

Ancestral comum mais baixo

Teoria dos Grafos

Prof. Daniel Saad Nogueira Nunes



**INSTITUTO
FEDERAL**

Brasília

Campus
Taguatinga

Sumário

LCA

Algoritmo Imediato

Binary-lifting

Redução a RMQ

LCA

Definição (Ancestral)

Seja $T = (V, E)$ uma árvore enraizada em r . Um vértice u é ancestral de um vértice v se u está no caminho de r até v . Em particular, v é ancestral de si mesmo.

LCA

Definição (Ancestral comum mais baixo)

Seja $T = (V, E)$ uma árvore enraizada em r . O ancestral comum mais baixo (LCA) de dois vértices u e v , denotado por $\text{LCA}(u, v)$ é o ancestral comum de u e v que está mais distante da raiz r (em uma profundidade maior).

LCA

O problema do LCA viabiliza uma série de aplicações como:

- ▶ Encontrar o menor caminho entre dois vértices em uma árvore;
- ▶ Similaridade entre dois organismos em uma árvore filogenética;
- ▶ Encontrar o menor ancestral comum entre dois elementos em uma hierarquia.
- ▶ Encontrar o menor valor possível em um intervalo de uma sequência de números.
- ▶ ...

LCA

- ▶ O LCA é de extrema importância prática.
- ▶ Necessitamos de abordagens e algoritmos eficientes.

Sumário

LCA

Algoritmo Imediato

Binary-lifting

Redução a RMQ

Algoritmo imediato

- ▶ O algoritmo imediato é bem simples, mas ineficiente.
- ▶ Primeiro computamos a profundidade de cada vértice da árvore, o que pode ser feito em $\Theta(n)$ com uma DFS.
- ▶ Para encontrar o LCA de dois vértices u e v , subimos na árvore a partir do vértice mais profundo até que ambos os vértices estejam na mesma profundidade.
- ▶ Em seguida, subimos na árvore a partir de ambos os vértices até que eles se encontrem. O ponto de encontro é o LCA.

Algoritmo imediato

Algoritmo 1: LCA-NAIVE(u, v)

```
1 while depth[ $u$ ] > depth[ $v$ ] do  $u \leftarrow p[u]$ 
2 while depth[ $v$ ] > depth[ $u$ ] do  $v \leftarrow p[v]$ 
3 while  $u \neq v$  do
4    $u \leftarrow p[u]$ 
5    $v \leftarrow p[v]$ 
6 return  $u$ 
```

Análise

Tempo de execução: pior-caso

- ▶ Gasta-se tempo $\Theta(n)$ para processar a árvore e computar depth e p .
- ▶ Cada consulta gasta tempo $\Theta(h)$, onde h é a altura da árvore.
- ▶ No pior caso, $h = n - 1$, portanto, o algoritmo gasta tempo $\Theta(n)$ por consulta.
- ▶ Tempo total: $\langle \Theta(n), \Theta(n) \rangle$.

Análise

- ▶ Em um cenário de múltiplas consultas, o algoritmo imediato é ineficiente.
- ▶ Precisamos melhorar o tempo de execução das consultas, mesmo que isso implique em gastar mais tempo no pré-processamento da árvore.

Sumário

LCA

Algoritmo Imediato

Binary-lifting

Redução a RMQ

Binary-lifting

- ▶ A técnica de binary-lifting pode ser utilizada para melhorar o tempo de execução das consultas do algoritmo imediato.
- ▶ A ideia é subir na árvore em saltos maiores.
- ▶ Para isso, pré-computamos uma tabela up , onde $up[u][i]$ é o ancestral do vértice u a uma distância de 2^i .
- ▶ Maior salto necessário: $2^{\lceil \lg n \rceil}$.
- ▶ Consideramos que $p[r] = r$. Isto é, o pai da raiz é a própria raiz.

