Алгоритмы и Структуры Данных

Коченюк Анатолий

2 октября 2021 г.

Глава 1

Алгоритмы на графах и строках и фане.

1.1 Графы и обход в ширину

кружочки и стрелочки

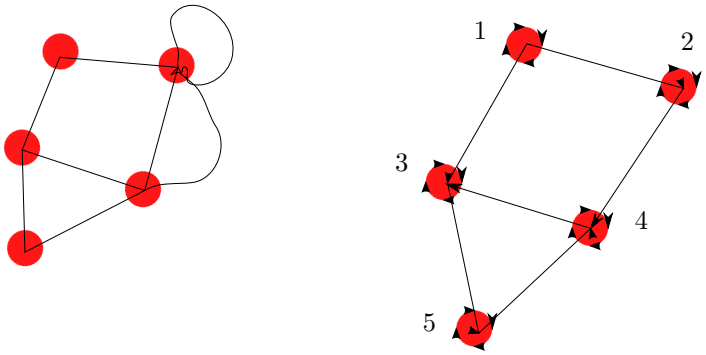

Рис. 1.1: fex

n вершин, m рёбер, $T(n, m)$

Связность – из любой вершины можно дойти до любой другой

$m \geqslant n - 1 \quad m \leqslant \frac{n(n-1)}{2}$ если связный и нет приколов с кратными рёбрами

и петлями

Определение 1. Матрица смежности – матрица $m(a,b) = \begin{cases} 1, \text{a и b связаны ребром} \\ 0, \text{иначе} \end{cases}$

Пример. 1. 2, 3

2. 4

3. 5

4. 2, 3

5. 4

более компактный и полезный способ хранить что с чем связано

Определение 2. Компонента связаности – класс эквивалентности по отношению эквивалентности быть связанным.

Определение 3 (Поиск в глубину)**.** Дали нам граф. Берём вершину и помечаем все вершины, которые из неё достижимы. Всё, что мы пометили это кмпонента связности. Дальше берём непомеченную и аналогично выделяем вторую компоненту и так пока вершины не закончатся

```
1    dfs(v):
2        mark[v] = True
3        for vu \in out(v):
4            if !mark[u]:
5                dfs(u)
```

Лемма 1. Мы пометили все достижимые и только их

Доказательство. Ходим только по рёбрам, значит все помеченные вершины достижимы из s

Есть вершина s и достижимая v. Предположим, что мы не дошли. Значит на пути до v была первая вершина, до которой мы не дошли. Но дошли до соседеней с ней, значит запустился оттуда и ведел непомеченную. Он её пометит в цикле, противоречие. ■

1.2 Что можно делать поиском в глубину в ориентированном графе

Определение 4. Топологическая сортировка – сортировка вершин, чтобы все рёбра шли слева направо

Задача 1. Построить топологическую сортировку у ациклического графа.

Задача 2. В ациклическом графе есть вершина, в которую ничего не входит. Вставим самой левой в сортировке и уберём из графа.

Возьмём любую вершину, в которую ничего не входит. Добавляем в сортировку, убираем из графа.

```
z = []
for v = 0 .. n-1:
     if deg[v] = 0
         z.insert(v)
 while !z.empty():
     x = z.remove()
     for y in out(x):
         deg[y]--
         if deg[y] == 0:
             z.insert(y)
```

Время алгоритма $O(m)$

Доказательство.

```
        mark[v] = True
    for m in out(v):
        if !mark[u]:
            dfs(u)
    topsort.add(v)

```

■

Утверждение 1. Все рёбра идут слева направо.

