Графы: мосты, точки сочленения, компоненты сильной связности

Сапожников Денис

Contents

1 Мосты и точки сочленения

1.1 Мосты

Определение. Ребро в графе будет называться **мостом**, если при удалении его, граф распадётся на 2 компоненты связности.

Лемма. *Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. Тогда, если текущее ребро (v, u) таково, что из вершины u 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.*

Proof. В самом деле, мы этим условием проверяем, нет ли другого пути из v в u, кроме как спуск по ребру (v, u) 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_u для каждой вершины u, являющейся непосредственным сыном v в дереве поиска:

$$fup_v = \min \begin{cases} tin_v \\ tin_p, & (v,p) - \text{это обратное ребро} \\ fup_u, & (v,u) - \text{это прямое ребро} \end{cases}$$

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

Таким образом, если для текущего ребра (v, u) (принадлежащего дереву поиска) выполняется $fup_u > tin_v$, то это ребро является мостом; в противном случае оно мостом не является.

```
vector<int> g[N];
bool used[N];
int timer, tin[N], fup[N];

void dfs(int v, int p = -1) {
  used[v] = true;
  tin[v] = fup[v] = timer++;
  for (int u : gr[v]) {
    if (u == p)  continue;
    if (used[u])
      fup[v] = min(fup[v], tin[u]);
    else {
      dfs(u, v);
      fup[v] = min(fup[v], fup[u]);
      if (fup[u] > tin[v])
        cout << v << ' ' << u << '\n';
    }
  }
}
```

Стоит заметить, что эта реализация некорректно работает при наличии в графе кратных рёбер: она фактически не обращает внимания, кратное ли ребро или оно единственно. Поэтому для кратных рёбер я рекомендую смотреть не на предка, а на «id предыдущего» ребра. В таком случае код по сути не меняется.

1.2 Точки сочленения

Определение. Вершина в графе будет называться **точкой сочленения**, если при удалении её, граф распадётся на 2 компоненты связности.

На самом деле это почти то же самое, что и мосты. Оставаясь в той же терминологии, что и в мостах, вершина будет точкой сочленения, если $fup_u \geq tin_v$ для хоты бы одного сына u вершины v. Таким образом, код нужно поменять лишь в одном месте для всех вершин, кроме корня. Корень – это отдельный случай, он будет являться точкой сочленения, если из него мы запустимся хотя бы в 2 сына (не путать со степенью вершины!).

```
void dfs(int v, int p = -1) {
  used[v] = true;
  tin[v] = fup[v] = timer++;
  int children = 0;
  for (int u : gr[v]) {
    if (u == p)  continue;
    if (used[u])
    fup[v] = min(fup[v], tin[u]);
    else {
      dfs(u, v);
      fup[v] = min(fup[v], fup[u]);
      if (fup[u] >= tin[v] && p != -1)
        is_articulation_point[v] = true;
      ++children;
    }
  }
  if (p == -1 && children > 1)
    is_articulation_point[v] = true;
}
```

1.3 2-мосты*

k рёбер называются k-мостом, если при их удалении граф распадается на несколько компонент связности.

Можно ли посчитать количество 2-мостов за быстро? Можно, на самом деле их количество можно найти за $O(n+m)$, но это требует дополнительных рассуждений, а для нахождения за $O(m \log m)$ уже требуется немаленькое их количество.

Построим дерево дфса. Осознаем, что существует 3 типа пар удаляемых рёбер:

1. Оба ребра не из дерева, тогда они не два-мост.

2. Оба ребра из дерева.

   Тут опять же есть несколько случаев:

   (a) Два ребра расположены не вертикально (то есть не в поддереве другого). Тогда одно из них точно мост, потому что иначе это не dfs.

   (b) Два ребра расположены вертикально, верхнее зафиксируем, хотим посчитать количество подходящих нижних. Разобьём дерево на 3 части: верхняя, средняя (между двумя рёбрами) и нижняя. Если верхняя или нижняя - это новые компоненты, то одно из рёбер было мостом. Остаётся третий вариант, когда часть посередине - это новая компонента, и из нижней части есть обратное ребро в верхнюю. Тогда заметим, что мы можем удалить любое из рёбер до самого не глубокого обратного ребра, чтобы часть посередине оказалась новой компонентой связности. Таким образом нам нужно поддерживать самое не глубокое обратное ребро, это умеет делать сливаемая куча по паре $depth, v$, где у нас есть обратное ребро в вершину v из вершины на глубине $depth$. Ещё нам понадобится сливаемая куча по ревёрснутым парам, чтобы уметь нормально удалять обратные рёбра.

