Introdução a Heavy light Decomposition

Enoque Alves de Castro Neto [1], **David Sena Oliveira**[2], **Wladimir Araujo Tavares**[3], **Alex Sandro Alves de Souza**[4]

[1]Universidade Federal do Ceará (UFC) - Campus Quixadá
Av. José de Freitas Queiroz, 5003 – Cedro – Quixadá – Ceará 63902-580

{enoquealvesufc,sena.ufc,wladimirufc}@gmail.com

{alex_sandro}@alu.ufc.br

Abstract. *In competitive programming, problems involving trees are common. A not very well known, but very efficient technique is Heavy Light Decomposition. This paper aims to address the Heavy Light Decomposition technique to efficiently solve the problem of finding the largest edge between two vertices belonging to the tree.*

Resumo. *Em programação competitiva é comum problemas envolvendo árvores. Uma técnica não muito conhecida, porém muito eficiente é Heavy Light Decomposition. Esse artigo tem como objetivo abordar a técnica Heavy Light Decomposition para resolver de forma eficiente o problema de encontrar a maior aresta entre dois vértices pertencente a árvore.*

1. Introdução

Em competições de programação, como a Maratona de Programação e Olimpíadas Brasileira de Informática(OBI), é comum propor vários problemas relacionados a grafos e árvores, como por exemplo, problema de caminho mínimo, problema da árvore geradora com custo mínimo, problema em busca em grafos e problema do menor ancestral comum. Alguns desses problemas exigem que você resolva os problemas diversas vezes com pequenas modificações na estrutura do grafo. Esse novo cenário requer uma estrutura de dados que consiga realizar a modificação e/ou resolução do problema de maneira eficiente. Todas as imagens usadas nesse artigo foram retiradas de *GeeksForGeeks* e os pseudocódigos foram baseados em *ANUDEEP'S BLOG*. Este artigo assume que o leitor conheça a estrutura de dados árvore de segmentos.

Considere o seguinte problema: Dada uma árvore ponderada desbalanceada (a diferença da altura das subárvores é maior que 1), é necessário fazer as seguintes operações várias vezes:

1. **atualizar(a,b,w)**: Atualizar o peso da aresta (a,b) para o valor *w*;
2. **maxEdge(a,b)**: Retornar a maior aresta entre o caminho *a* até *b*;

A Figura 1 apresenta um exemplo de uma árvore ponderada desbalanceada. Uma solução simples para operação $maxEdge(a, b)$ seria percorrer todas as arestas do caminho entre a e b que pode ser realizado em $\mathcal{O}(n)$. A operação $atualizar(a, b, w)$ pode ser realizada em tempo $\mathcal{O}(d(a)+d(b))$, onde $d(a)$ é o grau de a. Nos dois casos, considerando que o grafo seja representado utilizando uma lista de adjacência.

Neste artigo, será apresentado uma decomposição da árvore que permite realizar a operação $atualizar(a, b, w)$ com a complexidade $\mathcal{O}(logn)$ e a operação $maxEdge(a, b)$ com a complexidade $\mathcal{O}(log^2 n)$

2. *Heavy Light Decompisition*

O **HLD** (*Heavy Light Decompisition*) é dividido em duas partes para alcançar a solução final:

1. Decompor a árvore em subárvores de modo que seja preciso passar por no máximo $log\, n$ subárvores para todo caminho entre dois pares de vértices, sendo n o número de vértices da árvore.
2. Para cada subárvore S, é necessário achar a maior aresta entre qualquer par de vértices pertencente S e atualizar o valor de uma aresta com complexidade na ordem de $\mathcal{O}(logn)$. Uma estrutura de dados que satisfaz essas condições e comumente é usada para implementar o **HLD** é árvore de segmentos.

2.1. Decomposição da árvore

A decomposição da árvore é feita de modo que a árvore seja dividida em várias cadeias de vértices disjunta. É chamado de cadeia de vértice as subárvores obtidas pela a decomposição. Essa decomposição é feita usando o seguinte critério:

1. Todo nó não-folha tem exatamente um filho que é chamado de **filho especial**.
   (a) O filho especial de um nó é o filho que pertence à mesma cadeia de vértice que o pai.
   (b) O filho especial é o filho que possui a maior subárvore dentre todos os filhos.
2. Toda cadeia de vértice possui um vértice que é chamado de cabeça da cadeia e uma aresta especial que conecta a cabeça a outra cadeia.
3. Cada filho não especial de um nó dá origem a uma nova cadeia de vértices, sendo esse filho a cabeça da nova cadeia.
4. O primeiro nó encontrado de cada cadeia de vértice é considerado a cabeça da cadeia.

Na Figura 1, podemos ver como fica a árvore após a decomposição onde cada cor representa uma cadeia diferente.

