do balde permitido (p citado anteriormente) "criaremos" um novo balde.

Algoritmo 3 Modificação do algoritmo 2

```
1: função CalculaProfundidade(vertice, nivel, ancestralAnterior)
2:     profundidade[vertice] ← nivel
3:     baldeAnterior[vertice] ← ancestralAnterior
4:     se nivel % alturaBalde = 0 então
5:         anterior ← vertice
6:     senão
7:         anterior ← ancestralAnterior
8:     para cada filho em filhos[vertice] faça
9:         CalculaProfundidade(filho, nivel + 1, anterior)
```

Para a função que determina o LCA, primeiro faremos com que a e b estejam no mesmo balde cujo ancestral anterior seja o mesmo para ambos. Após isso, basta chamar a função de LCA simples, demonstrada no algoritmo 3.

Algoritmo 4 LCA utilizando o conceito de baldes

```
1: função LCA(a, b)
2:     enquanto baldeAnterior[a] != baldeAnterior[b]
3:         se profundidade[a] < profundidade[b] então
4:             troca(a, b)
5:             a ← baldeAnterior[a]
6:     devolve LCA_Simples(a, b)
```

4.3.1 Corretude

Assumindo o algoritmo 4 como correto, a ideia do algoritmo 5 é similar à apresentada no algoritmo 3: após a execução do laço da linha 2 teremos dois vértices no mesmo balde cujos ancestrais do balde anterior são os mesmos. Afinal, sabemos que percorrer o caminho de um vértice pulando de balde em balde resultará no mesmo destino do algoritmo 3, que caminha vértice após vértice: a raiz. Isso se deve ao fato de que no algoritmo 4 apresentado nessa seção, o objeto *baldeAnterior* apenas serve como um encurtador de caminhos - mais precisamente deixando cada pulo em um caminho de tamanho $alturaBalde$ (isto é, a partir de um vértice de profundidade p chegamos em outro de profundidade $p - alturaBalde$).

A linha 5 do código atualiza o valor de a para o próximo vértice do seu caminho encurtado até a raiz. Sabemos que a sempre é mais profundo do que b já que as linhas 3 e 4 trocam os vértices a e b caso o segundo seja mais profundo do que o primeiro.

O código termina com uma chamada da função LCA_Simples já provada no capítulo anterior, que devolve o LCA entre dois vértices de uma árvore enraizada.

4.3.2 Complexidade

O algoritmo 4 executa uma simples busca em profundidade com operações de tempo constante (linhas 2-7) e assim sua complexidade de tempo é $O(n+m)$, onde n é a quantidade de vértices e m a quantidade de arestas. A complexidade de espaço é $O(n)$.

Seja q a quantidade de baldes existentes. No laço das linhas 2-5 do algoritmo 5, sendo q_a e q_b as quantidades de baldes acima de a e b, respectivamente, em seus caminhos até a raiz, podemos dizer que são executadas $q_a + q_b$ operações. No pior caso, $q_a = q_b = q$, e portanto essa etapa do algoritmo é $O(q)$.

A chamada de LCA_Simples tem complexidade $O(h)$, onde h é a altura da árvore (no pior caso a quantidade de vértices $n = h$, então é correto assumir tal complexidade). Entretanto, ao dividir nossa árvore em baldes, garantimos que qualquer balde terá altura $h = alturaBalde = p$.

Assim, o algoritmo 5 tem complexidade de tempo $O(p + q)$. Dado que os valores p e q dependem diretamente da condição de inserção de um vértice em um balde, analisemos agora como fazer isso para atingir a melhor complexidade possível.

Tamanho de um balde

Para que o algoritmo descrito fique de fato mais eficiente do que o do capítulo anterior, devemos escolher bem quantos níveis de profundidade vamos permitir que estejam em um mesmo balde. Ou seja, determinar, para cada balde, o tamanho do intervalo de profundidades que descrevemos anteriormente.

Queremos minimizar a soma $p+q$, (duas fases do algoritmo: avaliar q baldes e percorrer p vértices). Além disso, também vale que $pq = h$, onde h é a altura da árvore. Então, nosso problema resume-se a:

$$\text{minimizar } p+q$$
$$\text{dado que } pq \leq h$$

Já que $p \geq 0$ e $q \geq 0$, temos que $(\sqrt{p} - \sqrt{q}))^2 \geq 0$
$\implies p+q \geq 2\sqrt{pq} \implies p+q \geq 2\sqrt{h}$

Como o objetivo é minimizar $p+q$, queremos que $p+q = 2\sqrt{h}$. Para que isso aconteça, por sua vez, segue que $(\sqrt{p} - \sqrt{q}))^2 = 0 \implies p = q = \sqrt{h}$.

Vale notar que no pior caso a altura h de uma árvore é igual à quantidade n de vértices que ela possui. Portanto, concluímos que o tamanho ótimo de um balde é sempre $\sqrt{n}$ (com $\sqrt{n}$ baldes), e assim a complexidade de tempo do algoritmo é de fato $O(\sqrt{n})$.

Capítulo 5

Programação Dinâmica

5.1 Introdução

Neste capítulo apresentaremos o primeiro algoritmo de complexidade sub-polinomial para calcular o LCA entre dois vértices em uma árvore enraizada. Este algoritmo terá complexidade de tempo $O(log)$ por consulta. A técnica utilizada para isso é programação dinâmica, e para mais informações consulte o material didático elaborado por Stefano Tommasini (3).

5.2 Descrição

Novamente pré-computaremos os ancestrais de cada vértice para poder realizar consultas.

Para cada vértice serão pré-computados os ancestrais cuja profundidade na árvore é menor do que a sua em uma potência de dois - ou seja - 1, 2, 4, 8, ... , h níveis acima dele, onde h é a maior potência de dois que não ultrapasse a altura da árvore. Em outras palavras, para cada vértice serão pré-calculados até $log(n)$ ancestrais diferentes.