Задача 3. Как понять есть ли циклы

Доказательство. Запустить topsort. Если получилась фигня, значит есть цикл. ■

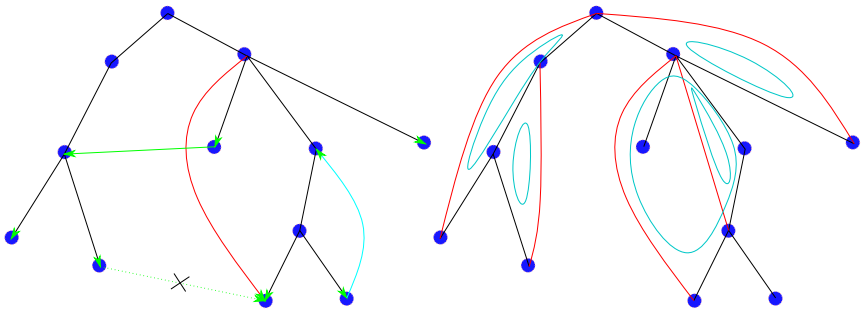

Рис. 1.2: dfstree

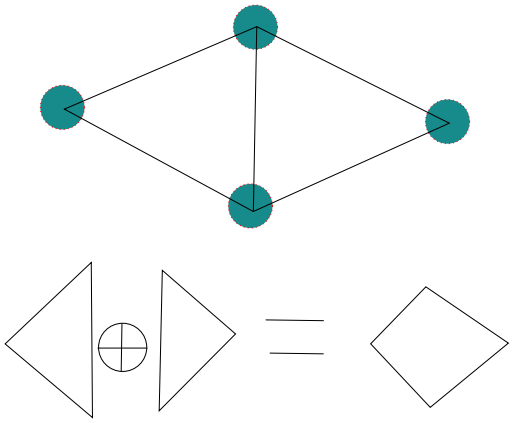

Рис. 1.3: xor

Выберем для всех рёбер, до которых мы не дошли в dfs по циклу из него и рёбер из dfs. Из получившихся циклов можно собрать (ксорами множеств) любой цикл)

1.3 Связность в ориентированных графах

Замечание. Сильная связность является отношением эквивалентности. Можно выбирать классы эквивалентности – компоненты связности. После

этого можно построить конденсацию – граф на компонентах связности, где обозначается односторонняя связь между компонентами.

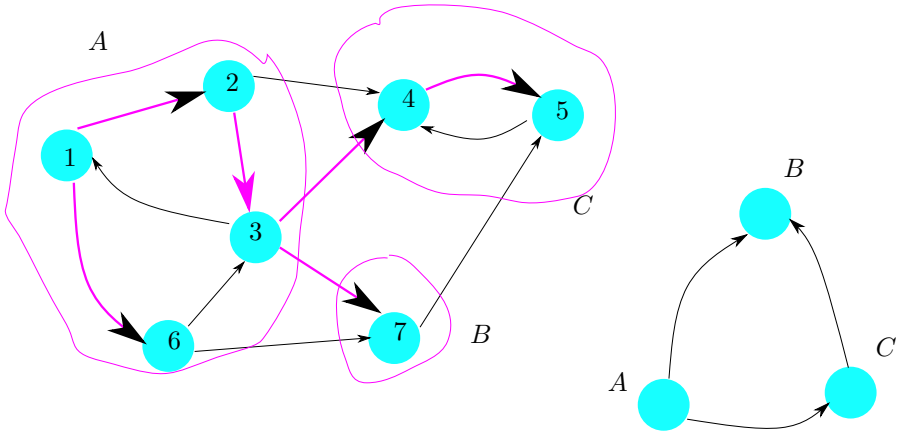

Рис. 1.4: dvureb

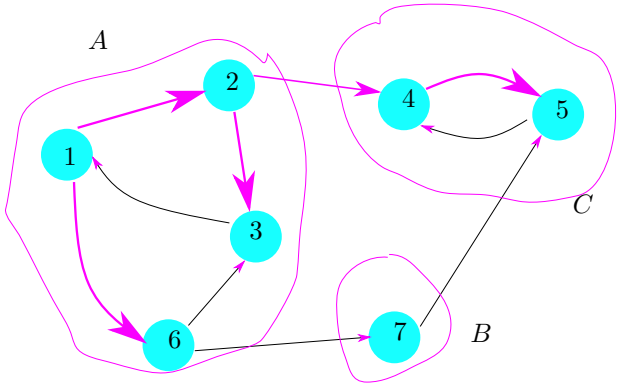

Рис. 1.5: neprimer

Алгоритм:

1. запускаем dfs, записываем вершины в порядке выхода.
2. Если идти по входящим рёбрам из первой вершины в списке, то мы пометим все вершины в компоненте связности 1. Затем перейдя к следующей не помеченной вершине, мы пометим вторую компоненту и

т.д.

```
dfs(v):
    ...
    p.push_back(v)
    -> 1 6 2 3 7 4 5 -- dvureb
    -> 1 6 7 2 3 4 5 -- neprimer
```

```
for i = 0 .. n-1
    dfs1(i)
reverse(p)
for i  = 0 .. n-1:
    if !mark[p[i]]:
        dfs2(p[i])
```

Доказательство. Возьмём компоненту связаности. Возьмём первую вершину оттуда, в которую зашёл dfs. Он оттуда не уйдёт, пока всё в ней не пометит.

