

Heavy-Light Decomposition (HLD)

Heavy-Light Decomposition (HLD)

Tác giả:

- Phạm Hoàng Hiệp – University of Georgia

Reviewer:

- Nguyễn Minh Hiển - Trường Đại học Công Nghệ, ĐHQGHN
- Nguyễn Minh Nhật - Trường THPT chuyên Khoa học Tự nhiên, ĐHQGHN
- Đặng Đoàn Đức Trung - UT Austin

Mở đầu

Giới thiệu về HLD

Heavy-Light Decomposition (HLD), dịch ra tiếng Việt là phân chia nặng nhẹ là một kỹ thuật thường được dùng trong những bài toán xử lý trên cây.

Trong bài viết này, để ngắn gọn và dễ nhớ, chúng ta sẽ gọi tên kỹ thuật là **HLD**.

Tuy nghe tên có vẻ kinh khủng nhưng trên thực tế, đây là một kỹ thuật có ý tưởng khá tự nhiên và có tính ứng dụng cao, có thể được sử dụng trong nhiều bài tập.

Kiến thức cần biết

- Các thuật toán duyệt đồ thị cơ bản (DFS, BFS, ...)
- Cây
- [Lowest Common Ancestor \(LCA\)](#) - Tổ tiên chung gần nhất
- [Segment Tree](#)
- [Euler tour on tree](#) (nên biết nhưng không bắt buộc)

Bài toán

Để trả lời câu hỏi HLD sẽ giúp chúng ta làm gì, chúng ta sẽ cùng giải một bài toán.

Trước hết, chúng ta sẽ đến với phiên bản dễ hơn của bài toán như sau.


Cho một mảng số nguyên dương gồm tối đa 10^5 phần tử. Chúng ta cần xử lý tối đa 10^5 truy vấn thuộc một trong hai loại sau:

- Cập nhật giá trị của phần tử thứ i thành x

- Tính tổng XOR của tất cả các phần tử trong đoạn từ l đến r

Bài toán trên là một bài toán quen thuộc và có thể được xử lý đơn giản bằng cách sử dụng Segment Tree. Nhưng, giả sử thay vì mảng một chiều, chúng ta cần xử lý bài toán trên cây thì phải làm như thế nào?

Phát biểu bài toán

Bạn đọc có thể đọc đề bài cụ thể và nộp code tại [đây](#) 

Cho một cây (một đồ thị có n đỉnh và $n - 1$ cạnh và giữa hai đỉnh bất kỳ có đúng một đường đi giữa chúng). Mỗi đỉnh được gán một giá trị. Chúng ta cần xử lý hai loại truy vấn

- Cập nhật giá trị của đỉnh i thành x
- Tính tổng XOR của tất cả các giá trị trên đường đi từ đỉnh u đến đỉnh v

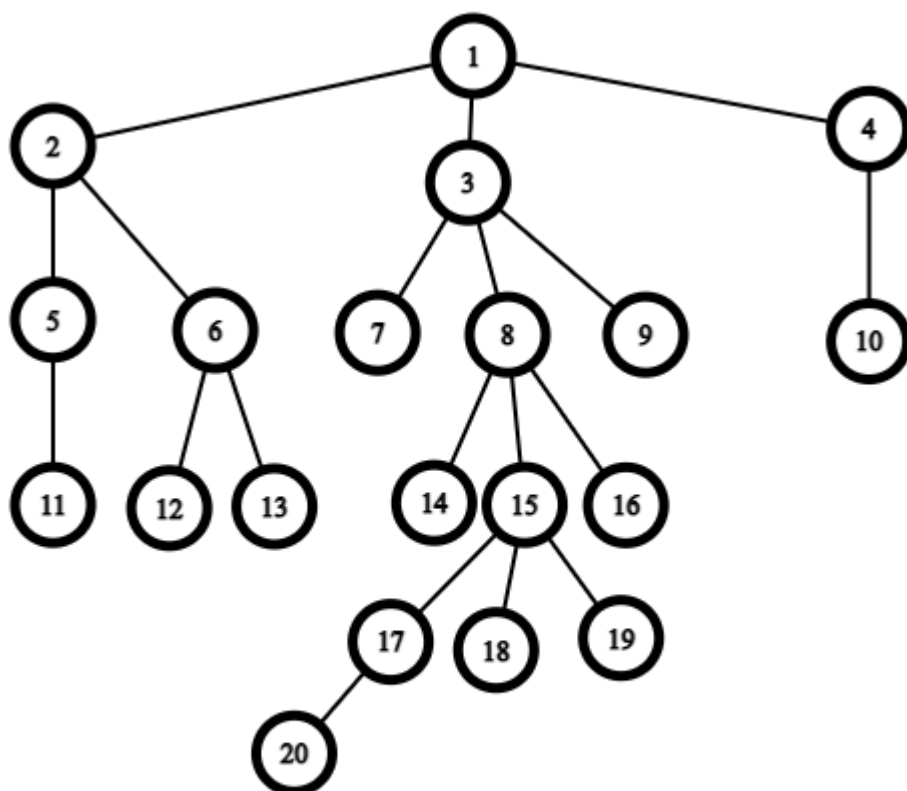
Đường đi từ u đến v trên đồ thị được định nghĩa là một chuỗi các đỉnh $u, w_1, w_2, \dots, w_k, v$ trong đó tồn tại một cạnh nối giữa u và w_1 , w_1 và w_2 , ..., w_{k-1} đến w_k , w_k đến v . Với mỗi cặp đỉnh u, v bất kỳ trên cây tồn tại một và chỉ một đường đi từ u đến v .

