Мосты и точки сочленения

Определение. *Мостом* называется ребро, при удалении которого связный неориентированный граф становится несвязным.

Определение. *Точкой сочленения* называется врешина, при удалении которой связный неориентированный граф становится несвязным.

Пример задачи, где их интересно искать: дана топология сети (компьютеры и физические соединения между ними) и требуется установить все единые точки отказа — узлы и связи, без которых будут существовать два узла, между которыми не будет пути.

Наивный алгоритм поочередного удаления каждого ребра (u, v) и проверки наличия пути $u \rightsquigarrow v$ потребует $O(m^2)$ операций. Чтобы научиться находить мосты быстрее, сначала сформулируем несколько утверждений, связанных с обходом в глубину.

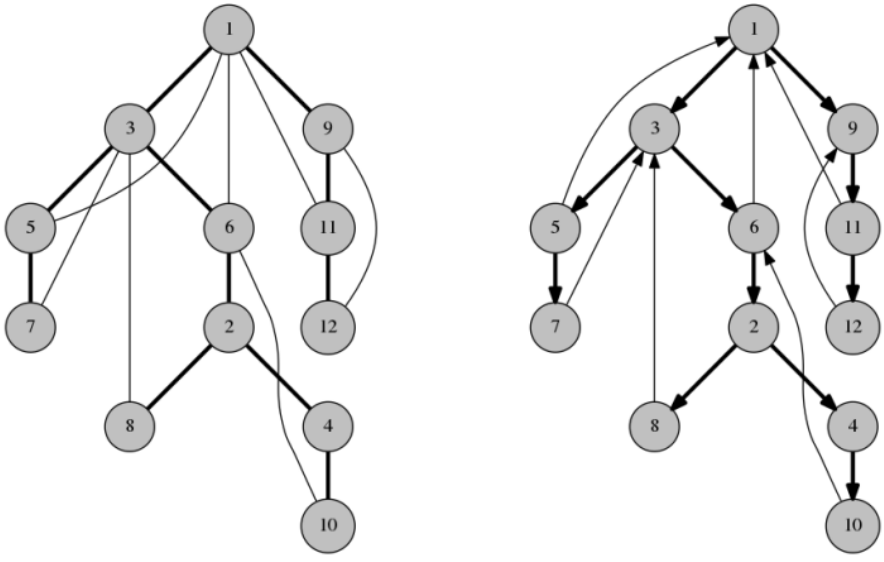
Запустим DFS из произвольной вершины. Введем новые виды рёбер:

- *Прямые* рёбра — те, по которым были переходы в dfs.
- *Обратные* рёбра — то, по которым не было переходов в dfs.

Заметим, что никакое обратное ребро (u, v) не может являться мостом: если его удалить, то всё равно будет существовать какой-то путь от u до v, потому что подграф из прямых рёбер является связным деревом.

Значит, остается только проверить все прямые рёбра. Это уже немного лучше — такой алгоритм будет работать за $O(nm)$.

Сооптимизировать его до линейного времени (до одного прохода dfs) поможет замечание о том, что обратные рёбра могут вести только «вверх» — к какому-то предку в дереве обхода графа, но не в другие «ветки» — иначе бы dfs увидел это ребро раньше, и оно было бы прямым, а не обратным.



Тогда, чтобы определить, является ли прямое ребро $v \to u$ мостом, мы можем воспользоваться следующим критерием: глубина h_v вершины v меньше, чем минимальная глубина всех вершин, соединенных обратным ребром с какой-либо вершиной из поддерева u.

Для ясности, обозначим эту величину как d_u, которую можно считать во время обхода по следующей формуле:

$$d_v = \min \begin{cases} h_v, & \\ d_u, & \text{ребро } (v \to u) \text{ прямое} \\ h_u, & \text{ребро } (v \to u) \text{ обратное} \end{cases}$$

Если это условие ($h_v < d_u$) не выполняется, то существует какой-то путь из u в какого-то предка v или саму v, не использующий ребро (v, u), а в противном случае — наоборот.

```
const int maxn = 1e5;

bool used[maxn];
int h[maxn], d[maxn];

void dfs(int v, int p = -1) {
    used[u] = true;
    d[v] = h[v] = (p == -1 ? 0 : h[p] + 1);
    for (int u : g[v]) {
        if (u != p) {
            if (used[u]) // если рябро обратное
                d[v] = min(d[v], h[u]);
            else { // если рябро прямое
                dfs(u, v);
                d[v] = min(d[v], d[u]);
                if (h[v] < d[v]) {
                    // ребро (v, u) -- мост
                }
            }
        }
    }
}
```

Примечание. Более известен алгоритм, вместо глубин вершин использующий их tin, но автор считает его чуть более сложным для понимания.

Точки сочленения

Задача поиска точек сочленения не сильно отличается от задачи поиска мостов.

Вершина v является точкой сочленения, когда из какого-то её ребёнка u нельзя дойти до её предка, не используя ребро (v, u). Для конкретного прямого ребра $v \to u$ этот факт можно проверить так: $h_v \le d_u$ (теперь нам нам достаточно нестрогого неравенства, так как если из вершины можно добраться до нее самой, то она все равно будет точкой сочленения).

