Конденсация графа

Материал из Algocode wiki
Перейти к: навигация, поиск

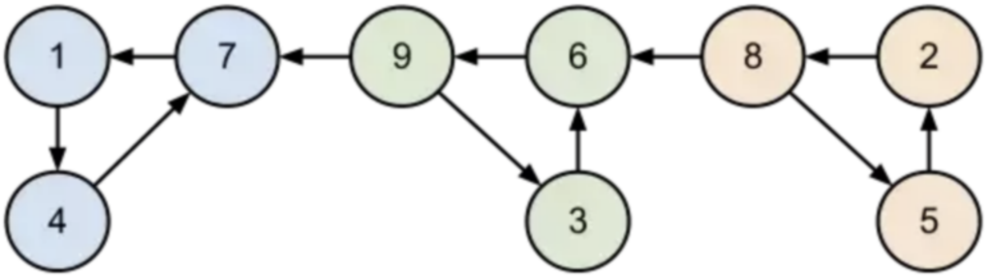
Компоненты сильной связности

Мы только что научились топологически сортировать ациклические графы. А что же делать с циклическими графами? В них тоже иногда требуется найти какую-то структуру.

Для этого можно ввести понятие *сильной связности*.

Определение. Две вершины ориентированного графа *связаны сильно* (англ. *strongly connected*), если существует путь из одной в другую и наоборот. Иными словами, они обе лежат в каком-то цикле.

Понятно, что такое отношение транзитивно: если a и b сильно связны, и b и c сильно связны, то a и c тоже сильно связны. Поэтому все вершины распадаются на *компоненты сильной связности* — такое разбиение вершин, что внутри одной компоненты все вершины сильно связаны, а между вершинами разных компонент сильной связности нет.



(/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Scc.png)

Самый простой пример сильно-связной компоненты — это цикл. Но это может быть и полный граф, или сложное пересечение нескольких циклов.

Часто рассматривают граф, составленный из самих компонент сильной связности, а не индивидуальных вершин. Очевидно, такой граф уже будет ациклическим, и с ним проще работать. Задачу о сжатии каждой компоненты сильной связности в одну вершину называют **конденсацией** графа, и её решение мы сейчас опишем.

Если мы знаем, какие вершины лежат в каждой компоненте сильной связности, то построить граф конденсации несложно: дальше нужно лишь провести некоторые манипуляции со списками смежности. Поэтому сразу сведем исходную задачу к нахождению самих компонент.

Лемма. Запустим dfs. Пусть A и B — две различные компоненты сильной связности, и пусть в графе конденсации между ними есть ребро $A \to B$. Тогда

$$\max_{a \in A}(tout_a) > \max_{b \in B}(tout_b)$$

Доказательство. Рассмотрим два случая, в зависимости от того, в какую из компонент dfs зайдёт первым.

Пусть первой была достигнута компонента A, то есть в какой-то момент времени dfs заходит в некоторую вершину v компоненты A, и при этом все остальные вершины компонент A и B ещё не посещены. Но так как по условию в графе конденсаций есть ребро $A \to B$, то из вершины v будет достижима не только вся компонента A, но и вся компонента B. Это означает, что при запуске из вершины v обход в глубину пройдёт по всем вершинам компонент A и B, а, значит, они станут потомками по отношению к v в дереве обхода, и для любой вершины $u \in A \cup B, u \neq v$ будет выполнено $tout_v] > tout_u$, что и утверждалось.

Второй случай проще: из B по условию нельзя дойти до A, а значит, если первой была достигнута B, то dfs выйдет из всех её вершин ещё до того, как войти в A.

Из этого факта следует первая часть решения. Отсортируем вершины по убыванию времени выхода (как бы сделаем топологическую сортировку, но на циклическом графе). Рассмотрим компоненту сильной связности первой вершины в сортировке. В эту компоненту точно не входят никакие рёбра из других компонент — иначе нарушилось бы условие леммы, ведь у первой вершины $tout$ максимальный. Поэтому, если развернуть все рёбра в графе, то из этой вершины будет достижима своя компонента сильной связности C', и больше ничего — если в исходном графе не было рёбер **из** других компонент, то в транспонированном не будет ребер **в** другие компоненты.