Para ilustrar este pré-processamento, introduzimos a figura a seguir:

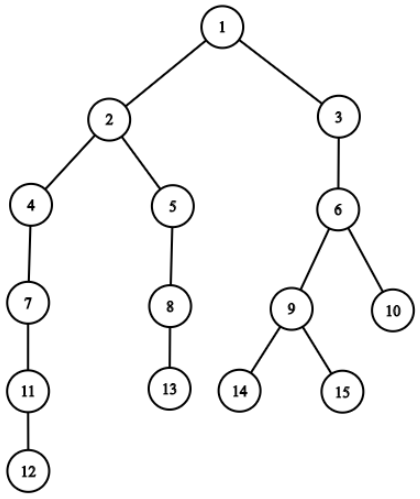

Figura 5.1: *Árvore enraizada de altura 6.*

Após o pré-processamento, teríamos como os ancestrais, para os vértices 11 e 15:

- Vértice 11

Altura relativa	-1	-2	-4
Ancestral	7	4	1

Tabela 5.1: *Ancestrais relativos do vértice 11*

- Vértice 15

Altura relativa	-1	-2	-4
Ancestral	9	6	1

Tabela 5.2: *Ancestrais relativos do vértice 15*

Por conveniência, a partir de agora vamos nos referir a cada ancestral de um vértice v como $ancestral_v(2^i)$ para todo inteiro $i \geq 0$.

Agora, para calcular o LCA entre dois vértices a e b faremos o que já vimos em todos os capítulos anteriores: vamos caminhar até a raiz com ambos. A maneira que faremos isso desta vez é dando pulos de tamanho "potência de dois" com auxílio dos valores que já foram pré-computados. Além disso, cada pulo é de um tamanho diferente.

Vamos assumir, sem perda de generalidade, que b é mais profundo do que a. Inicialmente, são feitos sucessivos pulos suficientes para que a e b estejam exatamente no mesmo nível. Por exemplo, se a profundidade de a é 3 e a de b é 6, devemos subir b em $6 - 3 = 3$ níveis: o que equivale a um pulo de tamanho 2 e um pulo de tamanho 1. Provaremos adiante que sempre conseguimos igualar os níveis dos vértices dando pulos diferentes de tamanho de alguma potência de dois cada.

Tendo a e b agora no mesmo nível, podemos checar se eles são iguais. Se forem, isso significa que a estava originalmente no caminho de b até a raiz e que portanto ela é o LCA. Caso contrário, devemos encontrar o maior tamanho de um pulo para subir simultaneamente os vértices a e b de forma que ainda sejam diferentes. Encontraremos este valor dando pulos distintos de tamanho de alguma potência de dois. A figura a seguir um possível cenário:

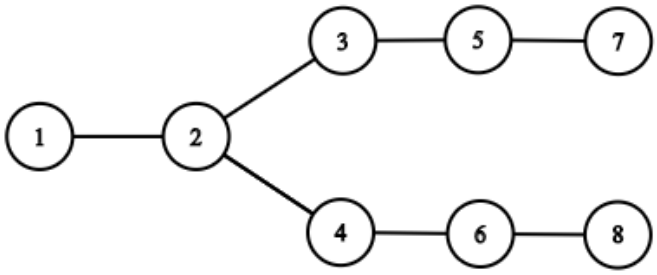

Figura 5.2: *Árvore enraizada de altura 5.*

Consideremos $a = 7$ e $b = 8$. Vamos listar os possíveis valores a serem escolhidos para subir simultaneamente a e b. Note que ao subir $2^3 = 8$ ou mais níveis a restrição da altura máxima da árvore seria violada, e por isso a última potência de dois a ser considerada é $2^2 = 4$.

- **4**: Ao subir 4 níveis, a e b são iguais (ambos = 1). Não o fazemos.
- **2**: Ao subir 2 níveis, a e b são diferentes. Então atualizamos a e b para tais ancestrais. Agora $a = 3$ e $b = 4$
- **1**: Ao subir 1 nível, os valores atualizados de a e b são iguais (ambos = 2). Não o fazemos.

Em outras palavras, o que encontramos são os dois vértices distintos mais próximos à raiz que contêm, respectivamente, os vértices a e b em suas sub-árvores. Assim, o pai destes vértices será o LCA já que será o primeiro vértice que contém ambos a e b em sua sub-árvore.

Voltando à figura 5.1, vamos simular os passos completos para encontrar o LCA(12, 15):

- Vértices 12 e 15 não estão na mesma altura. O vértice 12 é mais profundo, então vamos subir em direção à raiz.
- A diferença das profundidades de 12 e 15 é de um. Então agora seu novo valor é $ancestral_{12}(1)$ = vértice 11.
- Agora, vamos determinar qual é o maior inteiro i tal que $ancestral_{11}(i) \neq ancestral_{15}(i)$. Testando as potências de dois, da maior para a menor, tentaremos primeiro subir 4 níveis - o que resultaria em ambos irem até o vértice 1. Então, testamos 2 níveis - o que resulta nos vértices 4 e 6, respectivamente. Como são distintos, vamos para eles. Finalmente, tentamos subir 1 nível - o que resulta nos vértices 2 e 3 - ainda distintos. Vamos para eles.
- Finalmente, sabemos que os últimos dois vértices antes do LCA são 2 e 3, então basta retornar o pai de qualquer um deles para obter o LCA: o vértice 1.

5.3 Algoritmo

Vamos dividir nosso algoritmo em duas partes: pré-processamento da matriz de programação dinâmica e a função de obtenção do LCA entre dois vértices a e b.

Como explicado na seção 5.2, vamos calcular os ancestrais de "potência de dois" de cada vértice da árvore. Assim, $anc[i][j]$ representa o ancestral 2^j do vértice i.