Binary-lifting

$$\text{up}[u][i] = \begin{cases} p[u], & \text{se } i = 0 \\ \text{O ancestral a uma distância de } 2^i \text{ de } u, & \text{se } i > 0 \end{cases}$$

Binary-lifting

- ▶ Como computar $\text{up}[u][i]$?
- ▶ O ancestral a uma distância de 2^i de u é o ancestral a uma distância de 2^{i-1} do ancestral a uma distância de 2^{i-1} de u .
- ▶ Portanto, $\text{up}[u][i] = \text{up}[\text{up}[u][i-1]][i-1]$
- ▶ **Programação dinâmica.**

Binary-lifting

$$\text{up}[u][i] = \begin{cases} p[u], & i = 0 \\ \text{up}[\text{up}[u][i-1]][i-1], & i > 0 \end{cases}$$

Binary-lifting

Tempo de execução: pior-caso

- ▶ O tamanho da tabela up é $n \cdot \lceil \log_2 n \rceil + 1$. Precisamos de $\Theta(n \log n)$ palavras de memória.
- ▶ Cada célula pode ser computada em $\Theta(1)$, portanto, o pré-processamento da árvore gasta tempo $\Theta(n \log n)$.
- ▶ Tempo total do pré-processamento: $\Theta(n \lg n)$.

Sumário

Binary-lifting

Respondendo consultas com binary-lifting

LCA com binary-lifting

- ▶ Dados nós u e v , queremos encontrar $LCA(u, v)$ com binary-lifting.
- ▶ Se u é ancestral de v , então $LCA(u, v) = u$.
- ▶ Se v é ancestral de u , então $LCA(u, v) = v$.
- ▶ Caso contrário, subimos o máximo na árvore a partir de u usando potências de dois decrescentes até um nó w sem que w seja ancestral de v .
- ▶ Em outras palavras: w é o nó mais profundo que é ancestral de u , mas não é ancestral de v .
- ▶ w é filho de $LCA(u, v)$, portanto, $LCA(u, v) = p[w]$.

LCA com binary-lifting

Algoritmo 2: LCA-BINARY-LIFTING(u, v)

```
1 if IS-ANCESTOR( $u, v$ ) then return  $u$ 
2 if IS-ANCESTOR( $v, u$ ) then return  $v$ 
3 for  $i \leftarrow \lceil \lg n \rceil$  to 0 do
4   if not IS-ANCESTOR( $\text{up}[u][i], v$ ) then
5      $u \leftarrow \text{up}[u][i]$ 
6 return  $\text{up}[u][0]$ 
```

LCA com binary-lifting

Tempo de execução: pior-caso

- ▶ Cada consulta gasta tempo $\Theta(\lg n)$, pois o laço percorre $\lceil \lg n \rceil + 1$ iterações.
- ▶ Tempo total: $\langle \Theta(n \lg n), \Theta(\lg n) \rangle$.

LCA com binary-lifting

- ▶ Como saber se um nó u é ancestral de outro nó v ?
- ▶ Podemos fazer uma DFS na árvore e computar os tempos de entrada e saída de cada nó.
- ▶ Um nó u é ancestral de um nó v se e somente se u entra antes de v e sai depois de v .

LCA com binary-lifting

Algoritmo 3: $\text{DFS}(u)$

```
1   $t\_in[u] \leftarrow \text{dfs\_idx}$ 
2   $\text{dfs\_idx} \leftarrow \text{dfs\_idx} + 1$ 
3  for  $(u, v) \in E$  do
4    if  $v \neq p[u]$  then
5       $p[v] \leftarrow u$ 
6       $\text{DFS}(v)$ 
7   $t\_out[u] \leftarrow \text{dfs\_idx}$ 
8   $\text{dfs\_idx} \leftarrow \text{dfs\_idx} + 1$ 
```

LCA com binary-lifting

Algoritmo 4: IS-ANCESTOR(u, v)

1 **return** $t_in[u] \leq t_in[v]$ **and** $t_out[u] \geq t_out[v]$

LCA com binary-lifting

- ▶ Também é possível computar tudo em uma DFS só.
- ▶ Ao chegar em um nó u , temos upw computado para cada ancestral w de u .
- ▶ Portanto, podemos computar $up[u]$ em $\Theta(\lg n)$.
- ▶ O tempo total do pré-processamento continua sendo $\Theta(n \lg n)$.