После того, как мы сделали это с первой вершиной, мы можем пойти по топологически отсортированному списку дальше и делать то же самое с вершинами, для которых компоненту связности мы ещё не отметили.

```
vector<int> g[maxn], t[maxn];
vector<int> order;
bool used[maxn];
int component[maxn];
int cnt_components = 0;

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

// маркировка компонент сильной связности
void dfs2(int v) {
    component[v] = cnt_components;
    for (int u : t[v])
        if (component[u] == 0)
            dfs2(u);
}


// в содержательной части main:

// транспонируем граф
for (int v = 0; v < n; v++)
    for (int u : g[v])
        t[u].push_back(v);

// запускаем топологическую сортировку
for (int i = 0; i < n; i++)
    if (!used[i])
        dfs1(i);

// выделяем компоненты
reverse(order.begin(), order.end());
for (int v : order)
    if (component[v] == 0)
        dfs2(v);
```

TL;DR:

1. Сортируем вершины в порядке убывания времени выхода.
2. Проходимся по массиву вершин в этом порядке, и для ещё непомеченных вершин запускаем dfs на транспонированном графе, помечающий все достижимые вершины номером новой

компонентой связности.

После этого номера компонент связности будут топологически отсортированы.

Источник — https://wiki.algocode.ru/index.php?title=Конденсация_графа&oldid=1470 (https://wiki.algocode.ru/index.php?title=Конденсация_графа&oldid=1470)

2-SAT

Материал из Algocode wiki

Перейти к: навигация, поиск

Ликбез. Конъюнкция — это «правильный» термин для логического «И» (обозначается $\vee$ или &). Конъюкция возвращает `true` тогда и только тогда, когда обе переменные `true`.

Ликбез. Дизъюнкция — это «правильный» термин для логического «ИЛИ» (обозначается $\wedge$ или |). Дизъюнкция возвращает `false` тогда и только тогда, когда обе переменные `false`.

Рассмотрим конъюнкцию дизъюнктов, то есть «И» от «ИЛИ» от каких-то перемений или их отрицаний. Например, такое выражение:

```
(a | b) & (!c | d) & (!a | !b)
```

Если буквами: (A ИЛИ B) И (НЕ C ИЛИ D) И (НЕ A ИЛИ НЕ B).