Primeiro, para o pré-processamento, vamos apresentar a seguinte recorrência:

$$\text{anc[i][j]}=\begin{cases} pai[i], & \text{se } j = 0. \\ anc[anc[i][j-1]][j-1], & \text{caso contrário.} \end{cases}$$

Para preencher a matriz anc, linha a linha ($logNiveis$ = logaritmo da altura da árvore):

Algoritmo 5 Cálculo da matriz de programação dinâmica (pd)

```
1: função CalculaAncestrais()
2:     para cada nivel em logNiveis faça
3:         para cada vertice em vertices faça
4:             se nivel = 0 então
5:                 anc[vertice][nivel] ← pai[vertice]
6:             senão
7:                 anc[vertice][nivel] ← anc[anc[vertice][nivel − 1]][nivel − 1]
```

Agora podemos escrever o código que responderá as consultas de LCA entre a e b:

Algoritmo 6 Obtenção do LCA

```
1: função LCA_PD(a, b)
2:     se profundidade[a] < profundidade[b] então
3:         troca(a, b)
4:     i ← logNiveis
5:     enquanto i ≥ 0
6:         se profundidade[a] − 2^i ≥ profundidade[b] então
7:             a ← anc[a][i]
8:         i ← i − 1
9:     se a = b então
10:        devolve a
11:    i ← logNiveis
12:    enquanto i ≥ 0
13:        se anc[a][i] ≠ anc[b][i] então
14:            a ← anc[a][i]
15:            b ← anc[b][i]
16:        i ← i − 1
17:    devolve pai[a]
```

5.4 Corretude

5.4.1 Pré-processamento

Neste algoritmo é feito o pré-processamento da recorrência apresentada na seção 5.3.

Na primeira iteração do laço mais externo são calculados os *ancestrais* 2^0 de cada vértice (o caso base da recorrência). Note que tal ancestral é seu pai imediato, então podemos facilmente determinar $anc[x][0]$ para todo vértice x. Isto é feito na linha 5 do código.

Em seguida, da segunda iteração em diante do laço mais externo são obtidos os ancestrais "não diretos" (isto é, um ancestral que não é o pai imediato de um nó). Podemos notar que *ancestral* 2^i de um vértice v é o *ancestral* 2^{i-1} do *ancestral* 2^{i-1} de v. Afinal, $2^{i-1} + 2^{i-1} = 2^i$, e por este motivo podemos fazer com que no caminho de um vértice até o seu ancestral em questão visitemos este vértice intermediário. Então, concluímos que a linha 7 produz o resultado correto de $pd[vertice][nivel]$, já que todos os ancestrais $nivel - 1$ já foram computados e só dependemos deles nesta etapa.

5.4.2 LCA

Agora vamos analisar a corretude do algoritmo de obtenção do LCA baseado nos valores pré-computados durante a execução do algoritmo 6.

Nas linhas 2 e 3 os vértices a e b são trocados caso o vértice b seja mais profundo. O motivo disso é que queremos assumir que a profundidade do vértice a sempre seja maior ou igual do que a profundidade do vértice b.

No laço da linha 5 percorremos o caminho do vértice b até a raiz, subindo este vértice até o nó de seu caminho cuja profundidade seja a mesma do vértice a. Sendo $logNiveis$ o valor do *log* da altura da árvore, percorremos este laço de maneira decrescente começando em $logNiveis$. Nas linhas 6 e 7 atualizamos o valor atual do vértice b caso ao subir 2^i níveis a profundidade do novo nó obtido não seja menor do que a do vértice a - afinal, queremos garantir que no final do laço ambos estejam no mesmo nível. Como qualquer número inteiro pode ser representado como uma soma de potências de dois, garantimos que ao final da iteração teremos $profundidade[b] = profundidade[a]$.

Nas linhas 9 e 10 é verificado se $a = b$. Como a e b agora estão na mesma altura da árvore, eles podem coincidir (ou seja, b estava na subárvore de a) - concluindo que obtivemos o LCA e podemos parar o algoritmo.

Caso a execução do código alcance a linha 11, sabemos então que a e b ainda não se encontraram (e portanto ainda não sabemos quem é o LCA). No laço da linha 11 iteramos de $logNiveis$ até 0 para verificar todos os *ancestrais* 2^i de a e b em ordem decrescente. Na linha 12 é verificado se $ancestral_a\ 2^i$ difere de $ancestral_b\ 2^i$. Caso isso se confirme, sabemos que se for dado um pulo de tamanho 2^i ainda não teremos encontrado o LCA, já que se tais ancestrais fossem iguais poderíamos ter encontrado o LCA ou qualquer vértice acima do LCA no caminho da raiz. Então, atualizamos ambos a e b para seus respectivos ancestrais e

repetimos isso até o laço acabar. No final deste laço saberemos que a e b estarão nos vértices menos profundos dos seus caminhos até a raiz que ainda não são o LCA entre eles (pelo mesmo argumento de que conseguimos chegar em qualquer altura dando pulos de tamanho potência de dois). Assim, basta retornar o pai de qualquer um deles que este será o LCA de fato.

5.5 Complexidade

No pré-processamento são executados dois laços encadeados: o mais externo executa $log\ h$ iterações, onde h é a altura da árvore. O laço interno executa n iterações, onde n é a quantidade de vértices da árvore. Assim, como no pior caso $h = n$, concluímos que a função $CalculaAncestrais$ tem complexidade de tempo e de memória ambos $O(n\ log\ n)$.

No cálculo do LCA fazemos operações de tempo constante nas linhas 2, 3 e 4. Os laços das linhas 5 e 11 ambos executam $log\ h$ iterações cada, com h sendo a altura da árvore, e em cada iteração são feitas operações de tempo constante. Novamente, como no pior caso h é igual ao número n de vértices na árvore, concluimos que a função LCA_PD possui complexidade de tempo $O(log\ n)$.

Capítulo 6

Passeio de Euler

Agora que vimos um algoritmo eficiente que resolve o problema do LCA vamos explorar uma outra técnica que tem o mesmo tempo de complexidade. Neste capítulo estudaremos um algoritmo capaz de resolver problemas um pouco mais complexos em que, por exemplo, é necessário atualizar o valor de um nó da árvore rapidamente (diferente dos outros algoritmos em que a árvore deve ser sempre estática - isto é - sem modificações).

6.1 Descrição

Vamos imaginar que temos uma função $F(a, b)$ que devolve todos os vértices no caminho de a para b. Para melhor ilustrar isso, considere a seguinte árvore enraizada:

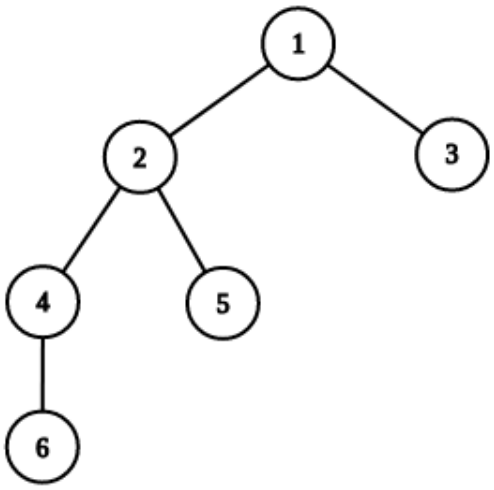

Figura 6.1: *Árvore enraizada de 6 vértices*

Assim, ao chamar $F(2, 3)$ a função nos retorna os vértices 2, 1 e 3.
Ao chamar $F(5, 6)$ os vértices devolvidos são 5, 2, 4 e 6.

