Задача о динамической связности оффлайн

Задача:

Имеется неориентированный граф из n вершин, изначально не содержащий рёбер. Требуется обработать m запросов трёх типов:

- *добавить ребро между вершинами u и v,*
- *удалить ребро между вершинами u и v,*
- *проверить, лежат ли вершины u и v в одной компоненте связности.*

В графе могут быть кратные рёбра и петли.

В этой статье приведено решение задачи в offline, то есть ответы на все запросы будут получены после обработки всех запросов, а не по мере их поступления.

Содержание

Решение упрощённой задачи

Если нет удалений рёбер, задачу можно решить при помощи системы непересекающихся множеств. Каждая компонента связности — одно множество в СНМ, и при добавлении рёбер они объединяются.

Время работы такого решения: $O(m \cdot \alpha(n))$, где α — обратная функция Аккермана.

Алгоритм

Построение дерева отрезков

Рассмотрим массив запросов. Каждое ребро в графе существует на некотором отрезке запросов: начиная с запроса добавления и заканчивая запросом удаления (либо концом запросов, если ребро не было удалено). Для каждого ребра можно найти этот отрезок, пройдя по массиву запросов и запоминая, когда какое ребро было добавлено.

Пусть есть k рёбер, i-е соединяет вершины v_i и u_i, было добавлено запросом L_i и удалено запросом R_i.

Построим на массиве запросов дерево отрезков, в каждой его вершине будем хранить список пар. i-е рёбро графа нужно добавить на отрезок $[L_i, R_i]$. Это делается аналогично тому, как в дереве отрезков происходит добавление на отрезке (процесс описан в статье "Несогласованные поддеревья. Реализация массового обновления"), но без $push$: нужно спуститься по дереву от корня и записать пару u_i, v_i в вершины дерева отрезков.

Теперь чтобы узнать, какие рёбра существуют во время выполнения i-го запроса, достаточно посмотреть на путь от корня дерева отрезков до листа, который соответствует этому запросу — рёбра, записанные в вершинах этого пути, существуют во время выполнения запроса.

Ответы на запросы

Обойдём дерево отрезков в глубину, начиная с корня. Будем поддерживать граф, состоящий из рёбер, которые содержатся на пути от текущей вершины дерева отрезков до корня. При входе в вершину добавим в граф рёбра, записанные в этой вершине. При выходе из вершины нужно откатить граф к состоянию, которое было при входе. Когда мы добираемся до листа, в граф уже добавлены все рёбра, которые существуют во время выполнения соответствующего запроса, и только они. Поэтому если этот лист соответствует запросу третьего типа, его следует выполнить и сохранить ответ.

Для поддержания такого графа и ответа на запросы будем использовать систему непересекающихся множеств. При добавлении рёбер в граф объединим соответствующие множества в СНМ. Откатывание состояния СНМ описано ниже.

СНМ с откатами

Для того, чтобы иметь возможность откатывать состояние СНМ, нужно при каждом изменении любого значения в СНМ записывать в специальный массив, что именно изменилось и какое было предыдущее значение. Это можно реализовать как массив пар (указатель, значение).

Чтобы откатить состояние СНМ, пройдём по этому массиву в обратном порядке и присвоим старые значения обратно. Для лучшего понимания ознакомьтесь с приведённой ниже реализацией.

Нужно заметить, что эвристику сжатия путей в этом случае применять не следует. Эта эвристика улучшает асимптотическое время работы, но это время работы не истинное, а амортизированное. Из-за наличия откатов к предыдущим состояниям эта эвристика не даст выигрыша. СНМ с ранговой эвристикой же работает за $O(\log n)$ на запрос истинно.

Запоминание изменений и откаты не влияют на время работы, если оно истинное, а не амортизированное. Действительно: пусть в СНМ произошло r изменений. Каждое из них будет один раз занесено в массив и один раз отменено. Значит, запись в массив и откаты работают за $\Theta(r)$. Но и сами изменения заняли $\Theta(r)$ времени, значит, откаты не увеличили асимптотическое время работы.

Вместо описанного способа откатывания состояния СНМ можно использовать персистентный СНМ, но этот вариант сложнее и имеет меньшую эффективность.

Время работы

Каждое из $O(m)$ рёбер записывается в $O(\log m)$ вершин дерева отрезков. Поэтому операций union в СНМ будет $O(m \log m)$. Каждая выполняется за $O(\log n)$ (СНМ с ранговой эвристикой). Откаты не влияют на время работы.

Можно считать, что $n = O(m)$, так как в запросах используется не более $2m$ вершин.

Время работы: $O(m \log m \log n) = O(m \log^2 m)$.