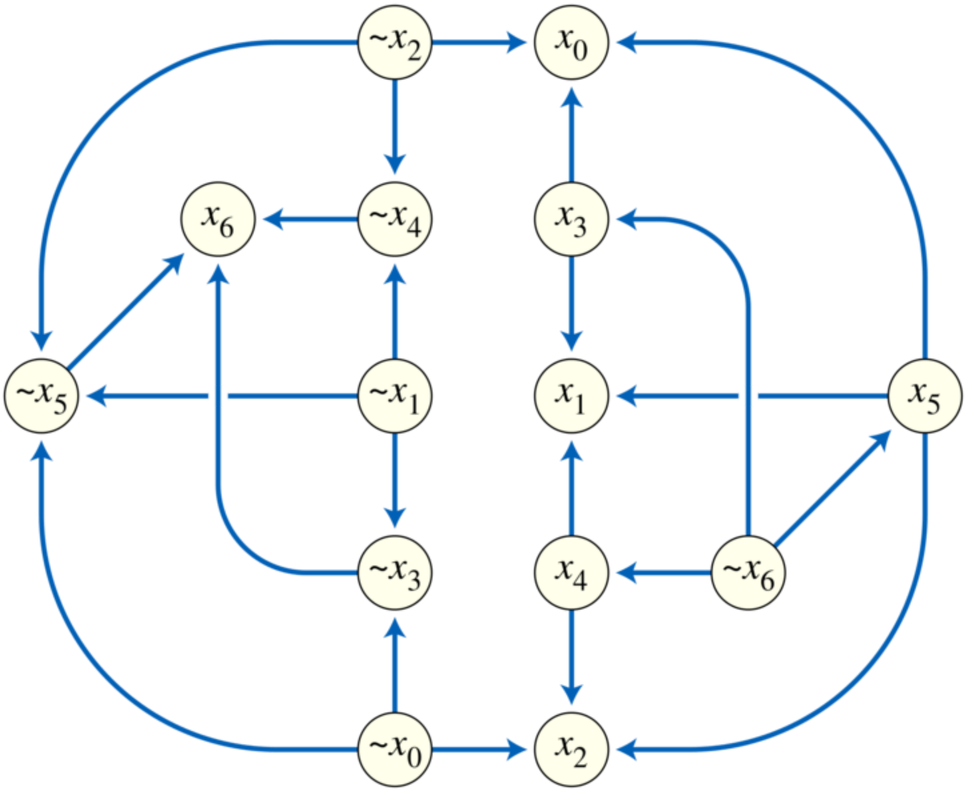
Можно показать (https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%8A%D1%8E%D0%BD%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BD%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%84%D0%BE%D1%80%D0%BC%D0%B0), что любую логическую формулу можно представить в таком виде.

Задача satisfiability (SAT) заключается в том, чтобы найти такие значения переменных, при которых выражение становится истинным, или сказать, что такого набора значений нет. Для примера выше такими значениями являются `a=1, b=0, c=0, d=1` (убедитесь, что каждая скобка стала `true`).

В случае произвольных формул эта задача быстро не решается. Мы же хотим решить её частный случай — когда у нас в каждой скобке ровно две переменные (2-SAT).

Казалось бы — причем тут графы? Заметим, что выражение $a \mid b$ эквивалентно $!a \to b \,\&\, !b \to a$. Здесь «$\to$» означает импликацию («если a верно, то b тоже верно»). С помощью это подстановки приведем выражение к другому виду — импликативному.

Затем построим граф импликаций: для каждой переменной в графе будет по две вершины, (обозначим их через x и $!x$), а рёбра в этом графе будут соответствовать импликациям.



(/index.php?title=%D0%A4%D0%B0%D0%B9%D0%BB:Https---upload.wikimedia.org-wikipedia-commons-thumb-2-2f-Implication_graph.svg-1920px-Implication_graph.svg.png)

Заметим, что если для какой-то переменной x выполняется, что из x достижимо $!x$, а из $!x$ достижимо x, то задача решения не имеет. Действительно: какое бы значение для переменной x мы бы ни выбрали, мы всегда придём к противоречию — что должно быть выбрано и обратное ему значение.

Оказывается, что это условие является не только достаточным, но и необходимым. Доказательством этого факта служит описанный ниже алгоритм.

Переформулируем данный критерий в терминах теории графов. Если из одной вершины достижима вторая и наоборот, то эти две вершины находятся в одной компоненте сильной связности. Тогда критерий существования решения звучит так: для того, чтобы задача 2-SAT имела решение, необходимо и достаточно, чтобы для любой переменной x вершины x и $!x$ находились в разных компонентах сильной связности графа импликаций.

Пусть решение существует, и нам надо его найти. Заметим, что, несмотря на то, что решение существует, для некоторых переменных может выполняться, что из x достижимо $!x$ или из $!x$ достижимо x (но не одновременно). В таком случае выбор одного из значений переменной x будет приводить к противоречию, в то время как выбор другого — не будет. Научимся выбирать из двух значений то, которое не приводит к возникновению противоречий. Сразу заметим, что, выбрав какое-либо значение, мы должны запустить из него обход в глубину/ширину и пометить все значения, которые следуют из него, т.е. достижимы в графе импликаций. Соответственно, для уже помеченных вершин никакого выбора между x и $!x$ делать не нужно, для них значение уже выбрано и зафиксировано. Нижеописанное правило применяется только к непомеченным ещё вершинам.