2.2. Construção da árvore de segmentos

Como é esperado, a árvore de segmentos necessita de um vetor base para sua construção. Nesse vetor base cada vértice possui um índice correspondente a esse vetor e seu valor é o peso da aresta usada para chegar até aquele vértice. A construção do vetor base é iniciado a partir da raiz (No grafo exemplo, é considerado que o vértice 1 é a raiz) e adicionando o restante usando uma busca em profundidade dando prioridade aos filhos especiais. Após construir o vetor base, a árvore de segmentos deve ser construída de forma que dando um intervalo **(a,b)** ela retorne o maior valor entre o intervalo. Cada elemento do vetor base será modelado por um par ordenado (a, w), onde a é o vértice e w é o peso da aresta usada para chegar em a. O vetor base para árvore Figura 1 é (1,-1),(2,13),(6,25),(8,5),(10,1),(11,6),(5,4),(3,9),(7,29),(8,30) e (4,23). A Figura 2 mostra como fica a árvore de segmento após construída.

2.3. Obtendo as respostas

Após decompor a árvore, as arestas que ligam cada cadeia são consideradas especiais. Para fazer a operação **maxEdge(a,b)**, podemos quebrar a operação entre o máximo de **maxEdge(a,LCA(a,b))** e **maxEdge(b,LCA(a,b))**, onde LCA(a,b) é o menor ancestral comum entre o vértice a e o vértice b. Dentro dessa busca é pego a maior aresta entre as arestas pertencente ao caminho que está na cadeia atual de a até a próxima cadeia e a aresta especial que liga as duas cadeias, após isso troca a cadeia atual e repete o processo. As figuras 3 e 4 representam esse processo descrito anteriormente.

Para a operação **atualizar(a,b,w)**: Seja $v = a$ caso a seja mais profundo que b na árvore e $v = b$ caso contrário, basta acessar o índice do vetor base correspondente ao vértice v e atualizar o valor para w. Logo após, atualizar todo restante necessário da árvore.

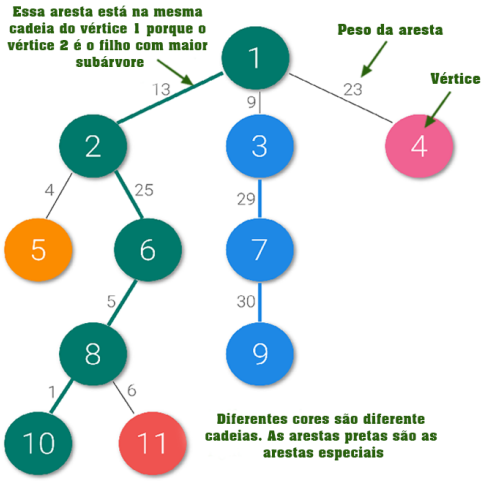

Figura 1. Decomposição de árvore

Figura 1.

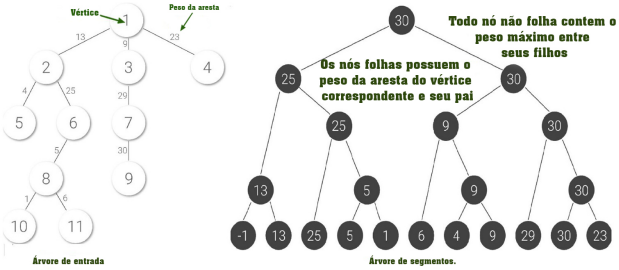

Figura 2. Construção da árvore de segmentos

Figura 2. Construção da árvore de segmentos

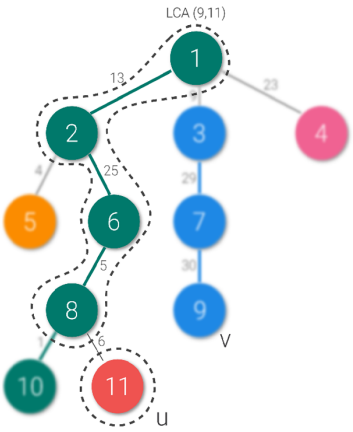

Figura 3. maxEdge(a,LCA(a,b))

Figura 3. maxEdge(a,LCA(a,b))

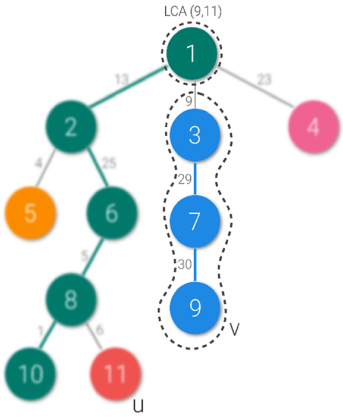

Figura 4. maxEdge(b, LCA(a,b))

Figura 4. maxEdge(b, LCA(a,b))

3. Algoritmo

Nesta seção, usaremos as seguintes funções auxiliares:

1. **add(No,Cadeia)**: Adiciona o no à cadeia.
2. **add_tree(**$Valor, Arvore\_segmentos$**)**: Adiciona o valor à árvore de segmento, retorna o índice referente ao valor adicionado.
3. **LCA(**a, b**)**: Retorna o menor ancestral comum de a e b.
4. **maior(**a, b**)**: Retorna o maior entre a e b
5. **busca_arvore(**a, b**)**: Retorna o maior elemento no intervalo [a,b] na árvore de segmento.
6. **cadeia(**u**)**: Retorna a cadeia a qual o vértice u pertence.
7. **cabeca(**$cadeia$**)**: Retorna o vértice cabeça da $cadeia$.
8. **pai(**u**)**: Retorna o vértice pai de u.