Além disso, como de costume vamos também pré-calcular as profundidades de cada vértice de nossa árvore:

Vértice	1	2	3	4	5	6
Profundidade	1	2	2	3	3	4

Tabela 6.1: *Profundidades dos vértices da árvore acima*

Agora, para saber o LCA entre dois nós a e b, usaremos os vértices retornados pela chamada da função $F(a, b)$ para eles (denotaremos o conjunto destes vértices de $P(caminho)$). Afinal, pela definição o LCA(a, b) é o vértice menos profundo que contém a e b em sua subárvore. Isto é, o vértice menos profundo que está no caminho de a até b.

Assim, basta verificar qual é o vértice de menor profundidade dentre os nós de $P(caminho)$: no primeiro exemplo, tal vértice é 1 - enquanto no segundo exemplo é 2. Podemos verificar que eles são, de fato, os respectivos LCAs de seus problemas.

6.2 Obtendo todos os vértices de qualquer caminho

Previamente assumimos que existe uma função F que retorna todos os vértices do caminho entre dois nós a e b de uma árvore. Entretanto, ainda não estudamos como escrever esta função.

Toda vez que queremos fazer uma consulta entre dois vértices a e b, poderíamos percorrer o caminho entre estes dois vértices visitando todos os nós entre eles um a um. Ou seja, um algoritmo linear - ineficiente para nosso problema.

Porém, podemos montar uma estrutura de dados que nos possibilita obter de maneira rápida todos os vértices no caminho entre quaisquer vértices a e b de uma árvore.

6.2.1 Passeio de Euler

O que faremos é "planificar" uma árvore, de forma que tenhamos uma sequência de vértices em uma determinada ordem.

A ideia é rodar uma DFS a partir da raiz contando um tempo crescente conforme a recursão é chamada. Considere o percurso de uma árvore usando uma DFS. Quando um nó é visitado podemos anotar o tempo que isso ocorre, de acordo com os seguintes critérios:

- Anotamos o tempo em que iniciamos a busca em cada filho de um dado vértice x
- Anotamos o tempo em que terminamos a busca de todos os filhos de x

Entender a ideia deste algoritmo é muito mais fácil introduzindo o código e um exemplo. Vamos então apresentar a DFS que executa o algoritmo mencionado. Para cada tempo decorrido guardamos qual foi o vértice visitado naquele momento. Note que é necessário que exista uma variável global *contador* inicializada com 1.

Algoritmo 7 Contando os tempos

```
1: função EULER(vertice)
2:     para cada filho em filhos[vertice] faça
3:         mapa[contador] ← vertice
4:         contador ← contador + 1
5:         Euler(filho)
6:     mapa[contador] ← vertice
7:     contador ← contador + 1
```

Para exemplificar usaremos novamente a árvore introduzida no início deste capítulo, já com os tempos que cada vértice guardou após a execução do código acima:

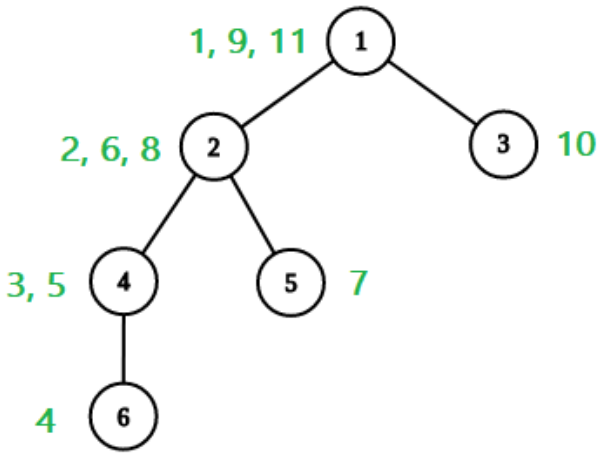

Figura 6.2: *Passeio de Euler na árvore da figura 6.1*

Para fazer uma consulta basta pegar **qualquer** tempo do primeiro vértice e **qualquer** tempo do segundo vértice e retornar todos os vértices cujo tempo esteja contido neste intervalo de tempos.

Vamos simular o resultado da consulta nessa estrutura para os vértices 4 e 5. Vamos escolher o tempo 3 (do vértice 4) e o tempo 7 (do vértice 5). Logo, os vértices retornados são:

$$4, 6, 4, 2, 5$$

Por fim, basta verificar qual desses vértices possui a menor altura - este será o LCA. No exemplo, o vértice com tal propriedade é 2, e de fato podemos verificar que ele é o LCA entre os nós 4 e 5.

Note que podem ser retornados vértices que não pertencem ao caminho entre os dois nós consultados. No exemplo, o vértice 6 não está no caminho entre 4 e 5 mas está contido no intervalo. Entretanto, podemos observar que a profundidade dele é maior do que a de ambos

4 e 5 (afinal, ele é filho de 4) - o que nos diz que sua presença no intervalo não faz diferença para a obtenção do LCA. Provaremos o porquê deste algoritmo sempre funcionar na seção 6.6 deste capítulo.

6.3 Otimização com árvore de segmentos

Embora agora conseguimos obter todos os vértices de qualquer caminho, ainda não conseguimos calcular de forma eficiente qual é o vértice de menor profundidade do conjunto V. Isto é, no pior caso temos n vértices em um caminho e teríamos que percorrer todos estes nós verificando um a um qual é o de menor profundidade, levando a um algoritmo de tempo $O(n)$.

Porém, usando uma famosa estrutura de dados chamada árvore de segmentos (do inglês, *segment tree*), discutida em (4), conseguimos obter o vértice de menor profundidade em tempo $O(log\ n)$. Essa estrutura permite fazer consultas de operações cumulativas (adição, máximo, mínimo, etc) em um intervalo de uma lista de maneira eficiente.

