Алгоритмы и структуры данных. Домашняя работа. Неделя 9

Автор: Петровский Александр M3139

Задание 1

Пусть каждый запрос представляет из себя пару {v, l}. Для решения этой задачи офлайн заведем массив a, в котором будем хранить вершины, которые сейчас обрабатываются дфс в порядке обхода. Тогда пусть мы находимся в некоторой вершине, которая присутствовала в запросе. В данный момент у нас будет массив a, в котором уже есть sz вершин (по сути это путь от корневой вершины до текущей). Тогда чтобы узнать l-ого предка, мы посмотрим на элемент a[sz – 1 – l] (нумерация с 0). Очевидно, что это и будет ответом на запрос.

Ниже представлен псевдокод решения:

a - array with vertex in dfs's stack

dfs(v) :

| **for** q - query with current vertex

| | ans[q.numer] = a[a.sz - 1 - q.l];

| a.add(v)

| **for** c - child of v

| | dfs(v)

| a.del(v)

Задание 2

Сведение задачи LCA к O(log n) запросам LA вообще очень широко известна. Для этого заведем массив anс[N][LG], так, что в ans[i][j] будет хранится предок вершинки i на расстоянии (2 << j) от нее. Пересчет производится в дфс из соображения, что

dfs(v, prev) :

| anc[v][0] = prev

| **for** lg = 1..log(n)

| | anc[v][i] = anc[anc[v][i - 1][i - 1]

Рассмотрим иллюстрацию, на которой изображены две вершины внизу (a, b) – вершинки запроса, вершина сверху (p) – их наименьший общий предок. Заметим, что начиная p вершинки c все вершинки ближе к корню будут являться предками как a, так и b. Воспользуемся данным замечанием. Пусть мы находимся в вершинке a. Так как мы можем за единицу получать предка на расстоянии (2 << j), то будем подыматься вверх на (2 << k), такое, что новая вершинка не является предком b. И так будем подыматься пока можем. В итоге мы окажемся в вершинке, которая не является предком b, но prev[b] – является. Он и будет ответом на запрос.

lca(a, b) :

| **if** (a - ancestor b) **return** a

| **if** (b - ancestor a) **return** b

| **for** lg = log(n)...0

| | **if** **not** anc[a][lg] - ancestor

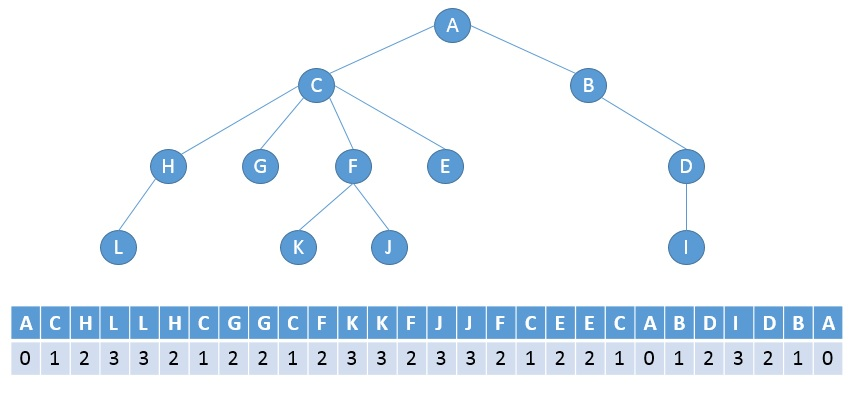
| | | a = anc[a][lg]

| **return** anc[a][0]

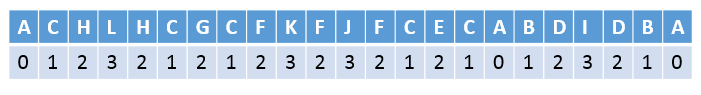
Проверку на предка можно делать с помощью времени входа/выхода дфс.