Используя этот факт, можно оставить алгоритм практически прежним — нужно проверить этот критерий для всех прямых рёбер $v \to u$:

```
void dfs(int v, int p = -1) {
    used[u] = 1;
    d[v] = h[v] = (p == -1 ? 0 : h[p] + 1);
    int children = 0; // случай с корнем обработаем отдельно
    for (int u : g[u]) {
        if (u != p) {
            if (used[u])
                d[v] = min(d[v], h[u]);
            else {
                dfs(u, v);
                d[v] = min(d[v], d[u]);
                if (dp[v] >= tin[u] && p != -1) {
                    // u -- точка сочленения
                }
                children++;
            }
        }
    }
    if (p == -1 && children > 1) {
        // v -- корень и точка сочленения
    }
}
```

Единственный крайний случай — это корень, так как в него мы по определению войдём раньше других вершин. Но фикс здесь очень простой — достаточно посмотреть, было ли у него более одной ветви в обходе (если корень удалить, то эти поддеревья станут несвязными между собой).

Построение компонент вершинной двусвязности

Содержание

Двупроходный алгоритм

Найти компоненты вершинной двусвязности неориентированного графа можно с помощью обхода в глубину.

Первый проход: ищем точки сочленения с помощью обхода в глубину, заполняем массивы tin и up.

Второй проход: точка сочленения принадлежит как минимум двум компонентам вершинной двусвязности. Вершина $v \neq root$ является точкой сочленения, если у нее есть сын u, такой что $up[u] \geqslant tin[v]$.
Это также значит, что ребро vu содержится в другой компоненте вершинной двусвязности, нежели ребро по которому мы пришли в вершину v , используя поиск в глубину. Получается, что перейдя по этому ребру, мы окажемся в другой компоненте вершинной двусвязности.
Используем это свойство, чтобы окрасить компоненты вершинной двусвязности в различные цвета.

Псевдокод второго прохода

- Во время первого запуска dfs будут заполняться массивы tin и up, поэтому при запуске функции $paint$ мы считаем, что они уже посчитаны.
- `maxColor` изначально равен 0, что эквивалентно тому, что никакое ребро не окрашено.
- `color` хранит в себе цвет, компоненты, из которой вызвалась функция paint для текущей вершины.
- `parent` — это вершина, из которой мы попали в текущую.

```
function paint(v, color, parent):
  visited[v] = true
  for (v, u) ∈ E:
    if u == parent
      continue
    if not visited[u]
      if up[u] ⩾ tin[v]
        newColor = ++maxColor
        col[vu] = newColor
        paint(u, newColor, v)
      else
        col[vu] = color
        paint(u, color, v)
    else if tin[u] < tin[v]
      col[vu] = color
```

```
function solve():
  for v ∈ V:
    dfs(v)
  for v ∈ V:
    if not visited[v]
      maxColor++
      paint(v, maxColor, -1)
```

Ребра каждой из компонент вершинной двусвязности окажутся окрашенными в свой цвет.
В алгоритме выполняется два прохода dfs, каждый из которых работает $O(|V| + |E|)$. Значит время работы алгоритма $O(|V| + |E|)$.

Однопроходный алгоритм

Заведем стек, в который будем записывать все дуги в порядке их обработки. Если обнаружена точка сочленения, дуги очередного блока окажутся в этом стеке, начиная с дуги дерева обхода, которая привела в этот блок, до верхушки стека.
Таким образом, каждый раз находя компоненту вершинной двусвязности мы сможем покрасить все ребра, содержащиеся в ней, в новый цвет.

Доказательство корректности алгоритма

Предположим, что граф содержит точку сочленения $i' \in V$, за которой следует один или несколько блоков. Вершины из этих блоков образуют подмножество $V' \subset V$. В таком случае:

1. Все вершины V' являются потомками i' в дереве обхода;
2. Все вершины V' будут пройдены в течение периода серого состояния i';
3. В G не может быть обратных дуг из V' в $V \setminus V'$.

Значит все дуги V' будут будут добавлены в стек после дуги ведущей из точки сочленения в блок. В стеке в момент обнаружения точки сочленения будут находиться только дуги блока, связанного с ней, т.к. блоки найденные до него (если таковые имеются) будет уже извлечены из стека и покрашены в свой цвет.