HLD

Ý tưởng chính

Vậy chính xác thì HLD sẽ làm gì để giúp chúng ta giải được phiên bản "trên cây" của bài toán trên? Liệu chúng ta có thể biến cây cho trước thành một mảng để giải bài toán trên đó? Câu trả lời là không. Tuy nhiên chúng ta có thể chia cây thành một số phần, và giải bài toán trên từng phần đó.

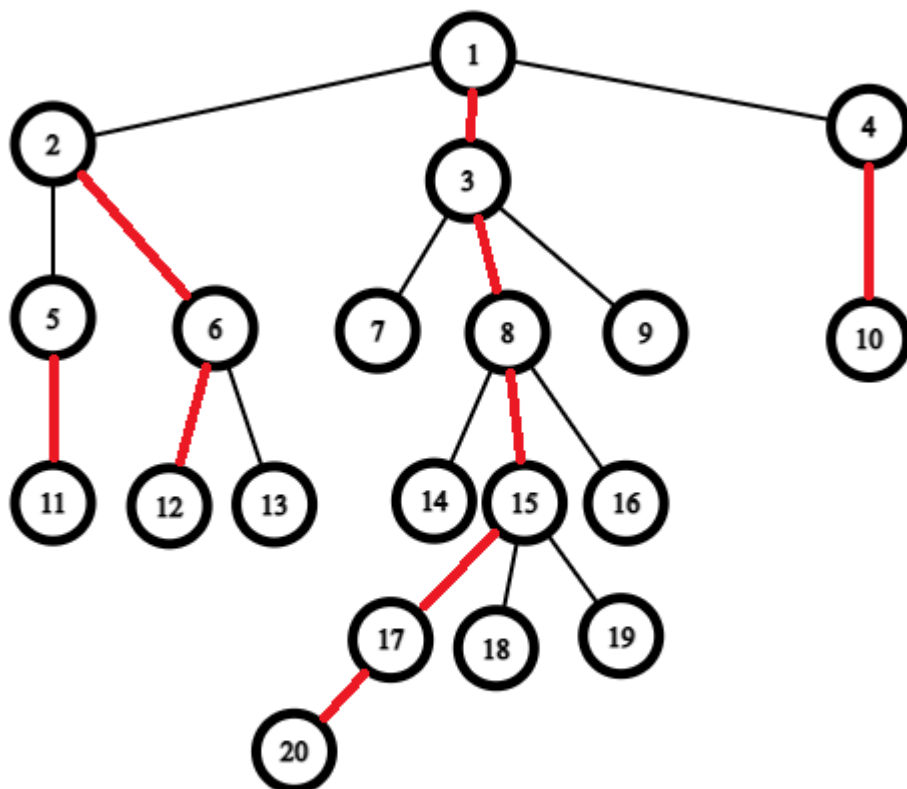
Cụ thể như sau, giả sử có cây sau đây



Không mất tính tổng quát, có thể coi đỉnh 1 là gốc của cây. Với mỗi đỉnh không phải lá trên cây, chúng ta sẽ đánh dấu những cạnh nối đỉnh đó với con có kích thước cây con lớn nhất của nó.

Ví dụ, xét đỉnh 15 có ba đỉnh con lần lượt là đỉnh 17, 18 và 19. Cây con ở hai đỉnh 18 và 19 có kích thước là 1, còn cây con ở đỉnh 17 có kích thước là 2.

