Algoritmos sobre árboles

Marcos Kolodny

FAMAF-UNC

November 3, 2019

Lowest common ancestor

Dado un árbol rooteado, un vértice es ancestro de otro si se encuentra en el camino entre él y la raíz (un nodo es ancestro de si mismo).

Lowest common ancestor

Dado un árbol rooteado, un vértice es ancestro de otro si se encuentra en el camino entre él y la raíz (un nodo es ancestro de si mismo).

Llamaremos LCA (Lowest Common Ancestor) al primer ancestro en cómun que tengan dos (o más) nodos.

Por simplicidad, vamos a asumir que siempre vamos a buscar el LCA entre dos nodos, y que la raíz es siempre el nodo 1.

Lowest common ancestor

Una estrategia es computar el padre de cada nodo mediante un dfs, así como también su profundidad en el árbol. Luego, suponiendo que queremos encontrar el LCA entre los nodos X e Y, podríamos ir "subiendo" por los padres (siempre haciendo escalar al nodo que está en un nivel más profundo) hasta que ambos índices sean iguales.

Lowest common ancestor

Una estrategia es computar el padre de cada nodo mediante un dfs, así como también su profundidad en el árbol. Luego, suponiendo que queremos encontrar el LCA entre los nodos X e Y, podríamos ir "subiendo" por los padres (siempre haciendo escalar al nodo que está en un nivel más profundo) hasta que ambos índices sean iguales.

La complejidad de hacer esto es $O(n)$, donde n es la cantidad de nodos del árbol. Suena bastante bien. Pero ahora supongamos que tenemos muchas queries. Podemos hacerlo mejor.

Lowest common ancestor

Una estrategia es computar el padre de cada nodo mediante un dfs, así como también su profundidad en el árbol. Luego, suponiendo que queremos encontrar el LCA entre los nodos X e Y, podríamos ir "subiendo" por los padres (siempre haciendo escalar al nodo que está en un nivel más profundo) hasta que ambos índices sean iguales.

La complejidad de hacer esto es $O(n)$, donde n es la cantidad de nodos del árbol. Suena bastante bien. Pero ahora supongamos que tenemos muchas queries. Podemos hacerlo mejor.

Si pudiésemos mantener todos los ancestros para cada nodo, podríamos hacer un estilo de binary search para encontrar el primer ancestro en común. Lamentablemente no podemos guardar explícitamente esa información.

Sparse table

Vamos a utilizar una matriz conocida como "Sparse table". Es muy útil en muchos problemas, y en particular se puede utilizar para obtener el LCA entre dos nodos rápidamente.

Supongamos que tenemos computado el padre directo de cada nodo. ¿Cómo puedo saber el segundo ancestro de un nodo?

Sparse table

Vamos a utilizar una matriz conocida como "Sparse table". Es muy útil en muchos problemas, y en particular se puede utilizar para obtener el LCA entre dos nodos rápidamente.

Supongamos que tenemos computado el padre directo de cada nodo. ¿Cómo puedo saber el segundo ancestro de un nodo?

Es el padre de mi padre. Ahora bien, si quisiera saber el cuarto nodo en mi camino hacia la raiz, ¿Qué podría hacer?

Sparse table

De esta forma, podemos computar el 2^x-ésimo padre para cada nodo, siempre aprovechando que ya computamos el 2^{x-1}-ésimo.

```
1 parent[vertex][pw]=parent[parent[vertex][pw-1]][pw-1]
```

Luego, computar esto para todos los nodos es $O(n * log(n))$.

Sparse table

Esta estructura también es muy útil en problemas en los que se realizan queries de rangos sobre arreglos con operaciones asociativas, como por ejemplo suma, mínimo, máximo, etc.

Lowest common ancestor

Volviendo a nuestro problema, podemos utilizar una sparse table sobre los padres para obtener el LCA entre dos nodos en $O(log(n))$:

```
int lca(int x, int y){ //dep(x)>=dep(y)
    int diff=dep[x]-dep[y];
    //igualar profundidades
    for(int pw=mx-1; pw>=0; pw--){
        if(diff & (1<<pw)) x=parent[x][pw];
    }
    for(int pw=mx-1; pw>=0; pw--){
        if(par[x][pw]!=par[y][pw]){
            x=parent[x][pw];
            y=parent[y][pw];
        }
    }
    return parent[x][0];
}
```

Heavy-Light decomposition

Dado un árbol donde cada nodo tiene un valor, queremos realizar dos tipos de queries:

- Cambiar el valor de un nodo.
- Preguntar suma de los nodos en el camino entre X e Y.