Псевдокод

```
function paint(v, parent):
  visited[v] = true
  tin[v] = up[v] = time++
  for (v, u) ∈ E:
    if u == parent
      continue
    if not visited[u]
      stack.push(vu)
      paint(u, v)
      if up[u] ⩾ tin[v]
        color = maxColor++
        while stack.top() != (vu)
          colors[stack.top()] = color
          stack.pop()
        colors[vu] = color
        stack.pop()
      if up[u] < up[v]
        up[v] = up[u]
    else if tin[u] < tin[v]
      stack.push(vu)
      if tin[u] < up[v]
        up[v] = tin[u]
    else if up[v] > tin[u]
      up[v] = up[u]
```

```
function solve():
  for v ∈ V:
    dfs(v)
  for v ∈ V:
    if not visited[v]:
      time = 0
      maxColor++
      paint(v, -1)
```

Во время алгоритма совершается один проход dfs, который работает за $O(|V| + |E|)$. Внутри него совершается еще цикл, который суммарно выполняет $O(|E|)$ операций, т.к. каждое ребро может быть добавлено в стек только один раз. Следовательно, общее время работы алгоритма $O(|V| + |E|) + O(|E|) = O(|V| + |E|)$

См. также

- Использование обхода в глубину для поиска точек сочленения
- Построение компонент реберной двусвязности

Источники информации

- В.А.Кузнецов, А.М.Караваев. "Оптимизация на графах" - Петрозаводск, Издательство ПетрГУ 2007
- Дискретная математика: Алгоритмы — Компоненты двусвязности, мосты и точки сочленения (http://rain.ifmo.ru/cat/view.php/vis/graph-general/biconnected-components-2005)

Построение компонент рёберной двусвязности

Построение компонент реберной двусвязности будет осуществляться с помощью обхода в глубину.

Содержание

Двупроходный алгоритм

Первый способ найти искомые компоненты — сначала определить критерий перехода в новую компоненту реберной двусвязности, а затем покрасить вершины графа в нужные цвета.

Определим критерий перехода к новой компоненте. Воспользуемся ранее доказанной леммой. Получается — перешли по мосту, следовательно началась новая компонента.

Первый проход: запустим алгоритм для поиска мостов, чтобы посчитать две величины: $tin(v)$ и $up(v)$.

Второй проход: окрашиваем вершины, т.е. если перешли по мосту, то оказались в новой компоненте реберной двусвязности.

Псевдокод второго прохода

- В переменной `color` хранится цвет текущей компоненты.
- `maxColor` изначально равен 0, что эквивалентно тому, что никакая компонента не окрашена.

```
function paint(v, color):
  colors[v] = color
  for (u,v) ∈ E:
    if colors[u] == 0:
      if up[u] > tin[v]:
        maxColor++
        paint(u, maxColor)
      else:
        paint(u, color)
```

```
function solve():
  for v ∈ V :
    colors[v] = 0
    if not visited[v]
      dfs(v)
  maxColor = 0
  for v ∈ V :
    if colors[v] == 0:
      maxColor++
      paint(v, maxColor)
```

Вершины каждой из компонент реберной двусвязности окажутся окрашенными в свой цвет.

Время работы алгоритма будет время работы двух запусков dfs, то есть $2 \cdot O(|V| + |E|)$, что есть $O(|V| + |E|)$.

Однопроходный алгоритм

Однопроходный алгоритм строится на базе алгоритма поиска мостов. Во-первых, создадим глобальный стек, и при спуске по дереву dfs добавляем в него вершины. Во-вторых, когда возвращаемся назад, проверяем не является ли ребро мостом (при помощи леммы). Если это так, то все вершины, находящиеся до текущего потомка в стеке, принадлежат одной компоненте.Заметим, что эта компонента будет висячей вершиной в дереве блоков и мостов, так как обходили граф поиском в глубину. Значит, ее можно выкинуть и продолжить поиск в оставшемся графе. Действуя по аналогии в получившемся графе, найдем оставшиеся компоненты реберной двусвязности.

Псевдокод

```
function paint(v):
  maxColor++
  last = -1
  while last != v and not stack.empty()
    colors[stack.top()] = maxColor
    last = stack.top()
    stack.pop()
```

```
function dfs(v)
  time =  time + 1
  stack.push(v)
  tin[v] = time
  up[v] = time
  for (v, u) ∈ E:
    if (v, u) – обратное ребро
      up[v] = min(up[v], tin[u])
    if not visited[u]
      dfs(u)
      up[v] = min(up[v], up[u])
      if up[u] > tin[v]
        paint(u)
```

Так же после вызова dfs нужно не забыть в конце вызвать ещё раз paint.

Теперь две вершины имеют одинаковый цвет тогда и только тогда, когда они принадлежат одной компоненте реберной двусвязности.

Время работы dfs $O(|V| + |E|)$. Покраска за $O(|V|)$. Итоговое время работы алгоритма $O(|V| + |E|)$.

См. также

- Построение компонент вершинной двусвязности
- Использование обхода в глубину для поиска мостов

Источники информации

- *Седжвик Р.* Фундаментальные алгоритмы на C++. Часть 5: Алгоритмы на графах. Пер. с англ. — СПб.: ООО «ДиаСофтЮП», 2002. — С. 123-128

- *Кузнецов В.А., Караваев. А.М.* "Оптимизация на графах" - Петрозаводск, Издательство ПетрГУ 2007
- Визуализация — Построение компонент реберной двусзяности (http://rain.ifmo.ru/cat/view.php/vis/graph-general/bridges-2001%7C)

Все задачи контеста учебные. Нужно написать соответствущий алгоритм.

В задаче E нужно понимать, что в графе не может быть трех вершин с нечетной степенью(количество вершин с нечетной степенью четно) и что в задаче не нужно минимизировать пройденный путь, так как нужно посетить все улицы(то есть все ребра).