Студент: Тимур Хабибуллин
Группа: 1
Дата: 5 мая 2022 г.

---

1. (a) Пусть в G есть какое-то остовное дерево T, которое можно получить с помощью алгоритма Краскала, расставив ребра не в порядке неубывания. Тогда существует какое-то ребро e_k (пусть это первое ребро в порядке убывания на позиции $k-$е в списке ребер `lst`, по которому шел алгоритм, т.е. все предыдущие $k-1$ ребро были в нужном порядке) такое, что e_{k-1} было тяжелее, чем e_k. Если алгоритм взял e_k, значит, оно соединяет вершины в разных компонентах связности на шаге k. Однако, это значит, что если это ребро поставить куда-то на позицию $m < k$ в списке `lst`, то оно все так же будет соединять вершины из разных компонент связности и алгоритм его возьмет в ответ. Тогда поставим это ребро на позицию, в которой оно будет идти в порядке неубывания. Сделаем так для всех ребер, которые стоят в неправильном порядке, тогда мы ничего не сломаем, но расставим ребра в нужном прядке.

   Теперь другая ситуация - пусть в `lst` ребра стоят в правильном порядке, но в нем есть ребра одинаковой длины, тогда они будут стоять подряд. Алгоритм в этой ситуации будет брать самое первое среди них ребро e_k, поэтому может так получиться, что мы не возьмем какое-нибудь ребро $e_l,\ l > k$, такой же длинны, соединяющее те же компоненты связности, что и e_k, относящееся к другому остовному дереву, но стоящее после e_k. Тогда чтобы алгоритм смог взять это ребро и получить другое остовное дерево, можно просто поменять местами e_k и e_l. В более общей ситуации, необходимо переставить ребра одинаковой длинны так, чтобы алгоритм взял нужные.

   (b) По предыдущему пункту, можно упорядочить ребра по неубыванию так, чтобы алгоритм выдавал нужное остовное дерево, т.е. число деревьев не больше чем число перестановок. Однако если в графе все ребра имеют разный вес, тогда существует единственная расстановка ребер по неубыванию. Значит, если в графе существует MST, то оно единственно.

   (c) Из первых двух пунктов можно заключить, что если в графе есть несколько остовных деревьев, то существуют ребра одинаковой длины, т.е. все неоднозначности могут возникнуть из-за ребер одинаковой длины, с ними то и будем разбираться. Отсортируем все ребра по возрастанию $O(E\log(V))$, найдем в отсортированном `lst` ребра одинаковой длины (например, за линию найдем все такие "участки") и начнем собирать остовное дерево. Когда дойдем до очередного "участка необходимо посмотреть, нет ли среди них ребер, концы которых относятся к одинаковым компонентам связности. Воспользуемся СНМ на лесе — каждой компоненте связности соответствует корневая вершина, поэтому компонты связности легко отличать.
   Заведем некоторый `set` и будем сохранять в нем неупорядоченные пары из номеров (т.е. корневых вершин) компонент связности, соотвествующих концам ребра из "участка". Будем перебирать ребра из "участка"и на каждом шаге будем проверять, есть ли данная неупорядоченная пара в `set` и если да то, то тогда MST не единственный, иначе добавляем и идем дальше. Если мы переберм все ребра из "участка"и ничего не обнаружим, тогда еще раз пройдемся по участку и объединим все компоненты связности, как в обычном алгоритме Краскала и пойдем дальше.

2. От противного — пусть мы проделали данный алгоритм и получили какое-то T, которое не является MST, тогда

$$\omega(T) > \omega(\mathrm{MST})$$

Заметим, что после работы алгоритма мы получили связное дерево, потому что мы удаляли ребра до тех пор, пока не терялась связность, т.е. у нас точно не осталось циклов.
Рассмотрим какое-нибудь ребро $e \in T$, которого нет в MST. В ходе работы алгоритма мы не удалили это ребро, значит, если бы мы удалили, то потеряли бы связность. Пусть e соединяет два подмножества вершин $A, B \subset V$. Рассмотрим ребро $g \in$MST, которое тоже соединяет эти подмножества. Оно существует, потому что иначе в MST не было бы связности, и единственно, потому что иначе в нем был бы цикл. В ходе работы алгоритма мы удалили ребро g раньше, чем рассмотрели e, потому что иначе мы бы удалили e. Значит,

$$\omega(e) \leqslant \omega(g).$$

Получается, что это верно для всех ребер $g \in T$. Но тогда мы приходим к противоречию, т.к. вес дерева T оказывается не больше, чем у MST:

$$\omega(T) \leqslant \omega(\text{MST})$$

Значит, T есть MST.