Como o objetivo deste trabalho é focar exclusivamente em resolver o problema do ancestral comum mais próximo, não discutiremos em detalhes como essa estrutura funciona, tampouco escreveremos seu código. Ao invés, assumiremos que temos as seguintes funções de nossa árvore de segmentos, que armazenará as profundidas dos nós:

- **Constrói(lista)**: monta uma árvore de segmentos a partir de uma lista de valores. Leva tempo $O(n)$ onde n é o tamanho da lista.
- **Atualiza(posição, valor)**: atualiza uma posição da árvore de segmentos com o valor passado. Leva tempo $O(log\ n)$.
- **Consulta(esquerda, direita)**: devolve o índice da posição de menor valor no intervalo $[esquerda,\ direita]$. Leva tempo $O(log\ n)$.

Por enquanto, vamos focar na função de consulta. Ela resolve o nosso problema inicial de achar um elemento mínimo em um intervalo de maneira rápida.

Na próxima seção utilizaremos todas as funções mencionadas, juntamente com a ideia descrita na seção **Passeio de Euler** para montar a nossa estrutura completa de determinar LCAs.

6.4 Código

Faremos uma modificação do algoritmo 8 apresentado neste capítulo. Conforme percorremos a árvore contando os tempos vamos simultaneamente construindo a árvore de segmentos. Assumiremos novamente que a variável global *contador* foi inicializada com o valor 1.

Algoritmo 8 Passeio de Euler

```
1: função EULER(vertice)
2:     para cada filho em filhos[vertice] faça
3:         mapa[contador] ← vertice
4:         arvoreSeg[contador] ← profundidade[vertice]
5:         contador ← contador + 1
6:         Euler(filho)
7:     tempo[vertice] ← contador
8:     mapa[contador] ← vertice
9:     arvoreSeg[contador] ← profundidade[vertice]
10:    contador ← contador + 1
```

Abaixo estão os resultados de *mapa*, *arvoreSeg* e *tempo* após a execução deste algoritmo para a árvore apresentada na descrição deste capítulo:

Índice	1	2	3	4	5	6	7	8	9	10	11
mapa	1	2	4	6	4	2	5	2	1	3	1
arvoreSeg	1	2	3	4	3	2	3	2	1	2	1
tempo	11	8	10	5	7	6	-	-	-	-	-

Tabela 6.2: *Resultado da execução do algoritmo 9 para o exemplo*

Algumas observações sobre o que fizemos:

- *mapa*$[x]$ guarda qual é o vértice visitado no instante de tempo x
- *arvoreSeg*$[x]$ guarda qual é a profundidade do vértice obtido ao consultar *mapa*$[x]$
- *tempo*$[v]$ guarda o último tempo registrado do vértice v (para que depois possamos usá-lo na consulta, já que qualquer tempo nos basta) - ou seja - sempre o maior valor de *contador*

O último passo é simplesmente chamar a função **Constrói(arvoreSeg)** para montar a árvore de segmentos a partir dos valores preenchidos em *arvoreSeg*. Assim, para cada consulta temos o algoritmo abaixo:

Algoritmo 9 LCA utilizando a estrutura de Euler

```
1: função LCA(a, b)
2:     esquerda ← tempo[a]
3:     direita ← tempo[b]
4:     se esquerda > direita então
5:         troca(esquerda, direita)
6:     indice ← Consulta(esquerda, direita)
7:     devolve mapa[indice]
```

6.5 Corretude

Vamos mostrar que, após percorrermos a árvore de Euler anotando os tempos de cada vértice da meneira mencionada, podemos escolher **qualquer** tempo de um vértice para ser usado na hora de achar o LCA. Em outras palavras, vamos provar que para qualquer par de tempos escolhido de a e b, o vértice de menor profundidade do intervalo contínuo $[tempo[a],\ tempo[b]]$ da árvore de Euler será o LCA(a, b).

Vamos assumir que o LCA de dois nós a e b é o vértice c. Então, quando chamamos a DFS do Passeio de Euler partindo da raiz sabemos que vamos encontrar o vértice c antes de a e b (afinal, ele é o LCA e tem profundidade menor do que seus descendentes). Estando em c, vamos assumir sem perda de generalidade que a subárvore do vértice a é chamada antes do que a de b. Assim, eventualmente a ganhará um ou mais tempos. O importante é que quando a subárvore inteira de a for visitada, a DFS voltará para o vértice c, dando a ele um novo tempo.

Logo, podemos notar que para todo tempo do vértice a existe um tempo de c que é maior.

Seguindo o algoritmo, quando a DFS for chamada para a subárvore de b, podemos notar que todos os tempos que serão eventualmente associados ao vértice b serão maiores do que algum tempo de c.

Por fim, também é verdade que todo tempo de b será estritamente maior do que todo tempo de a, já que a DFS só será executada para a subárvore de b quando terminar integralmente de percorrer a subárvore de a. Então, se colocarmos os tempos de a, b e c numa linha temporal, temos:

[todos os tempos do vértice a, algum tempo do vértice c, todos os tempos do vértice b]

E por este motivo podemos escolher qualquer tempo de a e qualquer tempo de b, pois sempre existirá um tempo do vértice c contido no intervalo formado pela escolha.

Outra observação deste algoritmo é que no intervalo $[tempo[a],\ tempo[b]]$ podem existir valores associados a vértices que não necessariamente estão no caminho de a até b. Porém, isso não é um problema devido ao fato de que todos eles serão descendentes do vértice c (o LCA do problema) e por isso terão profundidade estritamente maior. Como o intuito desta estrutura é nos possibilitar inferir quem é o LCA verificando o vértice de menor profundidade, estes nós adicionais no intervalo não fazem diferença alguma.

6.6 Complexidade

A complexidade deste algoritmo é fácil de ser obtida. Vamos analisar as três partes do código:

- Algoritmo 8: uma DFS com operações de tempo constante. Portanto, sua complexidade de tempo é $O(n + m)$
- Construção da árvore de segmentos: como discutido anteriormente, esta "função caixa-preta" possui complexidade de tempo $O(n)$, onde n é o tamanho da árvore a ser montada
- Algoritmo 9: esta outra "função caixa-preta" possui complexidade de tempo $O(log\ n)$, onde n é a quantidade de nós da árvore de segmentos. Podemos assumir que o retorno da função é executado em tempo constante já que poderíamos utilizar uma lista para armazenar o mapeamento de tempos para vértices.