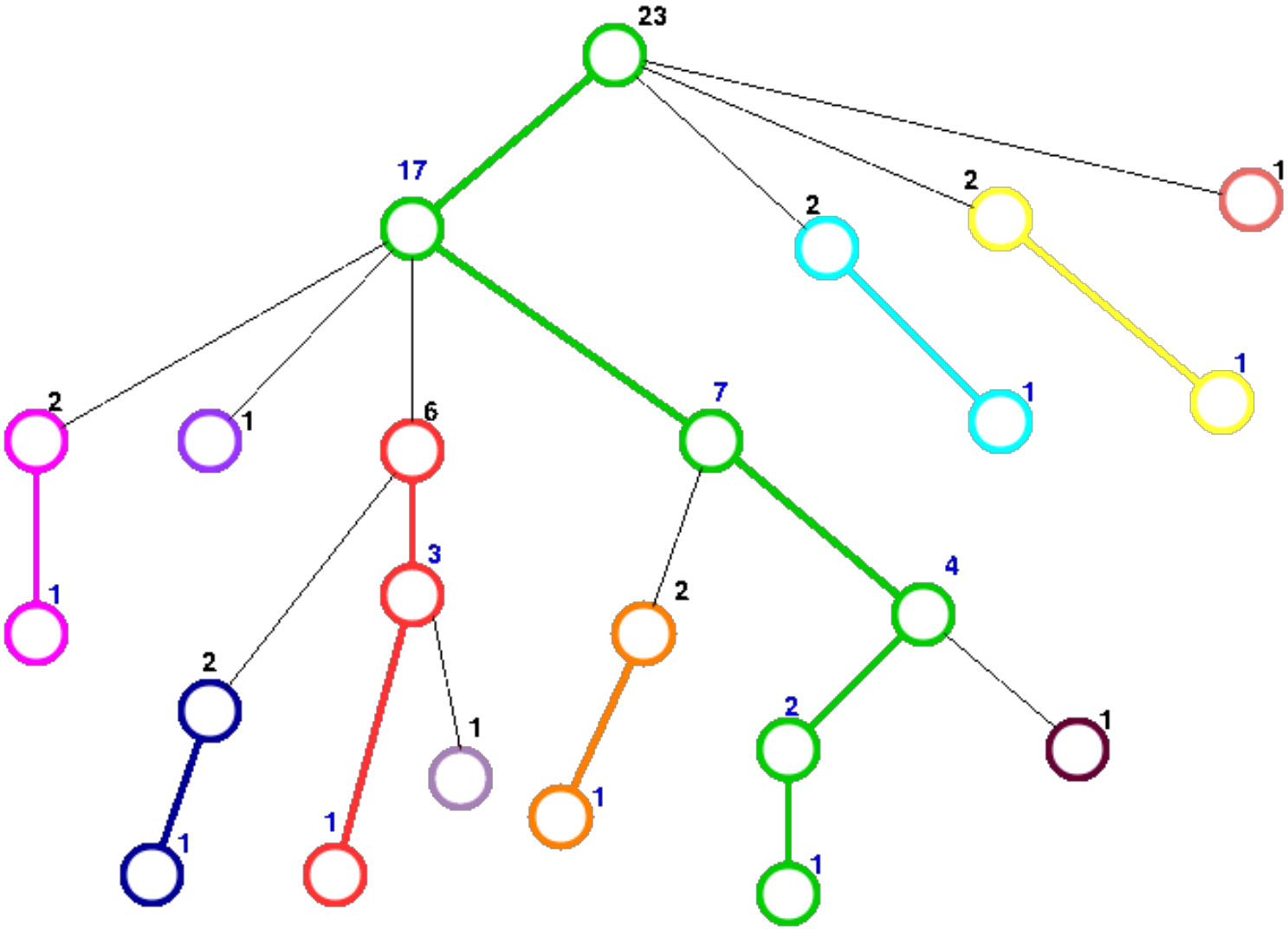
En otras palabras, lo que queremos es usar un segment tree "sobre árboles".

Heavy-Light decomposition

La técnica de Heavy Light decomposition consiste en:

- Rootear arbitrariamente el arbol en un nodo, y precalcular el tamaño del subárbol de cada nodo (mediante un dfs simple).
- Dividir las aristas del árbol en dos conjuntos: las aristas "heavy" y las aristas "light".
- para cada nodo que no sea una hoja, seleccionamos como "heavy" a la arista que nos dirige hacia el hijo que tenga mayor tamaño de subarbol, y al resto las asignamos como "light".

Heavy-Light decomposition



Heavy-Light decomposition

Gráficamente, lo que realizamos es una partición de las aristas en "cadenas" de lados "heavy". Esta partición tiene una propiedad clave:

En el camino desde un nodo hacia la raíz, atrevesaremos pocas aristas "light". ¿Por qué?

Heavy-Light decomposition

Gráficamente, lo que realizamos es una partición de las aristas en "cadenas" de lados "heavy". Esta partición tiene una propiedad clave:

En el camino desde un nodo hacia la raíz, atrevesaremos pocas aristas "light". ¿Por qué?

Al transicionar de un nodo X a su padre mediante una arista "light", el tamaño del subarbol que consideramos es al menos el doble del anterior (porque ahora incluímos el subarbol de la arista "heavy" del padre, que sabemos que es de tamaño mayor o igual al del nodo X). Esto nos permite concluir de que la cantidad de aristas "light" en el camino hacia la raiz es $O(log(n))$.