На компонентах сильной связности есть "топсорт" по первым вершинам в компонентах, которые рассматривает dfs. Это гарантирует нам, что мы будем запускать dfs по обратным рёбрам в компонентах, все входящие компоненты в которую мы уже пометили. Значит dfs2 останется только идти внутри компоненты, что нам и требовалось. ■

Задача 4 (2-SAT)**.** $(x \vee y) \wedge (!y \vee !x) \wedge (z \vee !x) = 1$

$x \vee y = !x \to y$

Если есть стрелки в обе стороны между x и $\neg x$, то это противоречие.

$A \implies B \iff \neg B \implies \neg A$ – кососимметричность импликации

Если есть пусть между u и v, то есть обратный между $\neg v$ в $\neg u$.

Алгоритм: в конденсации строим топсорт (он ациклический). В каждой паре компонент, ту, которая правее, делаем True.

Доказательство. Берём левую вершину. Все следствия из неё выполняются. Из неё рёбра только выходят. В ней значение False, значит все седствия выполняются. Рассмотрим симметричную к ней, она возможно где-то в середине. В неё только входят рёбра, у неё всё хорошо. Убираем их их по индукции всё хорошо. ■

1.4 Двусвязность

Рёберно-двуязный – два пути не пересекающихся по рёбрам

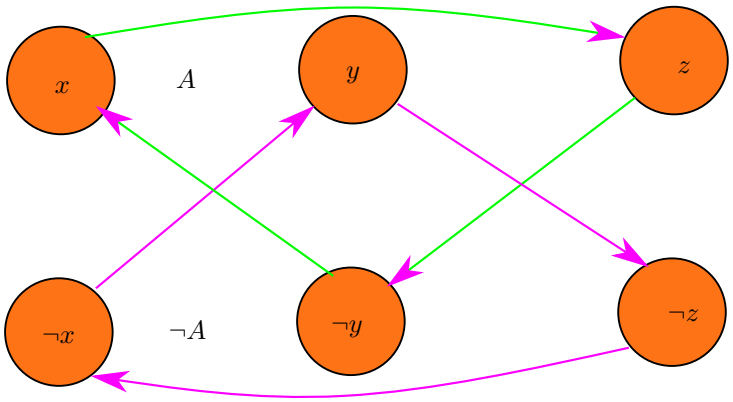

Рис. 1.6: vershinki

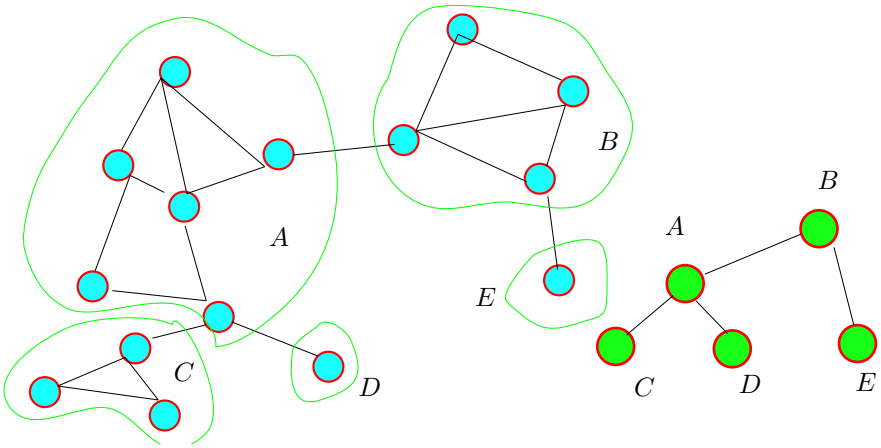

Рис. 1.7: лфлщштшигвкфзр

Утверждение 2. dfs обязательно пройдёт по всем мостам.

