Графы: Топологическая сортировка. Компоненты сильной связности. 2-SAT. Эйлеров путь

Сапожников Денис

Contents

1 Топологическая сортировка

Задача. Дан ориентированный ациклический граф и две вершины s и t. Нужно понять, сколько существует путей из s в t.

Такая задача легко решается подходом динамического программирования. Пусть dp_v – ответ для вершины v, а dp_u для всех u — детей v нам уже известно, тогда $dp_v = \sum_u dp_u$.

Все бы хорошо, вот только нам нужно уже знать все динамики для всех сыновей. Таким образом нам нужно перенумеровать вершины так, чтобы каждое ребро вело из вершины с меньшим номером в вершину с большим.

Такой порядок называется топологическим и, очевидно, что он может быть не единственным.

Вспомним, что к моменту выхода из вызова $dfs(v)$ все вершины, достижимые из v как непосредственно (по одному ребру), так и косвенно (по пути) — все такие вершины уже посещены обходом. Следовательно, если мы будем в момент выхода из $dfs(v)$ добавлять нашу вершину в начало некоего списка, то в конце концов в этом списке получится топологическая сортировка.

```
bool used[N];
vector<int> ans;

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

void topological_sort() {
  for (int i = 0; i < n; ++i)
    if (!used[i])
      dfs(i);
  reverse(ans.begin(), ans.end());
}
```

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();
    }
  }
}
```

3 2-SAT

Одним из применений компонент сильной связности является так называемая задача 2-SAT.

Пусть у нас есть булево выражение, представляющее собой 2-конъюнктивную нормальную форму (2-КНФ), а именно:

$$f(x_1, x_2, \ldots, x_n) = (x_{i_1}||!x_{j_1})\&\&(x_{i_2}||x_{j_2})\&\& \ldots \&\&(!x_{i_k}||x_{j_k})$$

То есть у нас есть скобки, внутри которых лежит логическое или от двух переменных (или их отрицаний).

Хочется проверить, а существует такой набор $(x_1, \ldots, x_n) : f(x_1, \ldots, x_n) = 1$ и если существует, то найти любой подходящий.

Для начала переведём выражение к импликациям. Заметим, что

$$a||b \Leftrightarrow (!a \Rightarrow b\&\&!b \Rightarrow a)$$

Теперь составим граф импликаций. Для каждой переменной заведём две её копии: x_i и $!x_i$. Теперь мы можем провести ориентированные рёбра из преобразованного выражения.

Мы получили какой-то граф, исследуем его свойства в терминах задачи.

Лемма 2. *Если x_i и $!x_i$ лежит в одной компоненте связности, то у КНФ не будет решения.*

Proof. Это легко понять, ведь если поставить $x_i = 0$, то получим, что $!x_i \Rightarrow x_i$, а это явно какая-то беда (из 1 не следует 0). Аналогично если $x_i = 1$. □

Лемма 3. *Если все переменные и их отрицания лежат в разных компонентах сильной связности, то найдётся решение КНФ.*

Proof. Для доказательства давайте просто предъявим алгоритм, который строит решение.

Пусть $comp[v]$ обозначает номер компоненты сильной связности, которой принадлежит вершина v, причём номера упорядочены в порядке топологической сортировки компонент сильной связности в графе компонентов (т.е. более ранним в порядке топологической сортировки соответствуют большие номера: если есть путь из v в w, то $comp[v] \leq comp[w]$). Тогда, если $comp[x] < comp[!x]$, то выбираем значение $!x$, иначе, т.е. если $comp[x] > comp[!x]$, то выбираем x.

Во-первых, докажем, что из x не достижимо $!x$. Действительно, так как номер компоненты сильной связности $comp[x]$ больше номера компоненты $comp[!x]$, то это означает, что компонента связности, содержащая x, расположена левее компоненты связности, содержащей $!x$, и из первой никак не может быть достижима последняя.

Во-вторых, докажем, что никакая вершина y, достижимая из x, не является «плохой», т.е. неверно, что из y достижимо $!y$. Докажем это от противного. Пусть из x достижимо y, а из y достижимо $!y$. Так как из x достижимо y, то, по свойству графа импликаций, из $!y$ будет достижимо $!x$. Но, по предположению, из y достижимо $!y$. Тогда мы получаем, что из x достижимо $!x$, что противоречит условию, что и требовалось доказать. □

Пусть вершина $2k$ отвечает за $x_k = 0$, а $2k + 1$ за $x_k = 1$. Построим граф импликаций и сконденсируем его, оставшаяся часть кода после конденсации будет очень простой:

```
int main() {
  //read and graph condansation

  for (int i = 0; i < n; i += 2)
    if (comp[i] == comp[i^1]) {
```

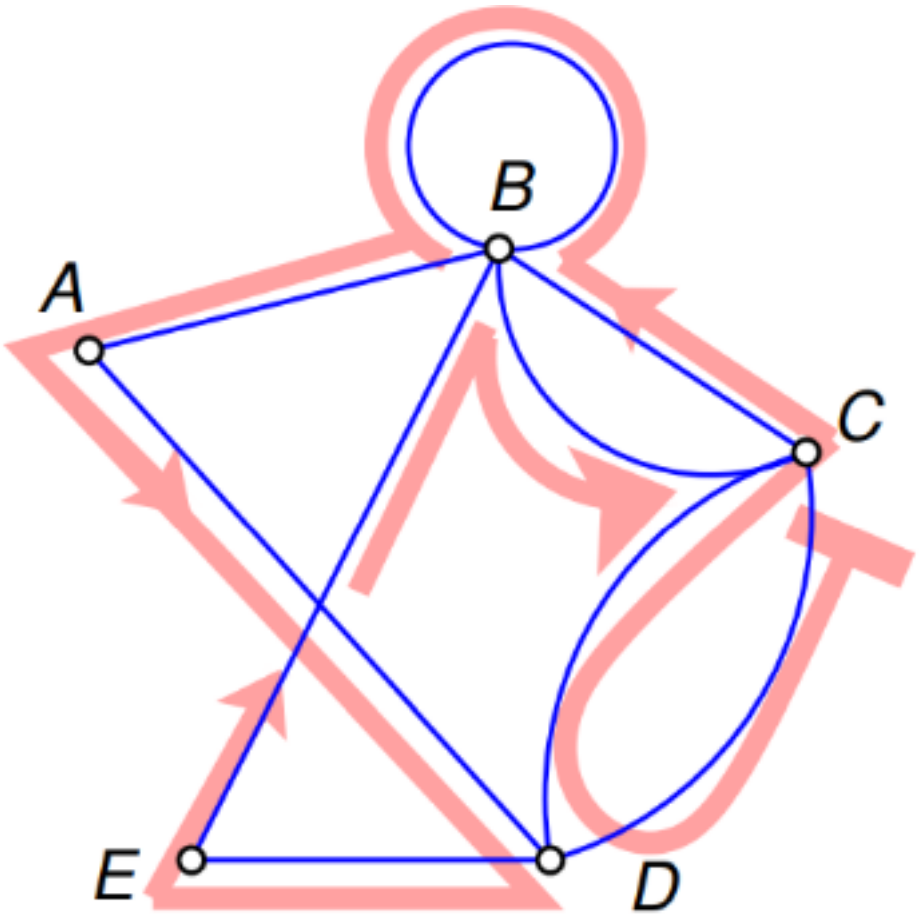
```
      cout << "NO SOLUTION";
      return 0;
    }
  for (int i = 0; i < n; i += 2)
    cout << "x[" << i << "] = " << (comp[i] < comp[i ^ 1]);
}
```

4 Эйлеровы цикл и путь

Определение. Эйлеров путь — это путь в графе, проходящий через все его рёбра.

Определение. Эйлеров цикл — это эйлеров путь, являющийся циклом.

Для простоты в обоих случаях будем считать, что граф неориентированный.



Граф на пяти вершинах и один из его эйлеровых циклов: CDCBBADEBC

Также существует понятие *гамильтонова* пути и цикла — они посещают все *вершины* по разу, а не рёбра. Нахождение гамильтонова цикла (*задача коммивояжера*, англ. *travelling salesman problem*) — одна из самых известных NP-полных задач, в то время как нахождение эйлерова цикла решается за линейное время, и мы сейчас покажем, как.

4.1 Нахождение эйлерова цикла

Теорема. *Эйлеров цикл существует тогда и только тогда, когда граф связный и степени всех вершин чётны.*

Proof. Необходимость показывается так: можно просто взять эйлеров цикл и ориентировать все его ребра в порядке обхода. Тогда из каждой вершины будет выходить столько же рёбер, сколько входить, а значит степень у всех вершин исходного неориентированного графа была четной.

Достаточность докажем конструктивно — предъявим алгоритм нахождения цикла.