   У нас осталось посчитать только случай, когда одно из рёбер - мост, количество таких пар легко посчитать по формуле. Пусть мостов k, тогда 2-мостов такого типа будет $k(m-1) - k(k-1)$.

3. Одно ребро из дерева обхода, второе — нет, тогда одо ведёт из поддерева в наддерево. Зафиксируем ребро. Если из поддерева существует хотя бы 2 обратных ребра, то мы не сможем удалить его и ещё одно так, чтобы граф оказался не связным. Значит нам нужно, чтобы было ровно одно обратное ребро. Перенумеруем вершины в порядке обхода dfs. Будем поддерживать сливаемую кучу обратных рёбер. Достаточно хранить лишь вершину, в которую ведёт это ребро, тогда нам нужно будет сливать две кучи, удалять топ и проверять размер, всё это поддерживает эта структура.

Победа, мы смогли в решение за $O(m \log m)$.

На самом деле после такого возникает вопрос: можно ли посчитать количество 3-мостов за O(быстро) или хотя бы проверить их наличие. На самом деле ответ на второй вопрос – можно за $O(m \log m)$, а ответа на 1 вопрос я не знаю.

2 Компоненты сильной связности

Компонентой сильной связности (strongly connected component) называется такое (максимальное по включению) подмножество вершин C, что любые две вершины этого подмножества достижимы друг из друга, т.е. для $\forall u, v \in C$:

$$u \mapsto v, v \mapsto u$$

где символом $\mapsto$ здесь и далее мы будем обозначать достижимость, т.е. существование пути из первой вершины во вторую.

Понятно, что компоненты сильной связности для данного графа не пересекаются, т.е. фактически это разбиение всех вершин графа. Отсюда логично определение конденсации G^{SCC} как графа, получаемого из данного графа сжатием каждой компоненты сильной связности в одну вершину. Каждой вершине графа конденсации соответствует компонента сильной связности графа G, а ориентированное ребро между двумя вершинами C_i и C_j графа конденсации проводится, если найдётся пара вершин$u \in C_i, v \in C_j$, между которыми существовало ребро в исходном графе, т.е. $(u, v) \in E$.

Важнейшим свойством графа конденсации является то, что он ацикличен. Действительно, предположим, что $C \mapsto C'$, докажем, что $C' \not\mapsto C$. Из определения конденсации получаем, что найдутся две вершины $u \in C$ и $v \in C'$, что $u \mapsto v$. Доказывать будем от противного, т.е. предположим, что $C' \mapsto C$, тогда найдутся две вершины $u' \in C$ и $v' \in C'$, что $v' \mapsto u'$. Но т.к. u и u' находятся в одной компоненте сильной связности, то между ними есть путь; аналогично для v и v'. В итоге, объединяя пути, получаем, что $v \mapsto u$, и одновременно $u \mapsto v$. Следовательно, u и v должны принадлежать одной компоненте сильной связности, т.е. получили противоречие, что и требовалось доказать.

Алгоритм представляет из себя 2 шага: первый – это топологическая сортировка графа. Второй – обход обратного графа в порядке топологической сортировки. То есть по сути, алгоритм очень простой, но нужно доказать, что он действительно работает.

На первом шаге алгоритма выполняется серия обходов в глубину, посещающая весь граф. Для этого мы проходимся по всем вершинам графа и из каждой ещё не посещённой вершины вызываем обход в глубину. При этом для каждой вершины v запомним время выхода $\text{tout}[v]$. Эти времена выхода играют ключевую роль в алгоритме, и эта роль выражена в приведённой ниже теореме.

Сначала введём обозначение: время выхода $\text{tout}[C]$ из компоненты C сильной связности определим как максимум из значений $\text{tout}[v]$ для всех $v \in C$. Кроме того, в доказательстве теоремы будут упоминаться и времена входа в каждую вершину $\text{tin}[v]$, и аналогично определим времена входа $\text{tin}[C]$ для каждой компоненты сильной связности как минимум из величин $\text{tin}[v]$ для всех $v \in C$.