Resumindo, a complexidade de tempo da construção de nossa estrutura de dados é $O(n)$, enquanto uma consulta de LCA é $O(log\ n)$.

Capítulo 7

Resolução de problemas

O intuito deste capítulo é utilizar os conceitos sobre LCA vistos nos capítulos anteriores para resolver alguns problemas de Maratona de Programação. Resolveremos questões do juíz online SPOJ, um dos mais famosos na Internet, com milhares de problemas disponíveis para praticar. Além de explicar a ideia de como resolver o problema, também apresentaremos pseudocódigos das partes mais cruciais dos problemas. Os códigos completos (em C++) se encontram no GitHub, com a URL específica no fim de cada explicação.

7.1 ANTS10 - Colônia de Formigas (SPOJ Brasil)

URL: https://br.spoj.com/problems/ANTS10/

Enunciado

Um grupo de formigas está muito orgulhoso pois construíram uma grande e magnífica colônia. No entanto, seu enorme tamanho tem se tornado um problema, pois muitas formigas não sabem o caminho entre algumas partes da colônia. Elas precisam de sua ajuda desesperadamente!

A colônia de formigas foi criada como uma série de N formigueiros conectados por túneis. As formigas, obssessivas como são, numeraram os formigueiros sequencialmente à medida que os construíam. O primeiro formigueiro, numerado 0, não necessitava nenhum túnel, mas para cada um dos formigueiros subsequentes, 1 até $N-1$, as formigas também construíram um único túnel que conectava o novo formigueiro a um dos formigueiros existentes. Certamente, esse conjunto de túneis era suficiente para permitir que qualquer formiga visitasse qualquer formigueiro já construído, possivelmente passando através de outros formigueiros pelo percurso, portanto elas não se preocupavam em fazer novos túneis e continuavam construindo mais formigueiros.

O seu trabalho é, dada a estrutura de uma colônia e um conjunto de consultas, calcular, para cada uma das consultas, o menor caminho entre pares de formigueiros. O comprimento do caminho é a soma dos comprimentos de todos os túneis que necessitam ser visitados.

Entrada

Cada caso de teste se estende por várias linhas. A primeira linha contém um inteiro N representando a quantidade de formigueiros na colônia ($2 \leq N \leq 105$). Cada uma das próximas $N-1$ linhas contém dois inteiros que descrevem um túnel. A linha i, para $1 \leq i \leq N-1$, contém Ai e Li, indicando que o formigueiro i foi conectado diretamente ao formigueiro Ai por um túnel de comprimento Li ($0 \leq Ai \leq i-1$ e $1 \leq Li \leq 109$). A próxima linha contém um inteiro Q representando o número de consultas que seguem ($1 \leq Q \leq 105$). Cada uma das Q linhas seguintes descreve uma consulta e contém dois inteiros distintos S e T ($0 \leq S, T \leq N-1$), representando, respectivamente, os formigueiros de origem e destino.

O último caso de teste é seguido por uma linha contendo apenas um zero.

Saída

Para cada caso de teste, imprima uma única linha com Q inteiros, os comprimentos do menor caminho entre os dois formigueiros de cada consulta. Escreva os resultados para cada consulta na mesma ordem em que aparecem na entrada.

Exemplo de entrada

```
6
0 8
1 7
1 9
0 3
4 2
4
2 3
5 2
1 4
0 3
2
0 1
2
1 0
0 1
6
0 1000000000
1 1000000000
2 1000000000
3 1000000000
4 1000000000
1
5 0
0
```

Exemplo de saída

```
16 20 11 17
1 1
5000000000
```

Neste problema, é dada uma árvore com peso nas arestas. Nosso problema consiste em determinar o comprimento do menor caminho entre dois formigueiros a e b - em outras palavras, a soma dos pesos das arestas escolhidas para chegar de b partindo de a. Como vimos anteriormente, dado que nosso grafo é uma árvore, só existe um caminho entre quaisquer dois vértices. Portanto, não temos que considerar vários caminhos e ter de fazer algum algoritmo que escolha o melhor deles - basta percorrer o único caminho e determinar seu comprimento.

Como estudamos, o LCA entre a e b **sempre** estará contido nos vértices do caminho entre a e b. Assim, podemos quebrar o problema de determinar o comprimento total de a até b nos seguintes subproblemas:

- Calcular o comprimento do caminho de a até o LCA(a, b)
- Calcular o comprimento do caminho de b até o LCA(a, b)

Vamos então agora analisar a árvore do exemplo da entrada do problema e a primeira consulta feita (comprimento do caminho entre os vértices 2 e 3):

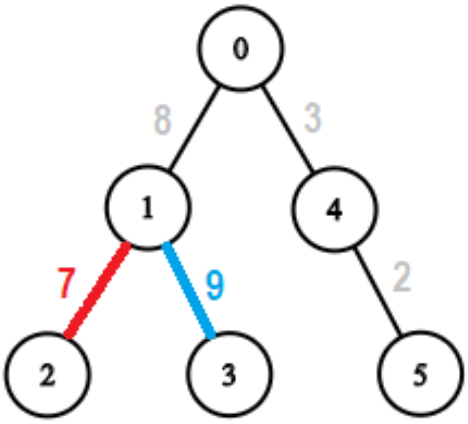

Figura 7.1: *Exemplo de entrada do problema com o comprimento do caminho de* 2 *a* 3

O LCA entre os vértices 2 e 3 é o nó 1. Assim, o comprimento total do caminho é o tamanho do caminho $(1, 2)$ somado ao tamanho do caminho $(1, 3)$. No caso, $9 + 7 = 16$, que é de fato a resposta para a primeira consulta.

Entretanto, como podemos determinar o tamanho do caminho entre o LCA e todos outros vértices de maneira eficiente? Afinal, poderíamos sempre descer na árvore a partir do LCA somando os pesos das arestas - ação que custaria $O(n)$ pois no pior caso teríamos que visitar todos os vértices.