Heavy-Light decomposition

Luego, podemos tener un segment tree para cada cadena, y podemos responder una query en $O(log(n)^2)$.

El algoritmo sería:

- Un update es simplemente actualizar el valor en el segment tree correspondiente a la cadena del nodo.
- Para la query, podemos subir desde cada uno de los nodos hasta el LCA mientras computamos el resultado. Subir en una misma cadena es hacer una query del segment tree correspondiente. Si necesitamos seguir subiendo, saltamos a la cadena "padre".

Heavy-Light decomposition

Para la implementación:

- es mucho más simple utilizar un único segment tree, y "mapear" los indices de los nodos a posiciones consecutivas si pertenecen a la misma cadena.
- Es útil computar la posición relativa de cada nodo en su cadena, así como también la "cabeza" de cada cadena.
- Si además guardamos la profundidad de cada nodo en el árbol, no necesitamos utilizar la sparse table para calcular el LCA.

Heavy-Light decomposition

Si en el problema original son las aristas las que tienen peso, el problema es esencialmente el mismo:

- Le ponemos el valor al nodo que está mas abajo de los dos.
- El nodo raíz tiene un valor cualquiera.
- Hay que ignorar el LCA en la query.

DSU on tree

Dado un árbol rooteado donde cada nodo tiene un color, hay que responder queries de "cantidad de nodos con un color C en el subarbol del vértice X".

Una forma simple es computar la respuesta mediante un dfs de cada nodo, pero esto es $O(n * q)$. Como podrán suponer, se puede hacer mejor.

DSU on tree

La idea es responder las queries offline mediante una técnica llamada "DSU on tree". Hay varias formas de implementar este truco. En esta clase veremos una sola, pero hay muchos links en Internet donde se pueden encontrar las variantes.

DSU on tree

Vamos a computar para cada nodo un map que contenga la información de la cantidad de veces que aparece cada color en el subarbol.

DSU on tree

Vamos a computar para cada nodo un map que contenga la información de la cantidad de veces que aparece cada color en el subarbol.

Calcular el map para un vértice, es "mergear" los maps de sus hijos, y sumar 1 a la key del color del nodo actual.

DSU on tree

Vamos a computar para cada nodo un map que contenga la información de la cantidad de veces que aparece cada color en el subarbol.

Calcular el map para un vértice, es "mergear" los maps de sus hijos, y sumar 1 a la key del color del nodo actual.

Esto a simple vista puede ser cuadrático en el peor caso.. ¿No?

DSU on tree

Vamos a computar para cada nodo un map que contenga la información de la cantidad de veces que aparece cada color en el subarbol.

Calcular el map para un vértice, es "mergear" los maps de sus hijos, y sumar 1 a la key del color del nodo actual.

Esto a simple vista puede ser cuadrático en el peor caso.. ¿No?

Bueno, resulta que se puede hacer un truco super simple para evitar ese peor caso.

DSU on tree

Supongamos que estamos computando el map del nodo actual (de nombre $M1$), y queremos agregar la información de uno de sus hijos (de nombre $M2$).

El truco consiste en mergear los valores del map mas chico hacia el de mayor tamaño.

En otras palabras, si el tamaño de $M2$ es mayor al de $M1$, swapeamos los maps. Luego mergeamos $M2$ a $M1$.

DSU on tree

Ahora bien, ¿Por qué esto mejora la complejidad?

DSU on tree

Ahora bien, ¿Por qué esto mejora la complejidad?

Analicemos el tamaño de $M1$ con respecto a $M2$ en el peor caso (cuando todos los colores son distintos).

Resulta que luego de mergear $M2$ en $M1$, $M1.size() >= 2 * M2.size()$ (porque previamente $M1$ era más grande que $M2$).

¡Pero entonces, un elemento que estaba en $M2$, ahora está en un map de al menos el doble de elementos! Y esto nos dice que este elemento a lo sumo se mergea a un map más grande $O(log(n))$ veces. Por lo que concluímos que al haber n nodos, la complejidad de ir computando los maps utilizando los de sus hijos es $O(n * log(n)^2)$.

DSU on tree

Ahora responder las queries de manera offline es simple: luego de computar el map correspondiente a un nodo, respondemos las queries que se hicieron sobre ese vértice preguntando por la cantidad de veces que un color aparece en el map.

DSU on tree

```
//who[x] = indice del map para el nodo x

void merge(int &x, int &y){    //mergear y en x
    if(cnt[x].size() < cnt[y].size()) swap(x,y);
    for(auto node: cnt[y]){
        cnt[x][node.first]+=node.second;
    }
}

void dfs(int vertex, int parent){
    for(int y:g[vertex]){
        if(y==parent) continue;
        merge(who[vertex],who[y]);
    }
    //responder queries usando el map who[vertex]
}
```

Dudas/preguntas

¿Preguntas?