3. Пусть сделали m_1 запросов `join`, m_2 запросов `get`. Все операции `join` работают за $O(1)$, потому что мы просто берем корни и цепляем за что попало. Тратим на все `join` не более $O(m)$ времени. При этом каждая вершина может быть подвешена один раз, так что максимум мы подвесим $n_1 \leqslant n$ вершин.
Пусть теперь мы делаем `get(v)` от какой-то вершины, тогда при его работе все промежуточные вершины от v до корня дерева `root(v)`, к которому было прицеплена `v`, переподвесятся к нему и больше никогда не будут обрабатываться. Т.е. каждая вершина может поучаствовать в `get` не более одного раза. Значит, если мы при вызовах `join` подвесили n_1 вершин, то максимальное время работы всех оставшихся `get` есть $O(n_1) + O(m)$ за счет того, что какие-то `get` могут отработать вхолостую за $O(1)$, $m_2 \leqslant m$, так что их не более чем m. Однако, поскольку $n_1 \leqslant n$, то все `get` отработают не более чем за $O(m + n)$ времени.
В итоге все m запросов отработают за $O(m + n)$.

4. Сделаем следующим образом — будем использовать СНМ на лесе, с ранговой эвристикой и сжатием путей, а также заведем дополнительный массив `d`, в котором будем хранить очки, но будем это делать необычным образом: для вершины дерева мы будем хранить число очков, которое принадлежит непосредственно ему, а для листьев — поправки к количеству очков относительно его родителя. Т.е. если изначально при инициализации каждая вершина — есть корень, поэтому для них храним свои значения очков. Затем хотим сделать `join` v_1 и v_2, тогда мы выбираем какая вершина будет корнем (как это происходит в оригинальном `join`) и цепляем, например, v_2 за v_1, тогда `p[v2]= v1`, `d[v2]=d[v2]-d[v1]`, делаем ранг v_1 на единицу больше. Если например присоединить к этому дереву еще и v_3, которое имеет ранг 1, то (как это происходит в оригинальном `join`) цепляем к дереву с большим рангом (v_1, v_2) дерево с меньшим (v_3). И тогда делаем `p[v3]= v1`, `d[v3]=d[v3]-d[v1]`. Т.е. когда мы делаем `join`, то мы делаем все то же самое, что и раньше, но еще для корня подцепляемого дерева обновляем значение очков как разность между старым значением и корнем другого дерева.

Чтобы получить значение очков `get(v)`, мы так же делаем все то же самое, что и в эвристике сжатия путей - переподвешивать все промежуточные вершины к корню, но при этом на каждом шаге складываем текущее значение опыта `d[v]` с опытом его родителя: `d[v]`$+=$ `d[p[v]]`. Т.е. пока мы не добрались до корня, мы будем складывать поправки, но когда дойдем до корня и сложим все промежуточные поправки с значением в корне, то получим истинное значение опыта вершины. Например, если бамбук $v_1 \to v_2 \to v_3$, где v_1— корень, то когда мы будем считать `d[v3]`, мы получим такую последовательность:

```
d[v3] = d[v3]+(d[v2]+d[v1])=d[v3]+(ист. кол. очков v2)= (ист. кол. очков очков v3)
```

После того, как дойдем до корня и найдем истинное количество очков, мы можем вывести его, а потом надо очки для (в нашем примере) v_3 снова записать обратно в формат поправки относительно корня, это легко сделать, потому что мы уже переподвесили v_3 к корню v_1.

Чтобы увеличить число опыта в дереве, в котором есть вершина v, достаточно просто найти корень этого дерева (т.е. проделать `get(v)` без вывода количества очков для v), добавить необходимое число очков к корню дерева и все, никакие поправки для вершин в дереве пересчитывать не нужно, изменение очков для любой вершины дерева пересчитается автоматически, потому что мы изменили значение в опорной значении (т.е. в вершине).

Мы на лекциях показывали, что со всеми этими эвристиками m операций над СНМ работает за $O((n+m)\log(n))$, здесь же мы по сути ничего не поменяли, просто завели доп массив, который как-то изменяется по ходу работы оригинального СНМ.

Чтобы сделать за $O((n+m)\alpha(n))$, надо внести такие же изменения в алгоритм работы СНМ, который работает за это же время.