LCA com binary-lifting

Algoritmo 5: $\text{DFS}(u, w)$

```
1   $t\_in[u] \leftarrow \text{dfs\_idx}$ 
2   $\text{dfs\_idx} \leftarrow \text{dfs\_idx} + 1$ 
3   $\text{up}[u][0] \leftarrow w$ 
4  for  $i \leftarrow 1$  to  $\lceil \lg n \rceil$  do
5     $\text{up}[u][i] \leftarrow \text{up}[\text{up}[u][i - 1]][i - 1]$ 
6  for  $(u, v) \in E$  do
7    if  $v \neq w$  then
8       $\text{DFS}(v, u)$ 
9   $t\_out[u] \leftarrow \text{dfs\_idx}$ 
10  $\text{dfs\_idx} \leftarrow \text{dfs\_idx} + 1$ 
```

A primeira chamada deve ser $\text{DFS}(r, r)$, em que r é a raiz da árvore. O segundo parâmetro é o pai do primeiro parâmetro.

Sumário

LCA

Algoritmo Imediato

Binary-lifting

Redução a RMQ

Redução a RMQ

- ▶ Os problemas de LCA e RMQ estão intimamente relacionados.
- ▶ Podemos reduzir o problema de LCA a RMQ e vice-versa.
- ▶ Isto é, se soubermos resolver um dos problemas, podemos resolver o outro.
- ▶ Como estamos interessados em resolver o problema de LCA, vamos mostrar que $LCA \leq_m RMQ$.

Sumário

Redução a RMQ

Tour Euleriano

Redução

Tabela esparsa

Tour Euleriano

Definição (Tour Euleriano)

Seja $T = (V, E)$ uma árvore não-direcionada e enraizada em r . Um tour Euleriano em T é um passeio que começa na raiz, passa por todos os vértices e percorre cada aresta exatamente uma vez em cada direção e termina na raiz.

Tour Euleriano

- ▶ O tour Euleriano pode ser obtido com uma DFS na árvore.
- ▶ Mantemos registro sempre que entramos e saímos de um vértice.
- ▶ Ao total teremos $n + 1$ entradas e n saídas, portanto, a sequência de vértices de um tour Euleriano terá tamanho $2n + 1$.

Tour Euleriano

- ▶ Para conseguir a redução $LCA \leq_m RMQ$, precisaremos computar três informações.
- ▶ $vertex[i]$ nos dá o identificar do vértice visitado no i -ésimo passo do tour.
- ▶ $first[u]$ indicará em qual índice u ocorre pela primeira vez na sequência de vértices produzidas pelo tour.
- ▶ $depth[i]$ indicará a profundidade do nó visitado no i -ésimo passo do tour.

Tour Euleriano

Algorithm 6: EULERIAN-TOUR(u, p, h)

```
1 vertex[tour_idx]  $\leftarrow u$ 
2 first[ $u$ ] = tour_idx
3 depth[ $u$ ]  $\leftarrow h$ 
4 tour_idx  $\leftarrow$  tour_idx + 1
5 for  $(u, v) \in E$  do
6   if  $v \neq p$  then
7     EULERIAN-TOUR( $v, u, h + 1$ )
8     vertex[tour_idx]  $\leftarrow u$ 
9     tour_idx  $\leftarrow$  tour_idx + 1
```

- A primeira chamada deve ser EULERIAN-TOUR($r, r, 0$), em que r é a raiz da árvore.