```
void euler(int v) {
  while (!g[v].empty()) {
    u = *g[v].begin();
    remove_edge(v, u);
    euler(u);
  }
  cout << v << " ";
}
```

Заметим, что:

- выведется последовательность из ровно $(m + 1)$ вершин (потому что мы m раз удаляем по ребру)
- между каждой соседней парой выписанных вершин есть ребро (потому что мы пришли в текущую вершину по существующему ребру, а выпишутся они в обратном порядке)
- каждое ребро будет выписано ровно один раз (потому что после прохода по ребру мы его удаляем)

Значит, если условия на связность и четность степеней выполняются, то выведенная последовательно вершин действительно будет эйлеровым циклом, причём и в случае ориентированного графа тоже.

□

4.2 Как удалять ребра

Проще всего хранить все списки смежности в set-подобной структуре и удалять их напрямую там:

```
const int maxn = 1e5;
set<int> g[maxn];

void euler(int v) {
  while (!g[v].empty()) {
    u = *g[v].begin();
    g[v].erase(u);
    g[u].erase(v);
    euler(u);
  }
  cout << v << " ";
}
```

Это будет работать за $O(m \log n)$, однако просто заменив дерево на хеш-таблицу (unordered_set) можно уменьшить время до линейного.

Также можно использовать более общий подход, который часто применяется в задачах, где ребра как-то изменяются. Создадим отдельный массив с мета-информацией о ребрах и будем хранить в списках смежности не номера вершин, а номера рёбер:

```
struct edge {
  int to;
  bool deleted;
};

vector<edge> edges;
stack<int> g[maxn];

void add_edge(int u, int v) {
  g[u].push(edges.size());
  edges.add({v, false});
  g[u].push(edges.size());
  edges.add({u, false});
}
```

Если добавлять каждое ребро неориентированного графа через add_edge, то у получившейся нумерации ребер будет интересное свойство: чтобы получить обратное к ребру e, нужно вычислить $e \oplus 1$.

Тогда во время обхода можно поддерживать эту информацию вместо какой-то сложной модификации структур:

```
void euler(int v) {
  while (!g[v].empty()) {
    e = g[v].top();
    g[v].pop();
    if (!edges[e].deleted) {
      edges[e].deleted = edges[e^1].deleted = true;
      euler(e.to);
    }
  }
  cout << v <<  " ";
}
```

Во всех вариантах реализации нужно быть чуть аккуратнее в случае, когда в графе есть петли и кратные ребра.

4.3 Эйлеров путь

Поговорим теперь про **эйлеровы пути**. Может, всё-таки можно что-нибудь сделать, даже если степени не всех вершин чётные?

Заметим, что, так как каждое ребро меняет четность степеней ровно двух вершин, и в пустом графе все степени изначально нулевые, то число вершин с нечетными степенями будет четным.

Если нечетных вершин ровно две, то можно сделать следующее: соединить их ребром, построить эйлеров цикл (ведь теперь степени всех вершин четные), а затем удалить это ребро из цикла. Если правильно сдвинуть этот цикл, мы получили эйлеров путь.

Альтернативно, можно просто запустить обход из одной из нечетных вершин и получить такой же результат.

Если нечетных вершин больше двух, то мы уже построить эйлеров путь не сможем, потому что любой эйлеров путь входит или покидает каждую вершину четное число раз, кроме, возможно двух своих концов.

Следствие. *Эйлеров путь существует тогда и только тогда, когда граф связен и количество вершин с нечётными степенями не превосходит 2.*

Задача. Дан связный неориентированный граф. Требуется покрыть его ребра наименьшим количеством непересекающихся путей. Асимптотика $O(n+m)$.

Задача. Сформулируйте и докажите аналогичные утверждения для случая ориентированного графа.

Источник: алгоритмика.