Студент: Маргарита Чудова
Группа: 2
Дата: 27 апреля 2022 г.

---

1. (a) Псевдокод алгоритма Краскала выглядел как-то так:

```
all_edges = [(v, u, w)]
sort(all_edges, key = w)
for v, u, w in all_edges:
if v, u в разных компонентах связности:
    добавить ребро между ними в ответ
```

   Различные MST могут отличаться ребрами одинакового веса. На самом деле за то, какое ребро возьмет алгоритм, отвечает часть с отсортированным массивом — какое ребро попадется первым, то и возьмет. Для того, чтобы взять конкретное MST, нужно all_edges отсортировать так, чтобы ребра из этого MST шли первыми среди ребер с равным им весом. Тогда мы не сможем взять не входящие в нужный MST ребра — за счет того, что нужные будут взяты в первую очередь, ненужные будут образовывать циклы.

   (b) В таком случае, упорядочить ребра по неубыванию в алгоритме Краскалла можно единственным образом, а значит и MST единственное по предыдущему пункту.

   (c) Понятно, что в графе может быть несколько MST (но не обязательно будут), если в нем несколько ребер одинаковой длины. Отсортируем за $O(E \log V)$ все ребра по неубыванию и найдем в этом массиве все ребра одинакового веса.
   Запустим Краскала, и для всех ребер с одинаковым весом будем сохранять пары компонент связности (их можно быстро достать, если использовать CHM на деревьях со всеми эвристиками), к которым относятся концы этого ребра (например в set, чтобы тоже быстренько доставать). Если пара компонет связности очередного ребра уже лежит в set, то MST не единственный, если ничего такого не произошло, то продолжаем построение дерева дальше.

2. Заметим, что алгоритм остановит свою работу тогда, когда останется какое-то остовное дерево. Почему? Потому что на каждом шаге граф остается связным, при этом из дерева мы не сможем выкинуть ни одного ребра, не нарушив связность.

   Допустим, это дерево — не MST (далее под обозначением MST будем понимать любое минимальное остовное дерево из существующих в графе), назовем его T. Тогда как минимум на каком-то одном шаге алгоритма мы выкинули какое-то ребро из MST и при этом оставили другое, которое останется в T.
   Рассмотрим первый такой шаг. Пусть на этом шаге у нас есть осталось подмножество ребер E, которое содержит и T, и MST, и мы выкидываем ребро $e \in$ MST. Тогда, поскольку в итоге мы должны получить связное дерево, в дереве T существует ребро e', такое, что e' связывает две компоненты связности, на которые распадается дерево MST после удаления ребра e. При этом гарантируется, что $\omega \geq \omega'$, так как иначе наш алгоритм бы выкинул e' раньше. Аналогично будет на каждом шаге.

   Таким образом получаем, что для всех ребер дерева T e'_i и для всех ребер MST e_i выполнено

$$\omega'_i \leq \omega_i$$

Тогда и суммарный вес T меньше или равен суммарному весу MST. Если он получился меньше, то по определению MST получаем противоречие, а если равным, то T тоже является MST, просто идет не по тем ребрам, которые мы выбрали изначально.

3. Операции join очевидно отработают за $\mathcal{O}(m)$ — каждая отдельная за $\mathcal{O}(1)$ работает.
Перейдем к оценке времени работы get. Допустим, мы сделали get для какой-то вершины, находящейся на глубине h. Тогда этот get отработает за $\mathcal{O}(h)$, при этом все h предков перевесятся и для них операция get впредь будет выполняться не более чем за $\mathcal{O}(1)$. Для следующей вершины get будет работать не более, чем $\mathcal{O}(n-h)$ — просто даже в худшем случае не хватит вершин, чтобы опуститься глубже. И так далее, если просуммировать, все get с вершинами, не подвешенными к корневой, точно отработают за $\mathcal{O}(n)$, а за счет того, что у нас может быть много вызовов, которые будут дергать $\mathcal{O}(1)$ памяти, вызывая вершины, подвешенные к корню, вся асимптотика будет $\mathcal{O}(n+m)$

4. В качестве основы возьмем CHM, реализованную с помощью леса с ранговой эвристикой и эвристикой сжатия путей. Будем делать по сути то же самое, но дополнительно заведем массив, где будем хранить в корнях как бы заготовку опыта, равную опыту корневого игрока, а в остальных вершинах значения, которые из этой заготовки надо вычесть, чтобы найти количество очков листового игрока.

   Тогда при add мы действуем аналогично исходному get: мы идем в корень и добавляем опыт V к нему, при этом по дороге при переподвешивании вершин нам надо прибавлять к вершинам значение родителя.

   В join все аналогично исходному join, но мы вычитаем из корня дерева, которое мы подвесили, опыт корня дерева, к которому мы подвесили.

   get действует почти как add, но поднимаясь от вершины к корню, считает сумму значений массива с заготовками всех вершин на пути, и возвращает ее.