Теорема 1. *Пусть C и C' – две различные компоненты сильной связности, и пусть в графе конденсации между ними есть ребро (C, C'). Тогда* $\text{tout}[C] > \text{tout}[C']$.

Proof. При доказательстве возникает два принципиально различных случая в зависимости от того, в какую из компонент первой зайдёт обход в глубину, т.е. в зависимости от соотношения между $\text{tin}[C]$ и $\text{tin}[C']$:

1. Первой была достигнута компонента C. Это означает, что в какой-то момент времени обход в глубину заходит в некоторую вершину v компоненты C, при этом все остальные вершины компонент C и C' ещё не посещены. Но, т.к. по условию в графе конденсаций есть ребро (C, C'), то из вершины v будет достижима не только вся компонента C, но и вся компонента C'. Это означает, что при запуске из вершины v обход в глубину пройдёт по всем вершинам компонент C и C', а, значит, они станут потомками по отношению к

v в дереве обхода в глубину, т.е. для любой вершины $u \in C \cup C', u \neq v$ будет выполнено $\text{tout}[v] > \text{tout}[u]$, что и т.д.

2. Первой была достигнута компонента C'. Опять же, в какой-то момент времени обход в глубину заходит в некоторую вершину $v \in C'$, причём все остальные вершины компонент C и C' не посещены. Поскольку по условию в графе конденсаций существовало ребро (C, C'), то, вследствие ацикличности графа конденсаций, не существует обратного пути $C' \not\to C$, т.е. обход в глубину из вершины v не достигнет вершин C. Это означает, что они будут посещены обходом в глубину позже, откуда и следует $\text{tout}[C] > \text{tout}[C']$, что и т.д.

□

Доказанная теорема является основой алгоритма поиска компонент сильной связности. Из неё следует, что любое ребро (C, C') в графе конденсаций идёт из компоненты с большей величиной tout в компоненту с меньшей величиной.

Если мы отсортируем все вершины $v \in V$ в порядке убывания времени выхода $\text{tout}[v]$, то первой окажется некоторая вершина u, принадлежащая «корневой» компоненте сильной связности, т.е. в которую не входит ни одно ребро в графе конденсаций. Теперь нам хотелось бы запустить такой обход из этой вершины u, который бы посетил только эту компоненту сильной связности и не зашёл ни в какую другую; научившись это делать, мы сможем постепенно выделить все компоненты сильной связности: удалив из графа вершины первой выделенной компоненты, мы снова найдём среди оставшихся вершину с наибольшей величиной tout, снова запустим из неё этот обход, и т.д.

Чтобы научиться делать такой обход, рассмотрим транспонированный граф G^T, т.е. граф, полученный из G изменением направления каждого ребра на противоположное. Нетрудно понять, что в этом графе будут те же компоненты сильной связности, что и в исходном графе. Более того, граф конденсации $(G^T)^{\text{SCC}}$ для него будет равен транспонированному графу конденсации исходного графа G^{SCC}. Это означает, что теперь из рассматриваемой нами «корневой» компоненты уже не будут выходить рёбра в другие компоненты.

Таким образом, чтобы обойти всю «корневую» компоненту сильной связности, содержащую некоторую вершину v, достаточно запустить обход из вершины v в графе G^T. Этот обход посетит все вершины этой компоненты сильной связности и только их. Как уже говорилось, дальше мы можем мысленно удалить эти вершины из графа, находить очередную вершину с максимальным значением $\text{tout}[v]$ и запускать обход на транспонированном графе из неё, и т.д.

```
vector<vector<int>> gr, grrev;
vector<char> used;
vector<int> order, component;

void dfs1(int v) {
  used[v] = true;
  for (int u : gr[v])
    if (!used[u])
      dfs1(u);
  order.push_back(v);
}

void dfs2(int v) {
  used[v] = true;
  component.push_back(v);
  for (int u : grrev[v])
    if (!used[u])
```

```
      dfs2(u);
}

int main() {
  //read
  used.assign(n, false);
  for (int i = 0; i < n; ++i)
  if (!used[i])
  dfs1(i);
  used.assign(n, false);
  reverse(order.begin(), order.end());
  for (int v : order) {
    if (!used[v]) {
      dfs2(v);
      //print component
      component.clear();
    }
  }
}
```