A solução para isso é semelhante ao problema de encontrar a soma dos valores em um intervalo consecutivo de números em uma lista. Suponha que temos a seguinte lista:

$$[4, 10, 5, 6, 8, 2, 1, 3]$$

E queremos a soma do intervalo de índices $[3, 6]$ (indexando a lista do 1). Ao invés de iterar a lista inteira somando os números do intervalo (algoritmo $O(n)$), podemos gerar uma lista das somas acumuladas dos números seguindo a seguinte recorrência:

$$soma[1, i] = soma[1, i-1] + valor[i]$$

Assim, o resultado deste algoritmo para a lista apresentada é a nova lista:

$$[4, 14, 19, 25, 33, 35, 36, 39]$$

Ou seja, cada posição agora é a soma de seu valor com todos os outros que o antecede. Para determinar $soma[e, d]$, basta fazer $soma[d] - soma[e-1]$. Assim, estamos somando a lista inteira até a posição mais a direita e subtraindo a soma do começo até o último elemento antes do nosso limite esquerdo da lista, pois essa parte não faz parte do nosso intervalo desejado. Então no caso dos índices $[3, 6]$, temos que:

$$soma[3, 6] = soma[1, 6] - soma[1, 2] = 35 - 14 = 21$$

Voltando ao nosso problema, adaptaremos essa técnica para calcular a soma dos pesos das arestas do caminho da raiz até cada vértice da árvore com uma simples DFS:

Algoritmo 10 Calculando a soma dos pesos da raiz até todo vértice

```
1: função CalculaSoma(vertice, somaAtual)
2:     soma[vertice] ← somaAtual
3:     para cada filho em filhos[vertice] faça
4:         CalculaSoma(filho, somaAtual + comprimento[vertice, filho])
```

Basta chamar a função *CalculaSoma* com a raiz e 0 (pois o custo de chegar na raiz partindo da raiz é zero). Teremos os seguintes valores de *soma* (em azul) para cada vértice:

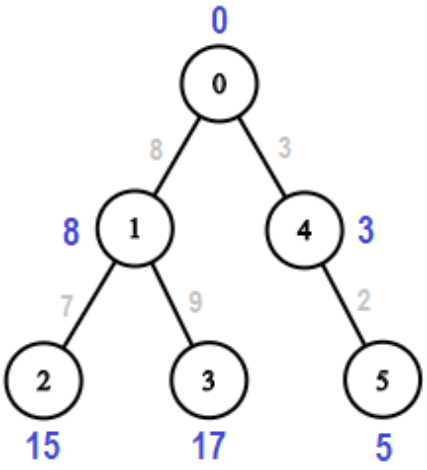

Figura 7.2: *Soma acumulada de cada vértice*

Agora finalmente conseguimos calcular o comprimento do caminho entre quaisquer dois vértices a e b da seguinte maneira:

$$comprimentoCaminho[a, b] = soma[a] - soma[b]$$

Para a consulta do comprimento do caminho $(2, 3)$, temos:

$$caminho(2, 1) = soma[2] - soma[1] = 15 - 8 = 7$$
$$caminho(3, 1) = soma[3] - soma[1] = 17 - 8 = 9$$
$$caminho(2, 3) = caminho(2, 1) + caminho(3, 1) = 7 + 9 = 16$$

Após o pré-processamento (calcular $soma[i]$ para cada vértice), que é feito em tempo linear, conseguimos responder cada consulta em $O(log\ n)$ - a complexidade de obter o LCA entre dois vértices, já que o comprimento do caminho é obtido em $O(1)$ (fazer a subtração das somas).

O código completo (em C++) para este problema encontra-se no GitHub:
https://github.com/PedroBortolli/TCC/blob/master/codes/problemas/ANTS10.cpp

7.2 QTREE2 - Query on a tree II (SPOJ)

URL: https://www.spoj.com/problems/QTREE2/

OBS: o enunciado original do problema está escrito em inglês. O texto abaixo foi traduzido para a língua portuguesa.

Enunciado

É dada uma árvore (um grafo conectado acíclico não direcionado) com N vértices e $N-1$ arestas numeradas $1, 2, 3...N-1$. Cada aresta tem um valor inteiro associado a ela, representando o seu tamanho.

Responda perguntas das seguintes formas:

- **DIST a b**: distância entre os vértices a e b
- **KTH a b k**: $k-$ésimo vértice no caminho de a até b

Entrada

A primeira linha da entrada contém um inteiro t, o número de casos de teste ($t \leq 25$). Então, t casos de testes seguem. Para cada caso de teste:

Na primeira linha é dado um inteiro N ($N \leq 10000$)

Nas próximas $N-1$ linhas, a $i-$ésima linha descreve a $i-$ésima aresta: três inteiros a b c denotam uma aresta entre a e b com custo c ($c \leq 100000$).

As próximas linhas contêm perguntas da forma **"DIST a b"** ou **"KTH a b k"**

O fim de cada caso de teste é indicado por uma string **"DONE"**

Existe uma linha em branco entre sucessivos testes

Saída

Para cada pergunta "DIST"ou "KTH", escreva um inteiro - o resultado da consulta.

Imprima uma linha em branco após cada caso de teste.

Exemplo de entrada

```
1

6
1 2 1
2 4 1
2 5 2
1 3 1
3 6 2
DIST 4 6
KTH 4 6 4
DONE
```

Exemplo de saída

```
5
3
```

Neste problema queremos duas coisas:

- Distância entre dois vértices a e b
- Descobrir o $k-$ésimo vértice no caminho de a até b

A primeira consulta já foi analisada no problema anterior desta seção, então não discutiremos sobre ela novamente aqui. O foco agora é descobrir como responder o segundo tipo de consulta de forma ótima - isto é - melhor do que o jeito trivial $O(n)$ de percorrer o caminho inteiro vértice a vértice.

Como estudamos no capítulo 5, sabemos que conseguimos chegar em qualquer vértice v a partir de um nó fazendo sucessivos pulos de "tamanho potência de dois". Em outras palavras, qualquer número pode ser representado como uma soma de distintas potências de dois.