Утверждается следующее. Пусть $\operatorname{comp}[V]$ 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 $\operatorname{comp}[v] \leq \operatorname{comp}[w]$). Тогда, если $\operatorname{comp}[x] < \operatorname{comp}[!x]$, то выбираем значение !x, иначе выбираем x.

Докажем, что при таком выборе значений мы не придём к противоречию. Пусть, для определённости, выбрана вершина x (случай, когда выбрана вершина $!x$, доказывается также).

Во-первых, докажем, что из x не достижимо $!x$. Действительно, так как номер компоненты сильной связности $\operatorname{comp}[x]$ больше номера компоненты $\operatorname{comp}[!x]$, 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 из любой вершины y, достижимой из x недостижима $!y$. 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.

Итак, мы построили алгоритм, который находит искомые значения переменных в предположении, что для любой переменной x вершины x и $!x$ находятся в разных компонентах сильной связности. Выше показали корректность этого алгоритма. Следовательно, мы одновременно доказали указанный выше критерий существования решения.

Источник — https://wiki.algocode.ru/index.php?title=2-SAT&oldid=1411 (https://wiki.algocode.ru/index.php?title=2-SAT&oldid=1411)

Алгоритм Форда-Беллмана

Задача:

Для заданного взвешенного графа $G = (V, E)$ найти кратчайшие пути из заданной вершины s до всех остальных вершин. В случае, когда в графе G содержатся отрицательные циклы, достижимые из s, сообщить, что кратчайших путей не существует.

Содержание

Введение

Количество путей длины k рёбер можно найти с помощью метода динамического программирования. Пусть $d[k][u]$ — количество путей длины k рёбер, заканчивающихся в вершине u. Тогда $d[k][u] = \sum\limits_{v:vu\ \in E} d[k-1][v]$.

Аналогично посчитаем пути кратчайшей длины. Пусть s — стартовая вершина. Тогда $d[k][u] = \min\limits_{v:vu\ \in E}(d[k-1][v]\ +\ \omega(u,v))$, при этом $d[0][s] = 0$, а $d[0][u] = +\infty$

Лемма:

Если существует кратчайший путь от s до t, то $\rho(s,\ t)\ =\ \min\limits_{k=0..n-1} d[k][t]$

Доказательство:

$\triangleright$

Пусть кратчайший путь состоит из k ребер, тогда корректность формулы следует из динамики, приведенной ниже.

$\triangleleft$

Псевдокод

Используя приведенные формулы, алгоритм можно реализовать методом динамического программирования.

```
for k = 0 to |V| − 2                                    // вершины нумеруются с единицы
   for v ∈ V
      for (u,v) ∈ E
         d[k + 1][v] = min(d[k + 1][v], d[k][u] + ω(u,v))      // ω(u,v) — вес ребра uv
```

Также релаксацию можно свести к одномерному случаю, если не хранить длину пути в рёбрах. Одномерный массив будем обозначать d', тогда $d'[u] = \min(d'[u],\ d'[v] + \omega(vu))$

Корректность

Лемма:

Пусть $G = (V, E)$ — взвешенный ориентированный граф, s — стартовая вершина. Тогда после завершения k итераций цикла for выполняется неравенство $\rho(s,u) \leqslant d'[u] \leqslant \min\limits_{i=0..k} d[i][u]$.

Доказательство:

▷

Воспользуемся индукцией по k:

База индукции

При $k = 0$ выполнено: $\rho(s,u) \leqslant +\infty \leqslant +\infty$

Индукционный переход

Сначала докажем, что $\rho(s,u) \leqslant d'[u]$.
Пусть после $k-1$ итерации выполняется $\rho(s,u) \leqslant d'[u] \leqslant \min\limits_{i=0..k-1} d[i][u]$ для всех u.

