Лекция по алгоритмам #10
Тема: DFS

11 ноября

Собрано 1 января 2015 г. в 19:55

---

1 Компоненты рёберной двусвязности и мосты

1.1 Основные понятия

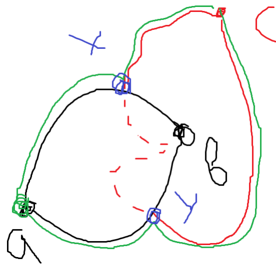
Рассматриваем связный неориентированный граф G.

Определение. Вершины v и u *рёберно двусвязны* ($v \sim u$) — существуют два простых не пересекающихся по рёбрам пути, соединяющих v и u.

Замечание. Эквивалентное определение: вершины v и u *рёберно двусвязны* ($v \sim u$), если лежат на одном рёберно-простом цикле.

Утверждение. *$\sim$ — отношение эквивалентности.*

Доказательство. Симметричность ($a \sim a$) и рефлексивность ($a \sim b \Longrightarrow b \sim a$) очевидны из определения. Необходимо доказать транзитивность: $a \sim b$, $b \sim c \Longrightarrow a \sim c$.



Рассмотрим два пути из c в b. Найдём их места первого пересечения с циклом $a \to b \to a$. Получили вершины x и y. Тогда есть рёберно не пересекающиеся пути $a \to x \to c$ и $a \to y \to c$.

□

Определение. *Компоненты рёберной двусвязности* — графы, индуцированные вершинами классов эквивалентности отношения $\sim$.

Определение. Граф G *рёберно двусвязен* — G состоит из одной компоненты рёберной двусвязности.

Определение. $e \in E$ называется *мостом* — $G \setminus e$ несвязен.

Утверждение. *G рёберно двусвязен $\Longleftrightarrow$ в G нет мостов.*

Доказательство. Если в G есть мост (vu), то между вершинами v и u только один простой путь, и G не рёберно двусвязен.

Если в G мостов нет, покажем по индукции, что G рёберно двусвязен. Покажем, что любые две вершины v и u графа рёберно двусвязны, индукция будет проводиться по расстоянию K между этими вершинами.

База. $K = 1$ — в G нет мостов, поэтому есть не единственный путь между соединёнными ребром вершинами v и u.

Переход. Рассмотрим вершины v, u на расстоянии $K+1$. $K+1 > 1$, поэтому можно рассмотреть вершину w на кратчайшем пути между v и u. Расстояния между вершинами v и w и между w и u меньше $K+1$, поэтому по предположению индукции $v \sim w$ и $w \sim u$. Тогда по транзитивности $v \sim u$. □

1.2 Алгоритм нахождения компонент рёберной двусвязности

Утверждение. *G — связный неориентированный граф, B — множество его мостов. Компоненты связности $G' = G \setminus B$ суть компоненты рёберной двусвязности G.*

Доказательство. Каждая из компонент связности G' не содержала мостов в G, следовательно, по доказанному выше утверждению, являлась двусвязным подграфом. При этом никакие две компоненты связности не могли лежать в одной компоненте двусвязности C, иначе C содержала бы мост, что противоречит доказанному утверждению. □

Пусть найдено множество всех мостов графа G за время T. Тогда построим граф $G' = G \setminus B$ и найдём его компоненты связности с помощью поиска в глубину. Итого, мы найдём компоненты рёберной двусвязности G за время $T + \mathcal{O}(V + E)$.

Если граф G несвязен, решаем задачу независимо для каждой из компонент связности.

1.3 Алгоритмы нахождения мостов

Граф G считаем связным.

Алгоритм 1. Переберём рёбра графа. Ребро e является мостом, если $G \setminus e$ несвязен — проверяем за $\mathcal{O}(V + E)$ с помощью поиска в глубину. Итого $\mathcal{O}(E(V + E))$.