Если в поддереве вершины есть рёбро выше неё, то ребро с ней уже не мост.

```
    dfs(v, p):
        t_in[v] = T++
        up[v] = t_in[v]
        mark[v] = True
        buf.push(v)
        for u in out[v]:
```

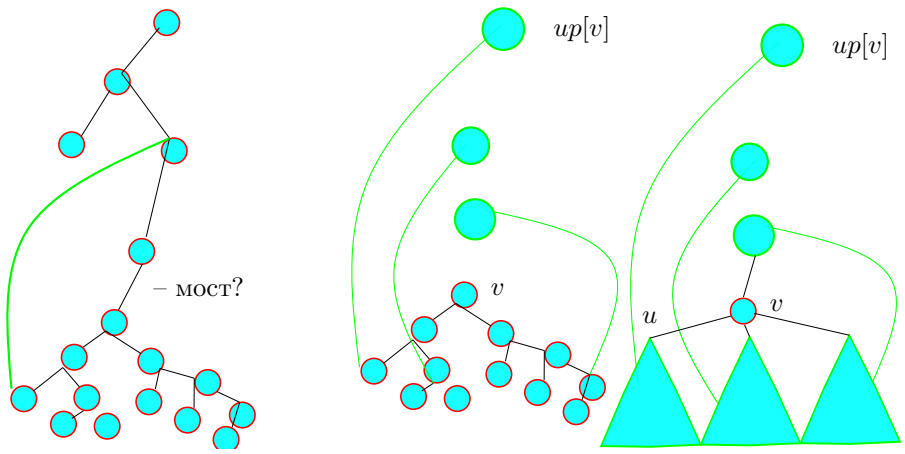

Рис. 1.8: sadfs

```
            if u = p:
                continue
            if !mark[u]:
                dfs(u,v)
                up[v] = min(up[v], up[u])
            else:
                up[v] = min(up[v], t_in[u])
        if up[v] == t_in[v]:
            while True:
                x = buf.pop()
                comp.add(x)
                if x == v:
                    break
```

Утверждение 3. $\forall u \in child[v]: \quad up[u] < t_{in}[v] \implies v$ – не точка сочления

верно для всех вершин, кроме корня

```
dfs(v, p):
    t_in[v] = T++
    up[v] = t_in[v]
    mark[v] = True
    ok = False
    c = 0
    for u in out[v]:
        if u = p:
            continue
        if !mark[u]:
            dfs(u,v)
            c++
            up[v] = min(up[v], up[u])
```

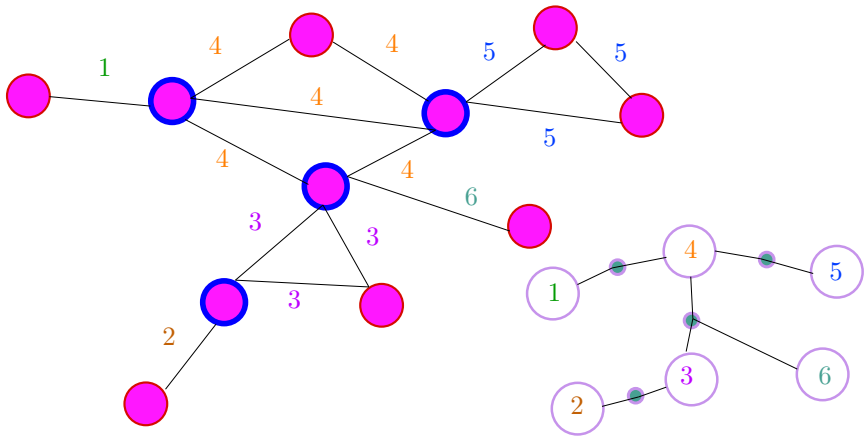

Рис. 1.9: лфкештлгрщсргтфкшыщмфе

```
14                  if up[u] >= t_in[v]:
15                      ok = True
16              else:
17                  up[v] = min(up[v], t_in[u])
18
19          if ok || p = None && c > 1:
20              v - точка сочленения
```

```
1       dfs(v, p):
2           t_in[v] = T++
3           up[v] = t_in[v]
4           mark[u] = True
5           ok = False
6           for u in out[v]:
7               if u = p:
8                   continue
9               if !mark[u]:
10                  buf.push(vu)
11                  dfs(u,v)
12                  up[v] = min(up[v], up[u])
13                  if up[u] >= t_in[v]
14                      while True:
15                          e = buf.pop()
16                          comp.add(e)
17                          if r = (vu):
18                              break
19              else:
20                  up[v] = min(up[v], t_in[u])
21                  if t_in[u] < t_in[v]:
22                      buf.push(vu)
```