Задание 3

Данную задачу разобьем на две задачи: «поиск lca» и «минимум на пути»

1) Пусть задано дерево T и две вершины запроса v, u. Нужно найти lca(v, u). Тогда выпишем его эйлеров обход как пару (вершина, глубина). Например:

Известным фактом является то, что наименьшим общим предком двух вершин является вершина, которая обладает минимальной высоте между ними в эйлеровом обходе. Воспользуемся этим, но аккуратно. Известно, что sparse table умеет отвечать на запрос за , но требует на прекальк. Разобьем ячейки на блоки размером и на них построим Sparse Table. Так как таких ячеек будет , то прекальк будет работать за . Таким образом, если вершинки находятся в разных блоках. Таким образом, если между вершинками v и u в эйлеровом обходе несколько обходов, то в них минимум мы можем найти за , в итоге сейчас ответ на запрос , где – оценка времени поиска минимума в блоке. Нужно свести к константе, чтобы отвечать на запрос за Для начала заметим, что в эйлеровом обходе нет смысла хранить две подряд одинаковые пары (вершина, высота, поэтому ужмем его). В примере:

**

Также заметим, что для его записи мы можем использовать бинарный алфавит по высотам: 1 – спустились вниз, 0 – поднялись вверх. Это значит, что мы можем закодировать эйлеров обход. Сколькими способами можно закодировать один блок, который тут определили: . Таким образом мы можем предпосчитать каждого блока индекс минимальной высоты на суффиксе, преффиксе и на целом блоке. Таким образом мы сделаем предпосчет всех возможных типов блоков за . Далее для каждого блока мы определяем тип блока и можем с ним спокойно за . Таким образом мы научились искать lca двух вершин.

2) Для решения поставленной задачи воспользуемся своего рода делением задачи на две неравные (маленькие и большую). Так как мы хотим делать прекальк за , то мы можем без проблем для начала построить long path decomposition. Легко заметить, что если бы была у нас уже была предпосчитана таблица для двоичного подъема, то мы могли бы отвечать за на запрос. Это делается следующим образом: Пусть нам надо найти k-ого предка вершины v, которая находится на расстоянии d от корня. Тогда, если мы сделаем переход к вершинке на расстоянии от v, где . Тогда находясь в этой вершине мы за единицу сможем воспользоваться long path decomposition, которая будет хранить как минимум предков пути, что, очевидно, будет включать и ответ. Известно, что прекальк для двоичного подъема работает за для одной вершины, поэтому будем считать его не для всех вершин, а для , а значит в результате это будет . Вершины, для которых мы будем считать двоичный подъем назовем прыг-вершинами. Так как мы не знаем информацию о родителях, то будем искать предков с помощью long path decomposition, каждый раз подымаясь выше на расстояние вдвое большее, чем ранее. Очевидно, что это также будет выполняться за для каждой вершины. Такими вершинами будут максимальной далекие вершины, у которым размер поддерева не более log(n) / 4. Такой размер поддерева взят из соображения, что всего возможных поддеревьев для такого количества будет , поэтому тут мы можем предпосчитать все за = . Таким образом мы умеем отвечать на запросы в том случае, если вершинка запроса и ответа находятся либо в маленьком (за , либо в большом (за . Если вершинка запроса попала в маленькое поддерево, а ответ в большом, то мы просто перейдем в прыг-вершину и из нее посчитаем ответ за .

Итого, мы умеем находить k-ого предка за . Для того, чтобы отвечать на запрос минимального ребра на пути от v к u, мы будем в двоичном подъеме вместе с anc[v][i] хранить min\_e[v][i] – минимальное ребро на пути от v, до -ого предка и ребро выше (для корня будем считать, что ребро выше имеет бесконечный вес). Пересчет:

min\_e[v][0] = e[v][prev[v]].weight

**for** lg = 1..log(n)

| min\_e[v][i] = min(min\_e[v][i - 1], min\_e[anc[v][i - 1]][i - 1])

В long path decomposition мы также помимо h предков цепи (h – размер цепи), будем хранить для каждого элемента из предков цепи минимальное ребро на пути от корня цепи, то рассматриваемой. Очевидно, это считается за линию.

ladder[chain][i] - i-th ancestor for path with number chain

min\_e[chain][i] - min edge on the way from chain\_root to ladder[chain][i]

ladder[chain][0] = chain\_root

min\_e[chain][0] = e[chian\_root][prev[chain[root]]].weight

**for** i = 1..chain\_size

| ladder[chain][i] = p[ladder[chain][i - 1]]

| min\_e[chain][i] = min(min\_e[chain][i - 1], e[ladder[chain][i]][ladder[chain][i - 1]].weight)

Тогда во время поиска предка на расстоянии d мы одновременно будем считать и минимальное на ребро на пути между ними.

Теперь, мы можем находить lca(v, u) за O(1), а потом находить минимальное ребро, так как легко можем найти расстояния между lca(v, u) и каждой из вершин. Задаче решена.

Задача 4