Тогда после k итераций
$\rho(s,v) = \min\limits_{u \in V}(\rho(s,u) + \omega(uv)) \leqslant \min\limits_{u \in V}(d'[u] + \omega(uv)) = d'[v]$.

Переходим ко второму неравенству.
Теперь возможно два случая:

1. $\min\limits_{i=0..k+1} d[i][u] = d[k+1][u]$
2. $\min\limits_{i=0..k+1} d[i][u] = d[j][u] = \min\limits_{i=0..j} d[i][u]$

Рассмотрим 1 случай:

$$\min_{i=0..k+1} d[i][u] = d[k+1][u]$$
$$d'[u] \leqslant d'[v] + \omega(vu) \leqslant d[k][v] + \omega(vu) = d[k+1][u]$$

2 случай расписывается аналогично.

Таким образом переход выполнен и $\rho(s, u) \leqslant d'[u] \leqslant \min\limits_{i=0..k} d[i][u]$ выполняется.

◁

Реализация алгоритма и ее корректность

```
bool fordBellman(s):
    for v ∈ V
        d[v] = ∞
    d[s] = 0
    for i = 0 to |V| − 1
        for (u,v) ∈ E
            if d[v] > d[u] + ω(u,v)       // ω(u,v) — вес ребра uv
                d[v] = d[u] + ω(u,v)
    for (u,v) ∈ E
        if d[v] > d[u] + ω(u,v)
            return false
    return true
```

В этом алгоритме используется релаксация, в результате которой $d[v]$ уменьшается до тех пор, пока не станет равным $\delta(s, v)$. $d[v]$ — оценка веса кратчайшего пути из вершины s в каждую вершину $v \in V$.
$\delta(s, v)$ — фактический вес кратчайшего пути из s в вершину v.

Лемма:

Пусть $G = (V, E)$ — взвешенный ориентированный граф, s — стартовая вершина. Тогда после завершения $|V| - 1$ итераций цикла для всех вершин, достижимых из s, выполняется равенство $d[v] = \delta(s, v)$.

Доказательство:

▷

Рассмотрим произвольную вершину v, достижимую из s. Пусть $p = \langle v_0, \ldots, v_k \rangle$, где $v_0 = s$, $v_k = v$ — кратчайший ациклический путь из s в v. Путь p содержит не более $|V| - 1$ ребер. Поэтому $k \leqslant |V| - 1$.

Докажем следующее утверждение:

> После $n : (n \leqslant k)$ итераций первого цикла алгоритма, $d[v_n] = \delta(s, v_n)$

Воспользуемся индукцией по n:

База индукции

> Перед первой итерацией утверждение очевидно выполнено: $d[v_0] = d[s] = \delta(s, s) = 0$

Индукционный переход

> Пусть после $n : (n < k)$ итераций, верно что $d[v_n] = \delta(s, v_n)$. Так как (v_n, v_{n+1}) принадлежит кратчайшему пути от s до v, то $\delta(s, v_{n+1}) = \delta(s, v_n) + \omega(v_n, v_{n+1})$. Во

время $l+1$ итерации релаксируется ребро (v_n, v_{n+1}), следовательно по завершению итерации будет выполнено

$$d[v_{n+1}] \leqslant d[v_n] + \omega(v_n, v_{n+1}) = \delta(s, v_n) + \omega(v_n, v_{n+1}) = \delta(s, v_{n+1}).$$

Ясно, что $d[v_{n+1}] \geqslant \delta(s, v_{n+1})$, поэтому верно что после $l+1$ итерации $d[v_{n+1}] = \delta(s, v_{n+1})$.
Индукционный переход доказан.

Итак, выполнены равенства $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$.
◁

Теорема:

Пусть $G = (V, E)$ — взвешенный ориентированный граф, s — стартовая вершина. Если граф G не содержит отрицательных циклов, достижимых из вершины s, то алгоритм возвращает $true$ и для всех $v \in V$ $d[v] = \delta(s, v)$. Если граф G содержит отрицательные циклы, достижимые из вершины s, то алгоритм возвращает $false$.