Sumário

Redução a RMQ

Tour Euleriano

Redução

Tabela esparsa

Redução $LCA \leq_m RMQ$

- ▶ Com posse de `vertex`, `first` e `depth`, estamos prontos para computar $LCA(u, v)$.
- ▶ Seja $i = \text{first}[u]$ e $j = \text{first}[v]$.
- ▶ Se $i \leq j$, então a resposta $LCA(u, v)$ é:

$$\text{vertex}[\text{RMQ}_{\text{depth}}(i, j)]$$

- ▶ Caso contrário:

$$\text{vertex}[\text{RMQ}_{\text{depth}}(j, i)]$$

- ▶ Ou seja, a consulta de RMQ é realizada sobre o vetor `depth`.

Redução $LCA \leq_m RMQ$

- ▶ Como computar $RMQ_{depth}(i, j)$?
- ▶ Várias possibilidades ...
- ▶ Uma delas é usando uma **árvore de segmentos**, como visto na última aula. Tempo $\langle \Theta(n), \Theta(\lg n) \rangle$.
- ▶ Outra opção, é utilizando a técnica de **tabela esparsa**.

Sumário

Redução a RMQ

Tour Euleriano

Redução

Tabela esparsa

Tabela esparsa

- ▶ Utilizaremos a técnica de tabela esparsa para computar as consultas de RMQ.
- ▶ Essa tabela não terá a computação de todas as respostas, mas combinando entradas dela, é possível chegar na resposta de $\text{RMQ}_V(i, j)$ em tempo $\Theta(1)$.
- ▶ Essencialmente, a tabela $S[i][j]$ conterá a resposta de $\text{RMQ}_V(i, i + 2^j - 1)$. Isto é, $S[i][j]$ possui o índice de onde ocorre o mínimo em um intervalo que começa em i e tem tamanho 2^j .
- ▶ Como computar $S[i][j]$? **Programação-dinâmica**, como no binary-lifting.

Tabela esparsa

$$S[i][j] = \begin{cases} i, & j = 0 \\ S[i][j-1], & j > 0 \text{ e } i + 2^j > n \\ S[i][j-1], & j > 0 \text{ e } V[S[i][j-1]] \leq V[S[i + 2^{j-1}][j-1]] \\ S[i + 2^{j-1}][j-1], & \text{caso contrário} \end{cases}$$

Tabela esparsa

Tempo de execução: pior-caso

- ▶ A tabela tem $n \cdot (\lfloor \lg n \rfloor + 1)$ entradas.
- ▶ Cada computação pode ser feita em tempo $\Theta(1)$.
- ▶ Logo, o tempo de pré-processamento é $\Theta(n \lg n)$.

Tabela esparsa

- ▶ Como computar $\text{RMQ}_V(i, j)$ com a tabela esparsa?
- ▶ Seja $\ell = \lfloor \lg(j - i + 1) \rfloor$.
- ▶ Sabemos que $S[i][\ell]$ possui o mínimo do intervalo $[i, i + 2^\ell - 1]$. Mas isso não é suficiente para cobrir o intervalo $[i, j]$ inteiro.
- ▶ Contudo, ao adicionar essa informação com $S[j - 2^\ell + 1][\ell]$, conseguimos cobrir o intervalo todo, visto que essa entrada cobre o intervalo $[j - 2^\ell + 1, j]$. Inclusive pode existir uma sobreposição dos intervalos cobertos pelas duas entradas da tabela esparsa, mas não há problema nisso, pois o problema que estamos resolvendo é o problema de mínimo.

Tabela esparsa

Algorithm 7: $\text{RMQ}_V(i, j)$

```
1  $\ell \leftarrow \lceil \lg j - i + 1 \rceil$ 
2  $a \leftarrow S[i][\ell]$ 
3  $b \leftarrow S[j - 2^\ell + 1][j]$ 
4 if  $V[a] \leq V[b]$  then
5   return  $a$ 
6 else
7   return  $b$ 
```

Tabela esparsa

Tempo de execução: pior-caso

- ▶ Pré-processamento $\Theta(n \lg n)$.
- ▶ Consulta $\Theta(1)$.
- ▶ Total: $\langle \Theta(n \lg n), \Theta(1) \rangle$.