Задача 5. Хотим найти эйлеров цикл.

Идём пока идём. Если не идётся, добавляем последнее ребро в циклс и смотрим идётся ли из предыдущей вершины... степени чётные, утыкаемся туда, откуда начали...

для ориентированного графа то же самое, та же логика..

```
1    dfs(v):
2         vu -- любое непомеченное ребро из v
3         if vu = None:
4              return
5          пометить vu
6          dfs(u)
7          ans.add(vu)
```

Структура данных: умеет проверять пустая ли и брать любой элемент .. любая структура данных.

1.5 дерево доминатор

Определение 5. s – помеченная вершина

u доминирует v, если любой путь от s до v есть u

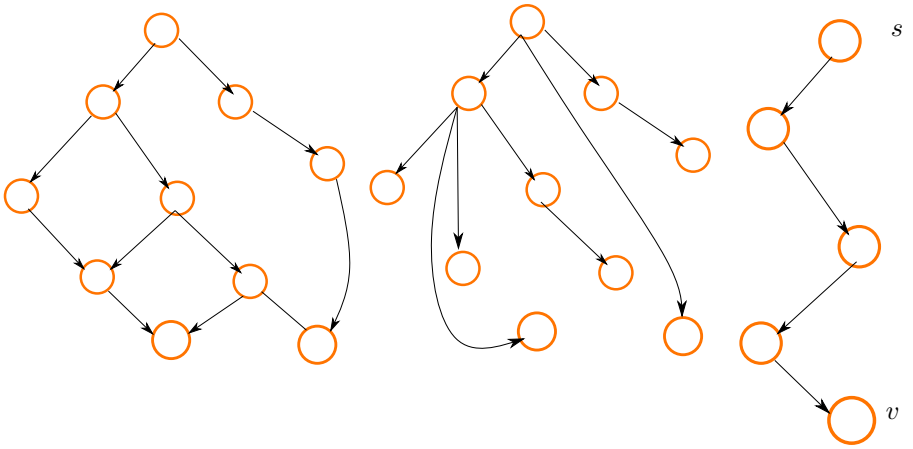

Рис. 1.10: krasiv

Свойство 1. u доминирует v, v доминирует $w \implies u$ доминирует w

u дом v, w дом $v \implies u$ дом w (или наоборот в зависимости от порядка в пути от s до v)

Доминаторы образуют цепочки. Нам достаточно знать ближайшего доминатора для всех вершин, чтобы выстроить полную цепочку

Дерево доминаторов – дерево, в котором вершины подвешены к своим ближайшим додминаторам

Если граф ацикличен, то идём по топсорту и подвешиемся к лца своих ближайших предков

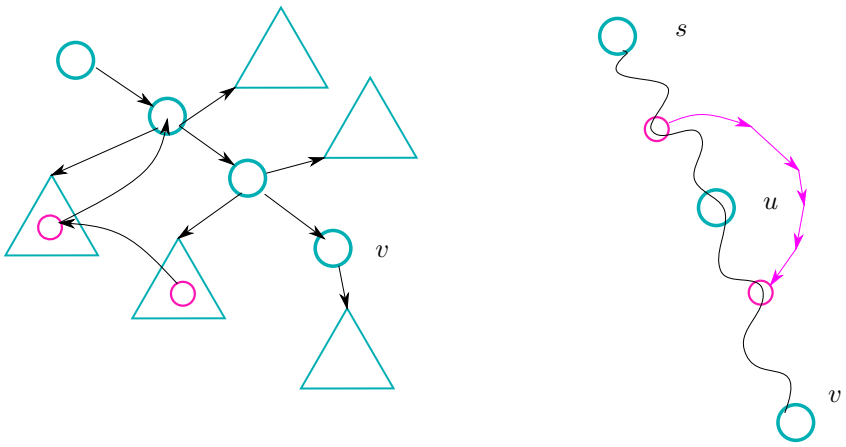

Рис. 1.11: cicles

Все сравнения вершин дальше – по времени входа в обходе в глубину

Лемма 2. $u < v$, есть путь из u в v. Тогда на этом пути мы обязательно встретим предком v

