Lowest Common Ancestor (най-нисък общ предшественик)

Съдържание:

Дадено е дърво G. Дадени са заявки от вида (v_1, v_2), където за всяка заявка трябва да намериме най-ниския общ предшественик (или най-малкия общ предшественик), т.е. връх v, който лежи на пътя от коерна до v_1 и на пътя от корена до v_2 и върхът трябва да е най-ниският. С други думи, търсеният връх v трябва да е най-долният предшественик на v_1 и v_2. Очевидно е, че техния най-нисък предсшественик се намира на най-краткия път между от v_1 и v_2. Също така, ако v_1 е предшественик на v_2, то v_1 е техния най-нисък общ предшественик.

Идея на алгоритъма

Преди да започнем да отговаряме на заявките, трябва да обработим предварително дървото. Ще направим DFS обход, започвайки от корена и ще изградим списък $euler$, който съхранява реда на върховете, които посещаваме (връх се добавя към списъка, когато го посетим за първи път и след DFS обхода на неговите деца). Това обхождане в дълбочина се нарича още обиколка на Ойлер из дървото. Ясно е че размера на този списък от върхове ще нараства линейно спрямо броя на върховете, т.е. $O(N)$. Ако искаме да сме още по-прецизни може да сметнем точния размер на този списък от върхове:

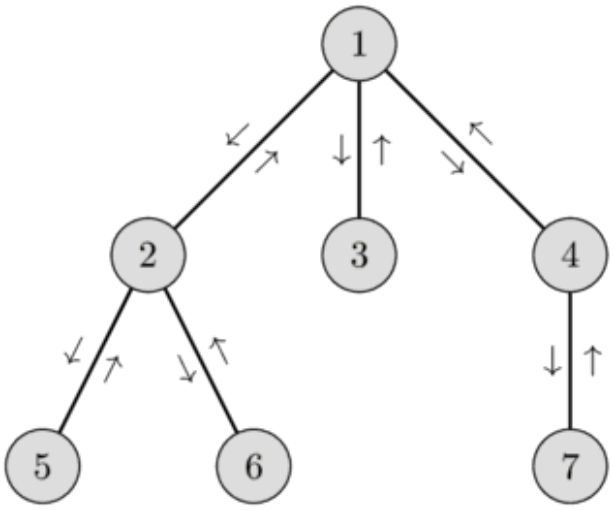
(*За всяко дърво е изпълнено* $|V| = |E| + 1$*, където с* $|V|$ *е броя на върховете, а* $|E|$ *броя на ребрата на дървото. Имаме, че* $|V| = N$ *и следователно броя на ребрата е* $N - 1$*. Сега, ако преобразуваме всяко ребро в две насочени ребра ще удвоим броя на ребрата и ще станат точно* $2(N - 1)$*. По дефиниция общия брой на върховете в ойлеров обход в дървото е* $2(N - 1) + 1 = 2N - 1$*.)*

Сега нека се върнем отново на алгоритъма. Необходимо ни е да построим масив $first[0..N - 1]$, който съхранява първото появяване на всеки връх v в обиколката на Ойлер. Това ще е първата позиция в $euler$, такава че $euler[first[i]] = i$. Също така, използвайки DFS ще намерим височината на всеки връh (разстоянието от корена до върха) и ще го съхраним в масив $height[0..N - 1]$.

И така, как можем да отговорим на заявки, използвайки обиколката на Ойлер и допълнителните два масива? Да предположим, че заявката е двойка от върхове v_1 и v_2. Разглеждаме върховете, които посещаваме в ойлеровия път между първото посещение на връх v_1 и първото посещение на връх v_2. Лесно се вижда, че $LCA(v_1, v_2)$ е върха с най-малка височина в този откъс от пътя. Вече забелязахме, че LCA трябва да бъде част от най-краткия път между v_1 и v_2. Ясно е, че това също трябва да е върхът с най-малка височина. В обиколката на Ойлер ние по същество използваме най-краткия път, с изключение на това, че допълнително посещаваме всички поддървета, които намерим по пътя. Но всички върхове в тези поддървета са по-ниски от LCA в дървото и следователно имат по-голяма височина. Така

$LCA(v_1, v_2)$ може да се определи уникално чрез намиране на върха с най-малката височина в обиколката на Ойлер между $first[v_1]$ и $first[v_2]$.

Нека илюстрираме тази идея. Разглеждаме следното дърво и обиколката на Ойлер със съответните височини:



Vertices:	1	2	5	2	6	2	1	3	1	4	7	4	1
Heights:	1	2	3	2	3	2	1	2	1	2	3	2	1

В обиколката започваща от връх 6 и завършваща във връх 4, посещаваме върховете $[6, 2, 1, 3, 1, 4]$. Измежду тези върхове, връх 1 е с най-малка височина, следователно $LCA(6, 4) = 1$.

За да обобщим: за да отговорим на заявка, просто трябва **да намерим върха с най-малка височина** в масива $euler$ в интервала от $first[v_1]$ до $first[v_2]$. По този начин **проблемът за намиране на LCA се свежда до проблема за намиране на RMQ** (Range Minimum Query - заявка за намиране на минимум в даден интервал).

Това може да се направи по редица различни начини:

- Използвайки $Sqrt\ Decomposition$, е възможно да се получи решение отговарящо на всяка заявка за $O(\sqrt{N})$ времева сложност с $O(N)$ времева сложност за предварителна обработка https://cp-algorithms.com/data_structures/sqrt_decomposition.html.
- С помощта на сегментно дърво може да отговорим на всяка заявка за $O(log(N))$ времева сложност с $O(N)$ времева сложност за предварителна обработка https://cp-algorithms.com/data_structures/segment_tree.html.
- Ако допуснем, че много рядко (почти никога) ще има актуализация на съхранените стойности (т.е. няма да добавяме нови върхове), разредената таблица (Sparse Table) може би ще бъде най-добрия избор, тъй като ще позволява $O(1)$ времева сложност за отговаряне на заявка с $O(N . log(N))$

времева сложност за предварителна обработка https://cp-algorithms.com/data_structures/sparse-table.html.

Тук може да намерите имплементация на алгоритъма за LCA, която използва сегментно дърво: https://cp-algorithms.com/graph/lca.html.

Но решенията, които ще имплементирам в задачите ще използват разредена таблица (Sparse Table) за RMQ.

github.com/andy489