Алгоритм 2. Построим остовное дерево графа с помощью поиска в глубину. Заметим, что любые рёбра, не лежащие в этом дереве, мостами не являются, так как при их удалении связность графа не теряется. Значит, достаточно перебирать (аналогично предыдущему алгоритму) только рёбра этого остовного дерева. Но их $V - 1$, поэтому итоговая сложность — $\mathcal{O}(V(V + E))$.

Алгоритм 3. Поиск всех мостов связного графа с помощью одного обхода в глубину за $\mathcal{O}(E)$.

`g[V]` — граф, хранящийся как список рёбер

`time_in[V]` — время входа в вершину (положительно, если вершина посещена)

`dfs(v)` возвращает минимальное достижимое время входа по путям, начинающимся в v, состоящим из прямых рёбер, кроме, возможно, последнего ребра (оно может быть обратным)

```
vector<int> g[V];
int time_in[V];
int timer = 0;
int dfs(int v, int parent=-1){
    time_in[v] = ++timer;
    int min_time = time_in[v];
    for (auto u : g[v])
        if (u != parent) {
            int t;
            if (time_in[u] == 0) { // not visited yet
                t = dfs(u, v);
                if (t > time_in[v])
                    is_bridge(v, u); // edge (v, u) is a bridge
            } else
                t = time_in[u];
            min_time = min(min_time, t);
        }
    return min_time;
}
...
dfs(0);
```

Если в графе могут быть кратные рёбра, необходимо запоминать не родителя вершины, а идентификатор ребра, по которому мы пришли в вершину, и пропускать в цикле только это ребро.

Алгоритм 4. Разбиение связного графа на компоненты рёберной двусвязности с помощью одного обхода в глубину за $\mathcal{O}(E)$.

```
vector<int> g[V];
vector<int> st; // stack
int time_in[V];
int timer = 0;
int dfs(int v, int parent=-1){
    time_in[v] = ++timer;
    int min_time = time_in[v];
    st.push_back(v);
    for (auto u : g[v])
        if (u != parent) {
            int t;
            if (time_in[u] == 0) { // not visited yet
                int cur_size = s.size();
                t = dfs(u, v);
                if (t > time_in[v]){
```

```
                        new_component(); // start new component
                    while (s.size() != cur_size){
                        add_to_component(s.back());
                        s.pop_back();
                    }
                    is_bridge(v, u); // edge (v, u) is a bridge
                }
            } else
                t = time_in[u];
            min_time = min(min_time, t);
        }
    return min_time;
}
...
dfs(0);
new_component(); // last component, includes vertex #0
while (!s.empty()){
    add_to_component(s.back());
    s.pop_back();
}
```

2 Компоненты вершинной двусвязности и точки сочленения

2.1 Основные понятия

Рассматриваем связный неориентированный граф G.

Определение. Рёбра e и f *вершинно двусвязны* ($e \sim f$) — e и f лежат на одном вершинно-простом цикле либо совпадают.

Утверждение. *$\sim$ — отношение эквивалентности.*

Доказательство. Симметричность ($e \sim e$) и рефлексивность ($e \sim f \implies f \sim e$) очевидны из определения. Необходимо доказать транзитивность: $e \sim f$, $f \sim g \Longrightarrow e \sim g$.

Найдём места первого пересечения цикла efe с циклом fgf. Получили вершины x и y. Тогда есть вершинно-простой цикл $exfye$. □

Определение. *Компоненты вершинной двусвязности (блоки)* — графы, индуцированные рёбрами классов эквивалентности отношения $\sim$.

Определение. *G вершинно двусвязен*, если состоит из одного блока.

Определение (1). *$v \in V$ — точка сочленения*, если $G \setminus v$ несвязен.

Определение (2). $v \in V$ — *точка сочленения*, если принадлежит не менее чем двум блокам.

Утверждение. *Определения 1 и 2 эквивалентны.*

Доказательство. Пусть v инцидентна каким-то двум рёбрам e и f, которые не лежат на одном цикле. Тогда при удалении v связность теряется.