Vì vậy, chúng ta sẽ đánh dấu cạnh nối giữa đỉnh 15 và đỉnh 17 (tô màu đỏ). Làm tương tự với các đỉnh còn lại, chúng ta được cây như hình vẽ dưới đây.



Chúng ta sẽ gọi những cạnh màu đỏ là những **"cạnh nặng" (heavy edges)** vì chúng nối một đỉnh với đỉnh con **"nặng nhất"**. Những cạnh còn lại sẽ được gọi là những **"cạnh nhẹ" (light edges)**

Có thể thấy rằng, những cạnh được đánh dấu sẽ tạo thành các **"chuỗi"** đi từ trên xuống dưới. Ví dụ như chuỗi $1 - 3 - 8 - 15 - 17 - 20$ hay chuỗi $5 - 11$. Các đỉnh là lá cũng có thể coi là một chuỗi riêng.

Với cách quy ước trên, ta có thể thấy rằng hai chuỗi "liên nhau" được nối với nhau bởi một cạnh nhẹ. Hai chuỗi "liên nhau" là hai chuỗi mà tồn tại đỉnh u nằm ở một chuỗi và v nằm ở chuỗi còn lại sao cho u và v có cạnh nối trực tiếp với nhau. Do u và v khác chuỗi nên cạnh nối giữa u và v là cạnh nhẹ. Hai chuỗi không thể nối ở nhiều hơn một cặp điểm.

Khi đó, chúng ta có thể chứng minh được rằng đường đi giữa hai đỉnh **bất kỳ** trên cây đi qua không quá $O(\log(n))$ chuỗi. Bạn đọc có thể tự chứng minh nhận xét trên trước khi đọc chứng minh bên dưới.

► Chứng minh

Ngoài ra, nếu ta duyệt cây bằng DFS *ưu tiên các đỉnh liền trong chuỗi trước*, ta có thể nhận thấy là các đỉnh trên cùng một chuỗi sẽ nằm kế tiếp nhau trên thứ tự DFS, và vì thế việc truy vấn một đoạn con bất kì trên một chuỗi nào đó có thể được thực hiện trên Segment tree với độ phức tạp là $O(\log(n))$.

Như vậy, chúng ta đi qua không quá $O(\log(n))$ chuỗi, với mỗi chuỗi chúng ta có thể thực hiện trả lời truy vấn hoặc update trong $O(\log(n))$ trên Segment tree nên độ phức tạp cho việc trả lời các truy vấn và cập nhật trên một đường đi giữa hai đỉnh bất kỳ trên cây là $O(\log^2(n))$

Chi tiết cài đặt

Thông thường, do việc sử dụng HLD sẽ đi kèm với một cấu trúc dữ liệu nào đó và duyệt đồ thị, code có thể sẽ dài và gồm nhiều phần. Tuy nhiên, nếu nắm chắc ý tưởng chính thì cài đặt HLD rất đơn giản.

Tiền xử lý

Đầu tiên, ta cần phải tính kích thước của cây con từng đỉnh để lấy ra các con "nặng" của từng đỉnh một. Ngoài ra, ta cần tính thêm độ sâu các đỉnh để phục vụ cho thao tác tính LCA.

```

1  void Dfs(int s, int p = -1) {
2      Sz[s] = 1;
3      for(int u: AdjList[s]) {
4          if(u == p) continue;
5          Par[u] = s;
6          Depth[u] = Depth[s] + 1;
7          Dfs(u, s);
8          Sz[s] += Sz[u];
9      }
10 }
```

mảng Sz , $Depth$ và Par lần lượt lưu kích thước cây con, độ sâu và cha (tổ tiên trực tiếp) của các đỉnh trên cây

mảng $AdjList$ là mảng vector để lưu lại thông tin về đồ thị. $AdjList[s]$ là vector gồm các đỉnh kề với s .

