добавлено: 10 Jun 2008 19:28
редактировано: 23 Aug 2011 11:23

Поиск мостов

Содержание [скрыть]

Пусть дан неориентированный граф. Мостом называется такое ребро, удаление которого делает граф несвязным (или, точнее, увеличивает число компонент связности). Требуется найти все мосты в заданном графе.

Неформально эта задача ставится следующим образом: требуется найти на заданной карте дорог все "важные" дороги, т.е. такие дороги, что удаление любой из них приведёт к исчезновению пути между какой-то парой городов.

Ниже мы опишем алгоритм, основанный на поиске в глубину, и работающий за время $O(n+m)$, где n — количество вершин, m — рёбер в графе.

Заметим, что на сайте также описан онлайновый алгоритм поиска мостов — в отличие от описанного здесь алгоритма, онлайновый алгоритм умеет поддерживать все мосты графа в изменяющемся графе (имеются в виду добавления новых рёбер).

Алгоритм

Запустим обход в глубину из произвольной вершины графа; обозначим её через root. Заметим следующий **факт** (который несложно доказать):

- Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину v или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является. (В самом деле, мы этим условием проверяем, нет ли другого пути из v в to, кроме как спуск по ребру (v, to) дерева обхода в глубину.)

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся "временами входа в вершину", вычисляемыми алгоритмом поиска в глубину.

Итак, пусть $tin[v]$ — это время захода поиска в глубину в вершину v. Теперь введём массив $fup[v]$, который и позволит нам отвечать на вышеописанные запросы. Время $fup[v]$ равно минимуму из времени захода в саму вершину $tin[v]$, времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра (v, p), а также из всех значений $fup[to]$ для каждой вершины to, являющейся непосредственным сыном v в дереве поиска:

$$fup[v] = \min \begin{cases} tin[v], \\ tin[p], & \text{for all (v,p) — back edge} \\ fup[to], & \text{for all (v,to) — tree edge} \end{cases}$$

(здесь "back edge" — обратное ребро, "tree edge" — ребро дерева)

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to, что $fup[to] \le tin[v]$. (Если $fup[to] = tin[v]$, то это означает, что найдётся обратное ребро, приходящее точно в v; если же $fup[to] < tin[v]$, то это означает наличие обратного ребра в какого-либо предка вершины v.)

Таким образом, если для текущего ребра (v, to) (принадлежащего дереву поиска) выполняется $fup[to] > tin[v]$, то это ребро является мостом; в противном случае оно мостом не является.

Реализация

Если говорить о самой реализации, то здесь нам нужно уметь различать три случая: когда мы идём по ребру дерева поиска в глубину, когда идём по обратному ребру, и когда пытаемся пойти по ребру дерева в обратную сторону. Это, соответственно, случаи:

- $used[to] = false$ — критерий ребра дерева поиска;
- $used[to] = true \ \&\& \ to \neq parent$ — критерий обратного ребра;
- $to = parent$ — критерий прохода по ребру дерева поиска в обратную сторону.

Таким образом, для реализации этих критериев нам надо передавать в функцию поиска в глубину вершину-предка текущей вершины.

```
const int MAXN = ...;
vector<int> g[MAXN];
bool used[MAXN];
int timer, tin[MAXN], fup[MAXN];

void dfs (int v, int p = -1) {
        used[v] = true;
        tin[v] = fup[v] = timer++;
        for (size_t i=0; i<g[v].size(); ++i) {
                int to = g[v][i];
                if (to == p)  continue;
                if (used[to])
                        fup[v] = min (fup[v], tin[to]);
                else {
                        dfs (to, v);
                        fup[v] = min (fup[v], fup[to]);
                        if (fup[to] > tin[v])
                                IS_BRIDGE(v,to);
                }
        }
}

void find_bridges() {
        timer = 0;
        for (int i=0; i<n; ++i)
                used[i] = false;
        for (int i=0; i<n; ++i)
                if (!used[i])
```

```
            dfs (i);
    }
```

Здесь основная функция для вызова — это find_bridges — она производит необходимую инициализацию и запуск обхода в глубину для каждой компоненты связности графа.

При этом $\text{IS\_BRIDGE}(a,b)$ — это некая функция, которая будет реагировать на то, что ребро (a,b) является мостом, например, выводить это ребро на экран.

Константе MAXN в самом начале кода следует задать значение, равное максимально возможному числу вершин во входном графе.

Стоит заметить, что эта реализация некорректно работает при наличии в графе **кратных рёбер**: она фактически не обращает внимания, кратное ли ребро или оно единственно. Разумеется, кратные рёбра не должны входить в ответ, поэтому при вызове IS_BRIDGE можно проверять дополнительно, не кратное ли ребро мы хотим добавить в ответ. Другой способ — более аккуратная работа с предками, т.е. передавать в dfs не вершину-предка, а номер ребра, по которому мы вошли в вершину (для этого надо будет дополнительно хранить номера всех рёбер).

Задачи в online judges

Список задач, в которых требуется искать мосты:

- UVA #796 **"Critical Links"** [сложность: низкая]
- UVA #610 **"Street Directions"** [сложность: средняя]