3.1. Construção do HDL

Algoritmo 1: HLD_BUILD

Entrada: $NoAtual, NoAnterior, Cadeia$
1 **início**
2 $w \leftarrow$ aresta $(NoAnterior, NoAtual)$
3 $indiceArvore[NoAtual] \leftarrow add\_tree(w, arvore\_segmento)$
4 **se** *NoAtual é folha* **então**
5 **retorna**
6 **fim**
7 $filhoEspecial \leftarrow$ filho do NoAtual de maior subarvore
8 $HLD\_BUILD(filhoEspecial, NoAtual, Cadeia)$
9 **para** ***cada*** $filhoNormalf \in NoAtual$ **faça**
10 **HLD_BUILD(**$f, NoAtual, novaCadeia$**)**
11 **fim**
12 **fim**

O algoritmo 1 mostra a construção da árvore de segmentos e da decomposição dos vértices em cadeia. Na linha 3, é adicionado a aresta $(NoAnterior, NoAtual)$ na árvore de segmentos e salvo o índice a qual ela pertence na árvore de segmento. Nas linhas seguintes, a função **HLD_BUILD** é chamada recursivamente paras todos os filhos de $NoAtual$ criando uma nova cadeia, caso esse filho seja norma l, ou, mantendo a mesma cadeia, caso o filho seja especial.

3.2. maxEdge

Algoritmo 2: MAXEDGE

Entrada: u, v
Saída: Maior aresta entre o caminho u até v
1 **início**
2 $lca \leftarrow$ **LCA(**a, b**)**
3 **retorna *maior(maxEdge_up(***u, lca***), maxEdge_up(***v, lca***))***
4 **fim**

Algoritmo 3: MAXEDGE_UP

Entrada: u, v
Saída: Maior aresta entre o caminho u até v

```
1  início
2      cadeiaU ← cadeia(u)
3      cadeiaV ← cadeia(v)
4      ans ← −1
5      repita
6          atual ← busca_arvore(indice_Arvore(cabeca(cadeiaU)),
             indice_Arvore(u))
7          ans ← maior(ans, atual)
8          u ← cabeca(cadeiaU)
9          u ← pai(u)
10     até cadeiaU == cadeiaV;
11     atual ← busca_arvore(indice_Arvore(v), indice_Arvore(u)
12     ans ← maior(ans, atual)
13     retorna ans
14 fim
```

O algoritmo 2 quebra o caminho (a,b) em duas partes e retorna o maior valor obtido entre as partes na função **maxEdge_up**. O algoritmo 3 descreve a função **maxEdge_up**. A função **maxEdge_up** calcula a maior aresta pertencente ao intervalo (**cabeca**($cadeiaU$), u) até que u esteja na mesma cadeia que v. Quando $cadeiaU == cadeiaV$ basta calcular a maior aresta no intervalo (v,u) e retornar a maior aresta dentre todas calculadas.

3.3. Complexidade das operações

Seja n o tamanho da subárvore de um vértice qualquer. Como cada filho especial possui a maior subárvore de todos os filhos, então o tamanho máximo da subárvore de um filho comum é no máximo $n/2$. Realizando essa operação para todos os filhos, o número cadeias é no máximo $\lceil log\ n \rceil$.

As operações de busca e atualização da árvore de segmentos são ambas na ordem de $O(log\ n)$. Como a operação $atualizar(a, b, w)$ é apenas uma atualização na árvore de segmentos, a complexidade final de $atualizar$ é $O(log\ n)$.

Já a operação de $maxEdge(a, b)$ faz uma busca na árvore de segmentos para cada cadeia de vértice no caminho entre a e b. Como existem no máximo $log\ n$ cadeias e a operação de busca na árvore de segmentos é $log\ n$, a complexidade final de $maxEdge(a, b)$ é $O(log^2\ n)$.

4. Referências

GEEKSFORGEEKS. **Heavy Light Decomposition — Set 1 (Introduction)**. Disponível em: <https://www.geeksforgeeks.org/heavy-light-decomposition-set-1-introduction/> Acesso em: 02 de outubro de 2018.

ANUDEEP'S BLOG. **Heavy Light Decomposition**. Disponível em: <https://blog.anudeep2011.com/heavy-light-decomposition/> Acesso em: 02 de outubro de 2018.