Tìm cạnh nặng, cạnh nhẹ và tạo các chuỗi

Sau đó chúng ta thực hiện phân chia các đỉnh vào các chuỗi. Với mỗi đỉnh, cần lưu lại chuỗi của đỉnh và vị trí của đỉnh khi đặt các chuỗi liên tiếp với nhau (để thuận tiện cho việc xử lý truy vấn). Với mỗi chuỗi, chúng ta cần biết đỉnh đầu tiên của chuỗi (để thực hiện việc nhảy giữa các chuỗi).

```

1  void Hld(int s, int p = -1) {
2      if(!ChainHead[CurChain]) {
3          ChainHead[CurChain] = s;
4      }
5      ChainID[s] = CurChain;
6      Pos[s] = CurPos;
7      Arr[CurPos] = s;
8      CurPos++;
9      int nxt = 0;
10     for(int u: AdjList[s]) {
11         if(u == p) continue;
12         Hld(u, s);
13         if(ChainID[u] != ChainID[s]) {
14             ChainHead[CurChain] = u;
15             CurChain++;
16             CurPos = 0;
17         }
18     }
```

```

12         if(u != p) {
13             if(nxt == 0 || Sz[u] > Sz[nxt]) nxt = u;
14         }
15     }
16     if(nxt) Hld(nxt, s);
17     for(int u: AdjList[s]) {
18         if(u != p && u != nxt) {
19             CurChain++;
20             Hld(u, s);
21         }
22     }
}

```

nxt là biến dùng để lưu lại đỉnh con "nặng nhất".

Arr là mảng dùng để lưu lại các chuỗi.

ChainID là mảng lưu lại số thứ tự của các chuỗi.

ChainHead là mảng lưu lại node đầu tiên (*Depth* bé nhất) của từng chuỗi để biết khi nào cần nhảy sang chuỗi mới qua cạnh nhẹ.

Mảng *Pos* lưu lại vị trí của các đỉnh trên *Arr* để tiện xử lý trên segment tree.

CurChain và *CurPos* lần lượt là các biến lưu lại chỉ số của chuỗi và vị trí trong mảng *Arr* để dùng cho chuỗi và đỉnh tiếp theo

Như vậy, với mỗi đỉnh, chúng ta sẽ tìm cạnh nặng và đi xuống cạnh đó trước. Sau đó, chúng ta sẽ lần lượt tạo ra các chuỗi mới và nhảy sang các đỉnh nhẹ. Như vậy, thứ tự duyệt đồ thị sẽ đảm bảo mỗi chuỗi được lưu đúng thứ tự từ trên xuống dưới trong một đoạn liên tiếp trên mảng *Arr*.

Tìm LCA

Trong phần lớn các bài toán sử dụng HLD để thực hiện truy vấn trên đường đi, chúng ta cần tìm tổ tiên chung gần nhất và thực hiện thao tác lần lượt từ hai đỉnh đến tổ tiên chung này. Rất may là chúng ta có thể sử dụng chính những thông tin đã lưu để tìm ra LCA một cách nhanh chóng.

Khi nhảy từ một node *v* qua một cạnh nhẹ lên cha của nó, chúng ta luôn nhảy sang một chuỗi có số thứ tự bé hơn (do thứ tự duyệt đồ thị trong hàm *HLD*) nên để tìm LCA của hai đỉnh, chúng ta sẽ tìm chuỗi chứa LCA đó.

Liên tục thực hiện nhảy từ chuỗi có số thứ tự lớn hơn lên một chuỗi có số thứ tự bé hơn (Để đảm bảo không bị nhảy quá, luôn chọn đỉnh đang ở chuỗi có số thứ tự lớn hơn để nhảy) cho đến khi hai đỉnh nằm trên cùng một chuỗi.

Ở đó, đỉnh có độ sâu thấp hơn là LCA của *u* và *v*.

```

1  int LCA(int u, int v) {
2      while(ChainID[u] != ChainID[v]) {
3          if(ChainID[u] > ChainID[v]) {
4              u = Par[ChainHead[ChainID[u]]];
5          }
6          else {
7              v = Par[ChainHead[ChainID[v]]];
8          }
9      }
10     if(Depth[u] < Depth[v]) return u;