Иначе v инцидентна только рёбрам одного блока, и тогда любые два ребра, инцидентные v, лежат на одном цикле. Тогда для каждой пары между их не-v концами есть простой путь. Значит, при удалении v связность не теряется. □

Теорема. *G вершинно двусвязен $\iff$ в G нет точек сочленения.*

Доказательство. Если есть точка сочленения — она лежит на двух блоках.

Иначе любые два ребра с общим концом лежат в одном блоке. Если рёбра e и f не имеют общего конца, то рассмотрим путь между этими рёбрами. В таком пути каждая пара соседних рёбер лежит в одном блоке, значит, по транзитивности отношения $\sim$, рёбра e и f лежат в одном блоке. □

2.2 Алгоритмы нахождения точек сочленения

Алгоритм 1. Проверим для каждой вершины v, является ли она точкой сочленения. Для этого запустим `DFS(v)`; если в дереве обхода у v есть $\geqslant 2$ сына — v является точкой сочленения, иначе нет. Итоговое время работы — $\mathcal{O}(VE)$.

Замечание. Этот алгоритм не получается модифицировать так, чтобы найти разбиение графа на блоки: удаление точек сочленения из графа не приводит к успеху.

Алгоритм 2. Запустим один обход в глубину, который для каждой вершины v посчитает $x(v)$ — количество детей в дереве обхода в глубину, из поддеревьев которых нет обратных рёбер выше v, и $y(v)$ — количество предков v (0 для корня и 1 для остальных вершин). Тогда v — точка сочленения $\iff x + y \geqslant 2$. (Это следует из того, что $x + y$ — количество компонент связности в графе $G \setminus v$.)

Получаем алгоритм поиск всех точек сочленения связного графа с помощью одного обхода в глубину за $\mathcal{O}(E)$.

`g[V]` — граф, хранящийся как список рёбер

`time_in[V]` — время входа в вершину (положительно, если вершина посещена)

`dfs(v)` возвращает минимальное достижимое время входа по путям, начинающимся в v, состоящим из прямых рёбер, кроме, возможно, последнего ребра (оно может быть обратным)

```
vector<int> g[V];
int time_in[V];
int timer = 0;
int dfs(int v, int parent=-1){
    time_in[v] = ++timer;
    int min_time = time_in[v], x = 0, y = (parent != -1);
    for (auto u : g[v])
```

```
        if (u != parent) {
            int t;
            if (time_in[u] == 0) { // not visited yet
                t = dfs(u, v);
                if (t >= time_in[v])
                    ++x;
            } else
                t = time_in[u];
            min_time = min(min_time, t);
        }
    if(x + y >= 2)
        is_cut(v); // v is cut vertex
    return min_time;
}
...
dfs(0);
```

Алгоритм 3. Разбиение связного графа на компноненты вершинной двусвязности с помощью одного обхода в глубину за $\mathcal{O}(E)$.

```
vector< pair<int, int> > g[V]; // pair = (to, edge's index)
vector<int> st; // stack
bool used[E];
int time_in[V];
int timer = 0;
int dfs(int v, int parent=-1){
    time_in[v] = ++timer;
    int min_time = time_in[v], x = 0, y = (parent != -1);
    for (auto p : g[v]){
        int u = p.first, id = p.second;
        if (u != parent) {
            int t, cur_size = st.size();
            if (!used[id]){
                    st.push_back(id);
                used[id] = 1;
            }
            if (time_in[u] == 0) { // not visited yet
                t = dfs(u, v);
                if (t >= time_in[v]){
                    ++x;
                    new_component();
                    while(st.size() != cur_size){
                        add_to_component(st.back());
                        st.pop_back();
```

```
                    }
                }
            } else
                t = time_in[u];
            min_time = min(min_time, t);
        }
    }
    if(x + y >= 2)
        is_cut(v); // v is cut vertex
    return min_time;
}
...
dfs(0);
```

3 Эйлеровы графы

3.1 Основные понятия

Рассматриваем связный неориентированный граф G.

Определение. *Эйлеров путь* — рёберно-простой путь, проходящий по всем рёбрам графа.

Определение. *Эйлеров цикл* — рёберно-простой цикл, проходящий по всем рёбрам графа.

