Поиск точек сочленения в режиме онлайн

Дан неориентированный граф. В нем точкой сочленения называется такая вершина, после удаления которой количество компонент связности увеличится. При этом граф может быть несвязным. Требуется найти все точки сочленения в графе (или их количество).

Задача звучит так: дан изначально несвязный набор из n вершин, в который поочередно добавляются ребра. После добавления каждого ребра необходимо вывести количество точек сочленения в этом графе.

Описываемый в данной статье алгоритм работает за $O((n+m)\log n * \alpha(n))$, где m – количество ребер в графе, и основан на структуре данных СНМ.

Структуры данных

Точки сочленения разбивают граф на компоненты вершинной двусвязности. Так что мы будем хранить две СНМ: для компонент связности всего графа и для компонент вершинной двусвязности. Также будем хранить граф G из точек сочленения и компонент вершинной двусвязности, построенный по следующему принципу:

- Каждая компонента вершинной двусвязности соединена ребром только с точками сочленения
- Точка сочленения соединена с компонентой если точка сочленения входит в эту компоненту (примыкает к ней)
- Точки сочленения соединены между собой, если они соединены ребром в глобальном графе и не находятся в одной компоненте вершинной двусвязности

Граф G подвешен и хранится неявно, в виде ссылки на родительскую вершину.

Переменные, которые используются в коде

Система непересекающихся множеств:

- `int parent[]` – массив предков в СНМ. `parent[root] = root`
- `int range[]` – массив рангов. Чтобы получить ранг компоненты, нужно дойти до корня
- `int size[]` – массив размеров множеств. Также хранится в корне

- `int ind[]` – массив индексов. В этом массиве хранится то, вершине с каким номером в СНМ соответствует вершина v в глобальном графе. Изначально `ind[v] = v`
- `int par_comp[]` – массив ссылок на родительскую вершину в графе G. В нем записан номер элемента, принадлежащей родительской компоненте
- `int max_ind` – максимальный индекс для данной СНМ. Изначально равен 0, изменяется при добавлении новых элементов

Мы создаем два элемента этой структуры: для хранения компонент связности (common) и компонент вершиной двусвязности (strong).

Глобальные переменные:

- `bool joint[]` – массив, в котором `joint[strong.ind[v]] = 1` если вершина `v` в глобальном графе является точкой сочленения
- `int tm` – счетчик, показывающий сколько раз ранее сжимали цикл
- `int used[]` – массив, показывающий при очередном сжатии цикла, обращались ли мы к данной вершине во время текущего сжатия
- `int cnt[]` – массив, показывающий, со сколькими вершинами в графе G соединена точка сочленения

Структура данных СНМ

Функции этой структуры практически идентичны функциям в статье, посвященной СНМ. Реализация:

```
struct Tree {
        int parent[N], range[N], ind[N], par_comp[N], size[N];
        int max_ind = 0;
        int make_set() {
                //создает новое множество и возвращает его номер
                parent[max_ind] = -1;
                range[max_ind] = 1;
                par_comp[max_ind] = -1;
                size[max_ind] = 1;
                return max_ind++;
        }

        int find_set(int v) {
                //возвращает корень множества, которому принадлежит v
                if (v == -1)return v;
                if (parent[v] == -1) return v;
                return parent[v] = find_set(parent[v]);
        }

        void union_sets(int a, int b) {
```

```
                //объединяет множества, которые содержат a и b
                a = find_set(a);
                b = find_set(b);
                if (a == b) return;
                if (range[a] < range[b]) swap(a, b);
                parent[b] = a;
                if (range[a] == range[b]) range[a]++;
                size[a] += size[b];
        }

        bool is_connected(int a, int b) {
                //проверяет, в одном ли множестве лежат a и b
                return find_set(a) == find_set(b);
        }
};

Tree common, strong;
```

Алгоритм

При добавлении ребра (a,b) возможны следующие варианты:

1. Если a и b находятся в одной компоненте вершинной двусвязности, ничего не меняется
2. Если a и b находятся в одной компоненте связности, но в разных компонентах вершинной двусвязности, то в графе G появляется цикл. В таком случае мы его находим и сжимаем, и G снова становится деревом.
3. Если a и b находятся в разных компонентах связности, то объединяем компоненты связности.