```

```

11 |     return v;
12 | }

```

Segment tree

Dưới đây là các thao tác xử lý trên segment tree. Phần này sẽ đủ để xử lý phiên bản không trên cây của bài toán.

```

1 | int ST[MaxN * 4];
2 | void Build(int id, int l, int r) {
3 |     if(l == r) {
4 |         ST[id] = Val[Arr[l]];
5 |         return;
6 |     }
7 |     int mid = (l + r) / 2;
8 |     Build(id * 2, l, mid);
9 |     Build(id * 2 + 1, mid + 1, r);
10 |    ST[id] = ST[id * 2] ^ ST[id * 2 + 1];
11 | }
12 |
13 | void Upd(int id, int l, int r, int pos, int val) {
14 |     if (l > pos || r < pos) return;
15 |     if (l == r && l == pos) {
16 |         ST[id] = val;
17 |         return;
18 |     }
19 |     int mid = (l + r) / 2;
20 |     Upd(id * 2, l, mid, pos, val);
21 |     Upd(id * 2 + 1, mid + 1, r, pos, val);
22 |     ST[id] = ST[id * 2] ^ ST[id * 2 + 1];
23 | }
24 |
25 | int Calc(int id, int tl, int tr, int l, int r) {
26 |     if (tl > r || tr < l) return 0;
27 |     if (l <= tl && tr <= r) return ST[id];
28 |     int mid = (tl + tr) / 2;
29 |     return Calc(id * 2, tl, mid, l, r) ^ Calc(id * 2 + 1, mid + 1, tr, l, r);
30 | }

```

Các thao tác trên cây

Và cuối cùng, chúng ta sẽ có các hàm để xử lý truy vấn trên cây. Tất nhiên có thể đưa toàn bộ phần này vào trong main mà không tăng độ dài code. Tuy nhiên để dễ nhìn và tiện debug, chúng ta sẽ code riêng hàm để xử lý truy vấn trên đường đi từ đỉnh u đến đỉnh v .

```

1 | void Update(int x, int val) {
2 |     Upd(1, 1, N, Pos[x], val);
3 | }
4 |

```

```

5   int Query(int u, int v) {
6       int lca = LCA(u, v);
7       int ans = 0;
8       while(ChainID[u] != ChainID[lca]) {
9           ans ^= Calc(1, 1, N, Pos[ChainHead[ChainID[u]]], Pos[u]);
10          u = Par[ChainHead[ChainID[u]]];
11      }
12      while(ChainID[v] != ChainID[lca]) {
13          ans ^= Calc(1, 1, N, Pos[ChainHead[ChainID[v]]], Pos[v]);
14          v = Par[ChainHead[ChainID[v]]];
15      }
16      if(Depth[u] < Depth[v]) {
17          ans ^= Calc(1, 1, N, Pos[u], Pos[v]);
18      }
19      else {
20          ans ^= Calc(1, 1, N, Pos[v], Pos[u]);
21      }
22      return ans;
23  }

```

Do bài toán chỉ yêu cầu cập nhật trên điểm nên hàm *Update* không có gì đáng chú ý. Độ phức tạp của hàm này là $O(\log(n))$.

Hàm *Query* dùng để trả lời truy vấn tổng XOR của các số trên đường đi từ u đến v . Sau đó, chúng ta thực hiện chia đường đi này thành các đoạn trên các chain rồi thực hiện thao tác tính trên từng đoạn.

Có thể thấy, cách nhảy trong khi tính toán Query chính là cách nhảy khi tìm LCA. Trên thực tế, chúng ta không cần tìm LCA trước mà có thể thực hiện việc đó ngay khi tính Query. Nhưng trong bài này, phần tìm LCA được code riêng một hàm để dễ hiểu và tiện giải thích.

Code đầy đủ

Dưới đây là code đầy đủ cho bài toán (kết hợp của tất cả các đoạn trên, khai báo biến và hàm main) để bạn đọc tham khảo. (Code nộp AC)









► Code

Bài viết tham khảo và bài tập luyện tập

Bài viết trên được tham khảo từ bài viết gốc [Hybrid Tutorial 1: Heavy-Light Decomposition](#) . Bạn đọc có thể tham khảo và xem video hướng dẫn kèm theo của galen_colin. Ngoài ra có thể tham khảo bài viết của [CP algo](#) .

Trước hết, bạn đọc nên tự cài đặt và nộp bài tập phía trên tại [đây](#) . Sau đó có thể làm thêm các bài tập luyện tập dưới đây

- [Path Queries \(CSES\)](#)
- [LUBENICA \(SPOJ\)](#)
- [QTREE \(SPOJ\)](#)
- [GRASSPLA \(SPOJ\)](#)

- [GSS7 \(SPOJ\)](#) 
- [QRYLAND \(CodeChef\)](#) 
- [MONOPLOY \(CodeChef\)](#) 
- [QUERY \(CodeChef\)](#) 
- [BLWHTREE \(CodeChef\)](#) 
- [Milk Visits \(USACO\)](#) 
- [Max Flow \(USACO\)](#) 
- [Exercise Route \(USACO\)](#) 

Được cung cấp bởi [Wiki.js](#)