Теорема. *Эйлеров цикл существует* $\Longleftrightarrow$ *степени всех вершин чётны.*

Доказательство. $\Longrightarrow$. Если в графе есть эйлеров цикл, то проходя по нему, мы в каждую вершину входим и выходим, используя чётное количество инцидентных ей рёбер. Все рёбра были использованы, значит, у каждой вершины степень четна.

$\Longleftarrow$. Рассмотрим самый длинный цикл C. Пусть он проходит не по всем рёбрам. Так как граф связен, то существует ребро, ведущее из вершины цикла $v \in C$. Пойдём по этому ребру, будем идти, пока можем идти по неиспользованным рёбрам. Заметим, что в графе $G \setminus C$ все вершины также имеют чётную степень, поэтому остановиться мы можем только в вершине v — назовём новый цикл D. Построим более длинный цикл: сначала обойдём цикл C, начиная из вершины v, затем обойдём цикл D. Получили цикл $C+D$, более длинный, чем C. Противоречие. Значит, цикл C был эйлеровым. $\square$

Теорема. *Эйлеров путь существует* $\Longleftrightarrow$ *степени не более чем двух вершин нечётны.*

Доказательство. $\Longrightarrow$. Если путь — не цикл, замкнём его дополнительным ребром. Тогда это — эйлеров цикл в новом графе, в нём степени всех вершин четны, значит, в исходном степени не более чем двух были нечетны.

$\Longleftarrow$. Если степени всех вершин четны, то эйлеровым путём будет эйлеров цикл.

Иначе есть ровно две вершины нечётной степени (т. к. их чётное количество в графе). Соединим их ребром e. Теперь в графе степени всех вершин четны, найдём эйлеров цикл. Удалим из него ребро e, получился эйлеров путь в исходном графе. $\square$

Замечание. Аналогичные понятия и критерии существования верны для ориентированного графа. Введём $\deg(v) = \deg_{out}(v) - \deg_{in}(v)$.

Эйлеров цикл в орграфе существует $\iff$ все вершины имеют $\deg(v) = 0$.

Эйлеров путь в орграфе существует $\iff$ все вершины имеют $\deg(v) = 0$, кроме, возможно, двух, которые имеют степени 1 и -1.

4 Алгоритмы поиска эйлеровых циклов и путей

Алгоритм 1. Поиск Эйлерова цикла в *ориентированном* графе за $\mathcal{O}(E)$.

```
vector< pair<int, int> > g[V]; // pair = (to, edge's index)
vector<int> answer; // indices of edges
void dfs(int v) {
    while (!g[v].empty()) {
        int x = g[v].back().first, i = g[v].back().second;
        g[v].pop_back();
        dfs(x);
        answer.push_back(i);
    }
}
...
dfs(0);
```

Алгоритм 2. Поиск Эйлерова цикла в *неориентированном* графе за $\mathcal{O}(E)$.

Необходимо удалять парное ребро: граф хранится как ориентированный, для каждого неориентированного ребра хранятся две ориентированные копии.

Будем гарантировать, что ребро k в неориентированном графе представляется в виде двух рёбер, имеющих индексы $2k$ и $2k + 1$.

```
bool deleted[E]; // NB: E is twice number of edges in undirected graph
vector< pair<int, int> > g[V]; // pair = (to, edge's index)
vector<int> answer; // indices of edges
void dfs(int v) {
    while (!g[v].empty()) {
        int x = g[v].back().first, i = g[v].back().second;
        g[v].pop_back();
        if (deleted[i])
                continue;

        deleted[i ^ 1] = 1; // 2k <-> 2k+1
        dfs(x);
        answer.push_back(i / 2); // 2k, 2k+1 -> k
    }
```

```
}
...
dfs(0);
```

Замечание. Для поиска эйлерова пути необходимо запускаться от вершины с «необычной» степенью: от вершины с нечётной степенью в неориентированном графе и от вершины со степенью $\deg(v) = \deg_{out}(v) - \deg_{in}(v) = 1$ в ориентированном случае.