Доказательство:
▷

Пусть граф G не содержит отрицательных циклов, достижимых из вершины s.

Тогда если вершина v достижима из s, то по лемме $d[v] = \delta(s, v)$. Если вершина v не достижима из s, то $d[v] = \delta(s, v) = 1$ из несуществования пути.

Теперь докажем, что алгоритм вернет значение $true$.

После выполнения алгоритма верно, что для всех $(u, v) \in E,\ d[v] = \delta(s, v) \leqslant \delta(s, u) + \omega(u, v) = d[u] + \omega(u, v)$, значит ни одна из проверок не вернет значения $false$.

Пусть граф G содержит отрицательный цикл $c = v_0, \ldots, v_k$, где $v_0 = v_k$, достижимый из вершины s. Тогда $\sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i) < 0$.

Предположим, что алгоритм возвращает $true$, тогда для $i = 1, \ldots, k$ выполняется $d[v_i] \leqslant d[v_{i-1}] + \omega(v_{i-1}, v_i)$.

Просуммируем эти неравенства по всему циклу: $\sum\limits_{i=1}^{k} d[v_i] \leqslant \sum\limits_{i=1}^{k} d[v_{i-1}] + \sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i)$.

Из того, что $v_0 = v_k$ следует, что $\sum\limits_{i=1}^{k} d[v_i] = \sum\limits_{i=1}^{k} d[v_{i-1}]$.

Получили, что $\sum\limits_{i=1}^{k} \omega(v_{i-1}, v_i) \geqslant 0$, что противоречит отрицательности цикла c.

◁

Сложность

Инициализация занимает $\Theta(V)$ времени, каждый из $|V| - 1$ проходов требует $\Theta(E)$ времени, обход по всем ребрам для проверки наличия отрицательного цикла занимает $O(E)$ времени. Значит алгоритм Беллмана-Форда работает за $O(VE)$ времени.

Нахождение отрицательного цикла

Приведенная выше реализация позволяет определить наличие в графе цикла отрицательного веса. Чтобы найти сам цикл, достаточно хранить вершины, из которых производится релаксация.

Если после $|V| - 1$ итерации найдется вершина v, расстояние до которой можно уменьшить, то эта вершина либо лежит на каком-нибудь цикле отрицательного веса, либо достижима из него. Чтобы найти вершину, которая лежит на цикле, можно $|V| - 1$ раз пройти назад по предкам из вершины v. Так как наибольшая длина пути в графе из $|V|$ вершин равна $|V| - 1$, то полученная вершина u будет гарантированно лежать на отрицательном цикле.

Зная, что вершина u лежит на цикле отрицательного веса, можно восстанавливать путь по сохраненным вершинам до тех пор, пока не встретится та же вершина u. Это обязательно произойдет, так как в цикле отрицательного веса релаксации происходят по кругу.

```
int[] negativeCycle(s):
    for v ∈ V
        d[v] = 1
        p[v] = -1
    d[s] = 0
    for i = 1 to |V| − 1
        for (u, v) ∈ E
            if d[v] > d[u] + ω(u, v)
                d[v] = d[u] + ω(u, v)
                p[v] = u
    for (u, v) ∈ E
        if d[v] > d[u] + ω(u, v)
            for i = 0 to |V| − 1
                v = p[v]
            u = v
            while u != p[v]
                ans.add(v)            // добавим вершину к ответу
                v = p[v]
            reverse(ans)
            break
  return ans
```

Источники информации

- Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 2-е изд — М.: Издательский дом «Вильямс», 2009. — ISBN 978-5-8459-0857-5.
- MAXimal :: algo :: Алгоритм Форда-Беллмана (http://e-maxx.ru/algo/ford_bellman)