Реализация на C++

```
#include <bits/stdc++.h>

using namespace std;
typedef pair < int , int > ipair;
const int N = 100321;

// СНМ
int dsuP[N], dsuR[N];
// В этот массив записываются все изменения СНМ, чтобы их можно откатить
// При изменении какого-то значения в СНМ в hist записывается пара < указатель, старое значение >
vector < pair < int*, int > > hist;

// Для элемента из СНМ возвращает корень дерева, в котором он находится
int dsuRoot(int v)
{
    while (dsuP[v] != -1)
        v = dsuP[v];
    return v;
}

// Объединяет два множества. Используется ранговая эвристика.
// При любом изменении содержимого массивов dsuP и dsuR
// в hist записывается адрес и старое значение
void dsuMerge(int a, int b)
{
    a = dsuRoot(a);
    b = dsuRoot(b);
    if (a == b)
        return;
    if (dsuR[a] > dsuR[b])
    {
        hist.emplace_back(&dsuP[b], dsuP[b]);
        dsuP[b] = a;
    } else if (dsuR[a] < dsuR[b])
    {
        hist.emplace_back(&dsuP[a], dsuP[a]);
        dsuP[a] = b;
    } else
    {
        hist.emplace_back(&dsuP[a], dsuP[a]);
        hist.emplace_back(&dsuR[b], dsuR[b]);
        dsuP[a] = b;
        ++dsuR[b];
    }
}

struct Query
{
    int t, u, v;
    bool answer;
};
int n, m;
Query q[N];

// Дерево отрезков, в каждой вершине которого хранится список рёбер
vector < ipair > t[N*4];

// Эта функция добавляет ребро на отрезок
// [l r] - отрезок, на который добавляется ребро
// uv - ребро, c - текущая вершина дерева отрезков,
// [cl cr] - отрезок текущей вершины дерева отрезков
void addEdge(int l, int r, ipair uv, int c, int cl, int cr)
{
    if (l > cr || r < cl)
        return;
    if (l <= cl && cr <= r)
    {
        t[c].push_back(uv);
        return;
    }
    int mid = (cl + cr) / 2;
    addEdge(l, r, uv, c*2+1, cl, mid);
```

```
    addEdge(l, r, uv, c*2+2, mid+1, cr);
}

// Обход дерева отрезков в глубину
void go(int c, int cl, int cr)
{
    int startSize = hist.size();
    // Добавляем рёбра при входе в вершину
    for (ipair uv : t[c])
        dsuMerge(uv.first, uv.second);

    if (cl == cr)
    {
        // Если эта вершина - лист, то отвечаем на запрос
        if (q[cl].t == 3)
            q[cl].answer = (dsuRoot(q[cl].u) == dsuRoot(q[cl].v));
    } else {
        int mid = (cl + cr) / 2;
        go(c*2+1, cl, mid);
        go(c*2+2, mid+1, cr);
    }

    // Откатываем изменения СНМ
    while ((int)hist.size() > startSize)
    {
        *hist.back().first = hist.back().second;
        hist.pop_back();
    }
}

int main()
{
    ios::sync_with_stdio(false);
    // Формат входных данных:
    // n и m, затем в m строках запросы: по три числа t, u, v
    // t - тип (1 - добавить ребро, 2 - удалить, 3 - принадлежат ли одной компоненте)
    // Нумерация вершин с нуля
    cin >> n >> m;
    for (int i = 0; i < n; ++i) // Инициализация СНМ
        dsuP[i] = -1;

    // В этом массиве для каждого ещё не удалённого ребра хранится
    // на каком запросе оно было создано
    set < pair < ipair, int > > edges;
    for (int i = 0; i < m; ++i)
    {
        cin >> q[i].t >> q[i].u >> q[i].v;
        // Поскольку рёбра неориентированные, u v должно означать то же самое, что и v u
        if (q[i].u > q[i].v) swap(q[i].u, q[i].v);
        // При добавлении ребра кладём его в set
        if (q[i].t == 1)
            edges.emplace(ipair(q[i].u, q[i].v), i);
        // При удалении ребра берём из set время его добавления - так мы узнаём отрезок заросов,
        // на котором оно существует. Если есть несколько одинаковых рёбер, можно брать любое.
        else if (q[i].t == 2)
        {
            auto iter = edges.lower_bound(make_pair(ipair(q[i].u, q[i].v), 0));
            addEdge(iter->second, i, iter->first, 0, 0, m - 1);
            edges.erase(iter);
        }
    }
    // Обрабатываем рёбра, которые не были удалены
    for (auto e : edges)
        addEdge(e.second, m - 1, e.first, 0, 0, m - 1);

    // Запускаем dfs по дереву отрезков
    go(0, 0, m - 1);
    // Выводим ответ.
    // При обходе дерева отрезков запросы обрабатываются в том же порядке, в котором они даны,
    // поэтому ответ можно выводить прямо в go без заполнения answer
    for (int i = 0; i < m; ++i)
        if (q[i].t == 3)
        {
            if (q[i].answer)
                cout << "YES\n";
            else
                cout << "NO\n";
        }
```

```
    return 0;
}
```

Замечания

- Дерево отрезков можно строить не на всех запросах, а только на запросах третьего типа. Это даст выигрыш по скорости и памяти, особенно если таких запросов немного по сравнению с общим числом запросов.
- Помимо проверки, лежат ли две вершины в одной компоненте связности, можно получать и другую информацию, которую можно получить из СНМ, напрмер:
  - Размер компоненты связности, которая содержит вершину v
  - Количество компонент связности
- Эту идею можно использовать и для других задач. Вместо СНМ можно использовать любую структуру данных, в которую можно добавлять, но не удалять.
  - Например, динамический рюкзак: добавлять предмет в него можно за $O(w)$ (w — максимальный вес), а удалять нельзя. Аналогично тому, как в dynamic connectivity offline добавляются и удаляются рёбра, можно удалять элементы из рюкзака.

См. также

- Система непересекающихся множеств
- Дерево отрезков

Источник — «http://neerc.ifmo.ru/wiki/index.php?title=Задача_о_динамической_связности_оффлайн&oldid=72633»

- Эта страница последний раз была отредактирована 12 февраля 2020 в 11:05.