Поиск цикла

Сначала мы находим $lca(a, b)$, просто поднимаясь из a и b и фиксируя в `used` то, что мы посетили очередную вершину. Оказавшись в посещенной ранее вершине, мы возвращаем ее как lca.

Этот алгоритм работает за длину цикла, так как каждый из указателей поднимается на одинаковое количество вершин, а один из них останавливается в lca.

Сжатие цикла

Узнав lca, мы создаем новую компоненту вершинной двусвязности `new_set`, с которой объединяем все компоненты вершинной двусвязности, входящие в цикл. Изначально новая компонента состоит из одной фиктивной вершины. Также мы находим точку сочленения `up_joint`, к которой мы подвешиваем `new_set`.

После объединения всех компонент вершинной двусвязности мы подвешиваем все точки сочленения, входящие в цикл, кроме `up_joint`, к `new_set`. При этом каждый раз, когда мы объединяем компоненты или переподвешиваем точку сочленения, мы уменьшаем `cnt` в компоненте, которая была родителем ранее. Если `cnt` какой-то точки сочленения стало равно 1, мы ее удаляем и объединяем с компонентой, с которой она соединена.

На первый взгляд, данный алгоритм работает очень медленно, так как сжатие цикла после каждого запроса происходит за линейное время. Однако заметим, что приведенный алгоритм является по сути эвристикой сжатия путей, только здесь все вершины на пути подвешиваются не к корню, а к `new_set`. Однако в доказательстве времени работы СНМ с эвристикой сжатия путей, приведенном, например, здесь, нигде не используется, что мы подвешиваем вершины именно к корню. Нужно просто подвешивать вершины к самой верхней на пути, что и делает данный алгоритм. Заметим, что фиктивная вершина является единственной вершиной в своей компоненте вершинной двусвязности только в том случае, если она возникла при сжатии графа только из точек сочленения. Но тогда есть точка сочленения, которая подвешена к этой компоненте. Поэтому компонент вершинной двусвязности, в которые входят фиктивные вершины, не больше n. Так что всего вершин (включая фиктивные) $O(n)$. Значит, будет $O((m+n)\log n)$ переподвешиваний, каждое из которых работает за $O(\alpha(n))$, включая сюда время на переход между вершинами. Таким образом, поиск и сжатие графа суммарно работает за $O((m+n)\log n * \alpha(n))$

Создание точки сочленения

Чтобы сделать вершину v точкой сочленения, мы создаем новую вершину `new_joint` и говорим, что `ind[v] = new_joint`, после чего подвешиваем `new_joint` к той компоненте, в которой была `v` изначально, а `cnt[new_joint]` становится равным `1`. В результате все `par_comp` и `parent` остаются корректными. При том они, возможно ссылаются на фиктивную вершину, которая раньше соответствовала вершине `v`. Это не проблема. Так что добавление точки сочленения работает за $O(\alpha(n))$

Переподвешивание компоненты

Чтобы граф G находился в подвешенном состоянии, при объединении компонент связности при добавлении ребра (a, b) необходимо сделать компоненту, включающую в себя одну из этих вершин, корнем своей в компоненте связности. Это можно сделать за размер компоненты связности, переходя по `par_comp`. Будем каждый раз переподвешивать компоненту с меньшим размером. В таком случае, если элемент находится в компоненте, которую переподвешивают, то после этого размер его компоненты увеличится хотя бы в 2 раза. Значит, каждый элемент находился в переподвешиваемой компоненте не более $\log n$ раз, а так как переподвешивание происходит за O от размера компоненты, то суммарно все переподвешивания будут работать за $O(n \log n)$. Точнее, за $O(n \log n * \alpha(n))$, так как переход между вершинами графа G происходит за $O(\alpha(n))$.

Объединение компонент

Сначала каждую из вершин a и b, если в ее компоненте связности больше одной вершины, делаем точкой сочленения. После этого переподвешиваем меньшую из компонент, делая

соответственную вершину корнем. Затем подвешиваем ее к другой компоненте связности и объединяем их в соответствующей СНМ (`common`).