План:

1. DFS, t_{in}
2. $semi - dom[v]$
3. $dom[v]$

Определение 6. Полудоминатор

u полудоминирует v, если существует путь от u до v все вершины которого $> v$

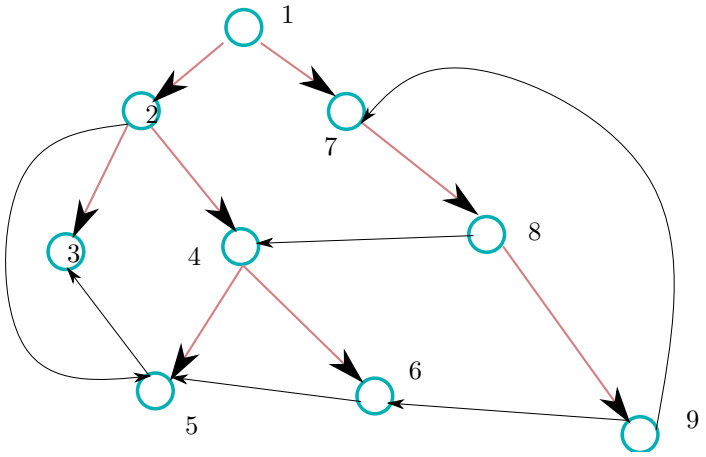

Рис. 1.12: semi

$sdom[v]$ – минимальный (по времени входа) полудоминатор v

1. $u \to v$ путь из одного ребра, перебираем все входящие рёбра. Берём минимум по ним
2. $u \to \ldots \to w \to v$. Переберём последнюю промежуточную вершину, она должна быть $> v$

   x – первая вершина на ветке между lca(u,w) и самой w

   $u \to \ldots \to x \to \ldots \to w \to v$

   u – полудоминатор $x \implies u$ полудоминатор v

```
sdom(v) = min(
  u: есть ребро u->v sdom[x]
  sdom[x]: x in ветке [w, lca(u,w)], w->v)

```

   Суммарное время: $O(m \log n)$. Если упороться в link-eval можно $O(m\alpha(m,n))$. Ребята могут делать за линию, но там совсем плохо

   $dom[s] \leqslant sdom[s]$

   (a) Пусть для всех $u \in [v, \ldots, sdom[v])$ $\quad sdom[u] \geqslant sdom[v]$

пускай не так, значит можем обойти. Найдём первую вершину на этом пути ниже полудоминатора. x – предок u (есть по лемме) $u \to x \to u \to v$, тогда x дом u

(b) $\exists u: \quad sdom[u] < sdom[v]$. Пусть $u : sdom[u]$ – минимальная. Тогда $dom[v] = dom[u]$

Алгоритм: ищем минимум по полудоминаторм между вершиной и её полдомнатором.

```
    for v = ...
        u = min_sdom [v, ..., sdom[v])
        if sdom[u] >= sdmo[v]
            dom[v] = sdom[v]
        else
            dom[v] = dom[u] # u < v -- нужно считать доминаторы по
    возрастанию номеров. Но с LinkEval хочется считать по уменьшению. На
    самом деле приходиться запоминать равенство гиперссылкой и потом
    высстанавливать
```

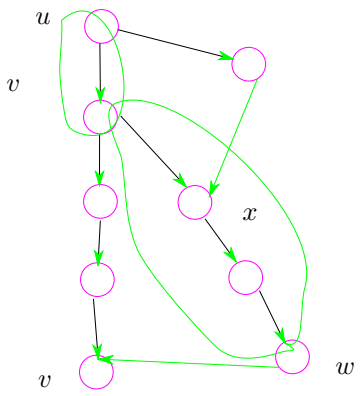

Рис. 1.13: semi2