Portanto, a sugestão para resolver este problema é utilizar o método de programação dinâmica para obter a resposta: pré-calcular os ancestrais 1, 2, 4, 8, etc níveis acima de todo vértice da árvore.

Assim, para obter o $k-$ésimo vértice temos que considerar dois casos:

- Ele pode estar contido no caminho de a até o LCA(a, b). Isso acontece se $profundidade[a] - profundidade[LCA] + 1 \geq k$. Ou seja, se a quantidade de vértices neste caminho é maior ou igual a k, então o $k-$ésimo nó está contido neste caminho
- Senão, ele está contido no caminho de b até o LCA(a, b)

Agora, sabendo em qual caminho devemos procurar, podemos utilizar os ancestrais de potência de dois já pré-calculados para obter nossa resposta. Vamos supor que o $k-$ésimo vértice está no caminho de a até o LCA. Podemos, a partir do nó a, executar o algoritmo de subir sucessivas potências de dois até alcançar o vértice que está $k - 1$ níveis acima de si (pois quando $k = 1$ o vértice a ser retornado é o próprio a). Por exemplo, quando $k = 4$, sabemos que queremos o vértice 3 níveis acima de a. Assim, podemos obter o vértice 2 níveis acima de a, e depois o vértice 1 nível acima deste novo nó encontrado. O resultado será o vértice $2 + 1 = 3$ níveis acima de a.

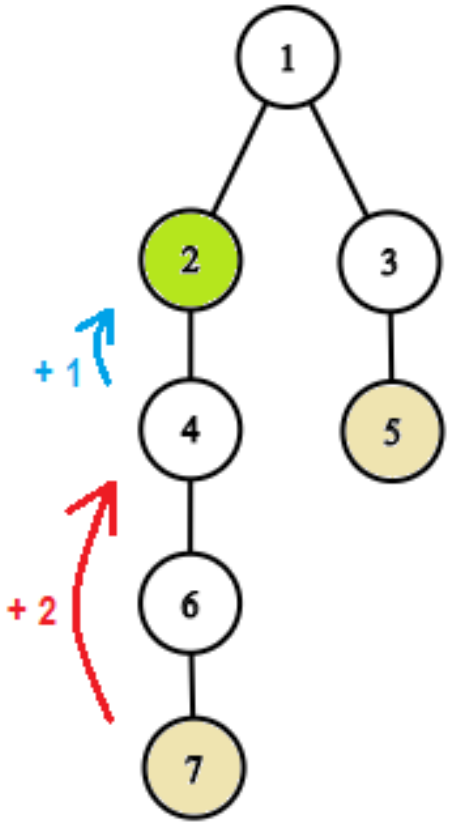

Figura 7.3: *Obtenção do quarto vértice no caminho de 7 até 5*

Seja $quantidadeA$ o número de vértices no caminho de a até o LCA(a, b) e $quantidadeB$ o número de vértices do caminho de b. No caso em que o $k-$ésimo vértice encontra-se no caminho de b até o LCA(a, b), o que queremos agora é o vértice que está $k - quantidadeA$ níveis **abaixo** do LCA. Entretanto, nosso pré-processamento apenas computa os ancestrais de um vértice - em outras palavras - os nós que se encontram acima de um vértice. Assim, precisamos transformar o nosso problema em encontrar um vértice **acima** de b, e podemos fazer isso da seguinte forma:

$$alvo = quantidadeA + quantidadeB - k$$

Agora o valor da variável $alvo$ é a quantidade de níveis que precisamos subir a partir de b para encontrar o $k-$ésimo vértice do caminho de a até b.

Vamos agora escrever o código que encontra o $k-$ésimo vértice, como discutido nesta seção:

Algoritmo 11 Encontrando o $k-$ésimo vértice do caminho de a até b

```
1: função KVERTICE(a, b)
2:     lca ← LCA(a, b)
3:     quantidadeA ← profundidade[a] − profundidade[lca] + 1
4:     quantidadeB ← profundidade[b] − profundidade[lca]
5:     se k ≤ quantidadeA então
6:         alvo ← k − 1
7:         i ← logNiveis
8:         enquanto i ≥ 0
9:             se 2^i ≤ alvo então
10:                alvo ← alvo − 2^i
11:                a = pd[a][i]
           devolve a
12:    senão
13:        alvo ← quantidadeA + quantidadeB − k
14:        i ← logNiveis
15:        enquanto i ≥ 0
16:            se 2^i ≤ alvo então
17:                alvo ← alvo − 2^i
18:                b = pd[b][i]
           devolve b
```

Desta forma, conseguimos encontrar o $k-$ésimo vértice de um caminho em $O(log\ n)$.

O código completo (em C++) para este problema encontra-se no GitHub:
https://github.com/PedroBortolli/TCC/blob/master/codes/problemas/QTREE2.cpp

Capítulo 8

Conclusões

Um dos objetivos deste trabalho foi o aperfeiçoamento em algoritmos que determinam o ancestral comum mais próximo entre dois vértices de uma árvore. Além disso, outro desafio era aprimorar e praticar o rigor matemático para argumentar as corretudes de algoritmos. Ambas tarefas foram cumpridas, uma vez que agora existe uma maior aptidão para abordar problemas de LCA e entender todos os motivos do porquê eles funcionam.

Além disso, outro desejo com este trabalho era de criar um material didático na língua portuguesa para abordar este tema. Considerando este assunto de extrema importância para competições de Maratona de Programação, este trabalho pode ajudar muito a comunidade de Maratona brasileira que está em constante crescimento.

Referências Bibliográficas

[1] TopCoder, "Range minimum query and lowest common ancestor." https://www.topcoder.com/community/competitive-programming/tutorials/range-minimum-query-and-lowest-common-ancestor/. 1

[2] R. L. R. Thomas H. Cormen, Charles E. Leiserson and C. Stein, *Introduction to Algorithms*. MIT Press, 3 ed., Aug. 2009. 3

[3] S. Tommasini, "Programação dinâmica." https://bcc.ime.usp.br/tccs/2014/stefanot/template.pdf,, 2014. Acessado em 12/10/2019. 15

[4] M. de Mello Santos Oliveira, "Árvores de segmentos." https://linux.ime.usp.br/~matheusmso/mac0499/monografia.pdf, 2018. Acessado em 12/10/2019. 24