Реализация

Ниже приведена реализация алгоритма за $O((m+n)\log n * \alpha(n))$

```
const int N = 1e6;

int tm = 0, used[N], cnt[N];
bool joint[N];

//здесь реализация СНМ

void delete_joint(int v) {
        int u = strong.find_set(strong.par_comp[v]);
        strong.union_sets(v, u);
        joint[v] = 0;
        strong.par_comp[strong.find_set(v)] = strong.par_comp[v] == -1 ? -1
            : strong.par_comp[u];
}

void make_joint(int v) {
        if (joint[strong.ind[v]])return;
        int new_j = strong.make_set();
        int p = strong.find_set(strong.ind[v]);
        strong.ind[v] = new_j;
        strong.par_comp[new_j] = p;
        joint[new_j] = 1;
}

int lca(int a, int b) {
        ++tm;
        while (1) {
                if (a != -1) {
                        a = strong.find_set(a);
                        if (used[a] == tm)return a;
                        used[a] = tm;
                        a = strong.par_comp[a];
                }
                if (b != -1) {
                        b = strong.find_set(b);
                        if (used[b] == tm)return b;
                        used[b] = tm;
                        b = strong.par_comp[b];
                }
        }
```

```
}

void merge(int a, int b) {
        a = strong.find_set(a);
        b = strong.find_set(b);
        vector<int>vj, vc;
        int l = lca(a, b);
        int up_j = joint[l] ? l : strong.par_comp[l];
        while (a != l) {
                a = strong.find_set(a);
                if (a == l)break;
                if (joint[a])vj.push_back(a);
                else vc.push_back(a);
                a = strong.par_comp[a];
        }
        while (b != l) {
                b = strong.find_set(b);
                if (b == l)break;
                if (joint[b])vj.push_back(b);
                else vc.push_back(b);
                b = strong.par_comp[b];
        }
        if (joint[l])vj.push_back(l);
        else vc.push_back(l);
        if (vj.size() + vc.size() <= 2)return;
        int new_set = strong.make_set();
        for (int i : vc) {
                int p = strong.find_set(strong.par_comp[i]);
                if (joint[p]) cnt[p]--;
                if(p!=up_j)strong.par_comp[i] = new_set;
                strong.union_sets(new_set, i);
        }
        strong.par_comp[strong.find_set(new_set)] = up_j;
        cnt[up_j]++;
        for (int i : vj) {
                int p = strong.find_set(strong.par_comp[i]);
                if (i != l) {
                        if (joint[p]) cnt[p]--;
                        strong.par_comp[i] = new_set;
                }
        }
        for (int i : vj) {
                if (cnt[i] <= 1) {
                        if (cnt[i]<=0 && i != l) delete_joint(i);
                        else if (strong.par_comp[i] == -1) {
                                joint[i] = 0;
                                strong.union_sets(i, new_set);
                                strong.par_comp[strong.find_set(new_set)] = -1;
```

```
                        }
                }
        }
}

void make_root(int v) {
        v = strong.find_set(v);
        int child = -1;
        while (v != -1) {
                int p = strong.find_set(strong.par_comp[v]);
                strong.par_comp[v] = child;
                child = v;
                v = p;
        }
        cnt[v]++;
}

void add_edge(int a, int b) {
        int sa = strong.ind[a], sb = strong.ind[b];
        if (strong.is_connected(sa, sb)) return;
        if (!common.is_connected(a, b)) {
                int ca = common.find_set(a),
                        cb = common.find_set(b);
                if (common.size[ca] > common.size[cb]) swap(ca, cb), swap(a, b);
                if (common.size[ca] > 1)make_joint(a);
                if (common.size[cb] > 1)make_joint(b);
                sa = strong.ind[a], sb = strong.ind[b];
                cnt[sb]++;
                make_root(sa);
                sa = strong.find_set(sa);
                strong.par_comp[sa] = sb;
                common.union_sets(ca, cb);
        }
        else merge(sa, sb);
}
```

Стоит заметить, что большинство массивов испольуются только в одной CHM, так что их можно вынести в глобальные переменные. Так что они присутствуют в каждой структуре скорее для того, чтобы было понятно, с какой именно CHM мы работаем в данный момент. Также не стоит забывать, что в основной функции нужно проинициализировать изначальные n точек.

```
for(int i = 0; i < n; i++) {
                common.make_set();
                strong.make_set();
                common.ind[i] = i;//не стоит переносить это в make_set,
                strong.ind[i] = i;//так как ind корректен только для i=0..n-1
        }
```