

Алгоритмы и Структуры Данных

Коченюк Анатолий

12 октября 2021 г.

Глава 1

Алгоритмы на графах и строках и фане.

1.1 Графы и обход в ширину

кружочки и стрелочки

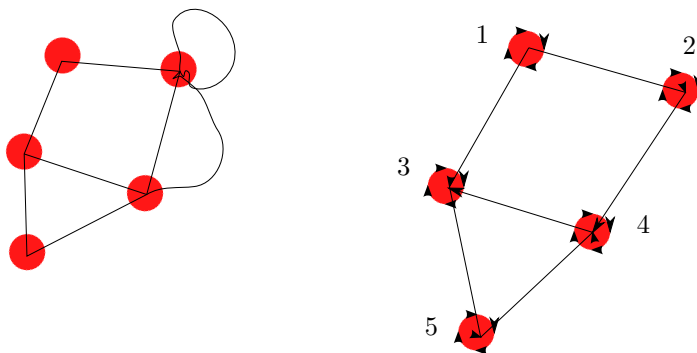


Рис. 1.1: fex

n вершин, m рёбер, $T(n, m)$

Связность – из любой вершины можно прийти до любой другой

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

и петлями

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

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 связности – 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 поместили это кмпонента связности. Дальше берём непомеченную и аналогично выделяем вторую компоненту и так пока вершины не закончатся

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

Лемма 1. Мы поместили 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 соседей с ней, значит запустился оттуда и велел 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.

```
1  z = []
2  for v = 0 .. n-1:
3      if deg[v] == 0:
4          z.insert(v)
5      while !z.empty():
6          x = z.remove()
7          for y in out(x):
8              deg[y]--
9              if deg[y] == 0:
10                 z.insert(y)
```

Время алгоритма $O(m)$

Доказательство.

```
mark[v] = True
2  for m in out(v):
3      if !mark[u]:
4          dfs(u)
5  topsort.add(v)
6
```

■

Утверждение 1. Все рёбра идут слева направо.

Задача 3. Как понять есть ли циклы

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

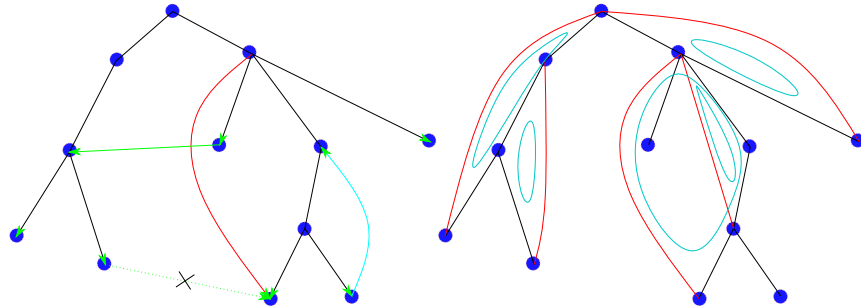


Рис. 1.2: dfstree

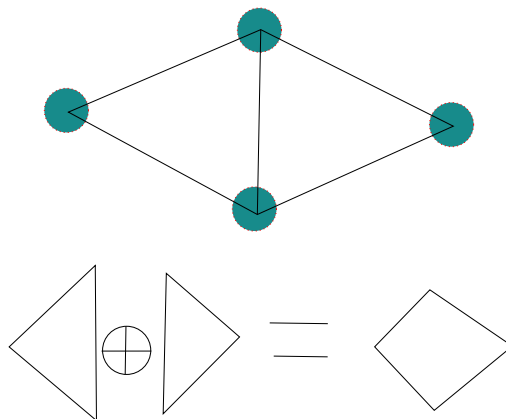


Рис. 1.3: xor

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.

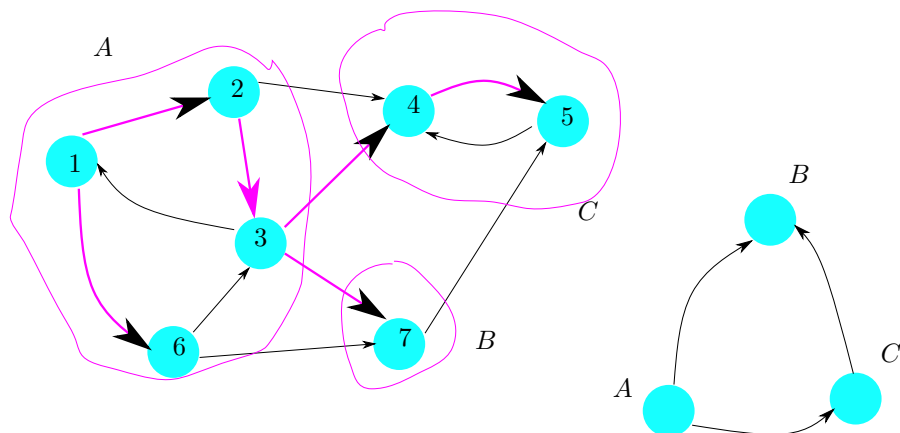


Рис. 1.4: dvureb

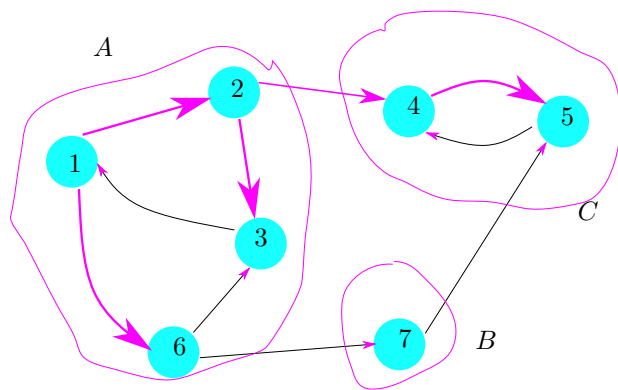


Рис. 1.5: neprimer

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.

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

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

Доказательство. Возьмём компоненту связности. 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). $(x \vee y) \wedge (!y \vee !x) \wedge (z \vee !x) = 1$

$x \vee y = !x \rightarrow y$

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 следствия выполняются. Рассмотрим симметричную к ней, она возможно где-то в середине. В неё только входят рёбра, у неё всё хорошо. Убираем их по индукции всё хорошо. ■

1.4 Двусвязность

Рёберно-двухсвязный – два пути не пересекающихся по рёбрам

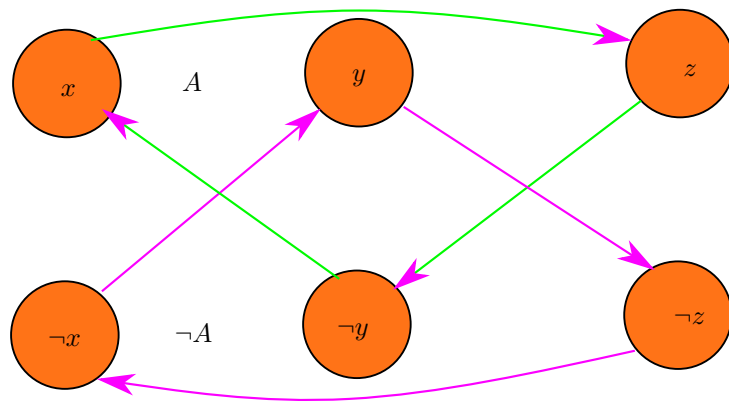


Рис. 1.6: vershinki

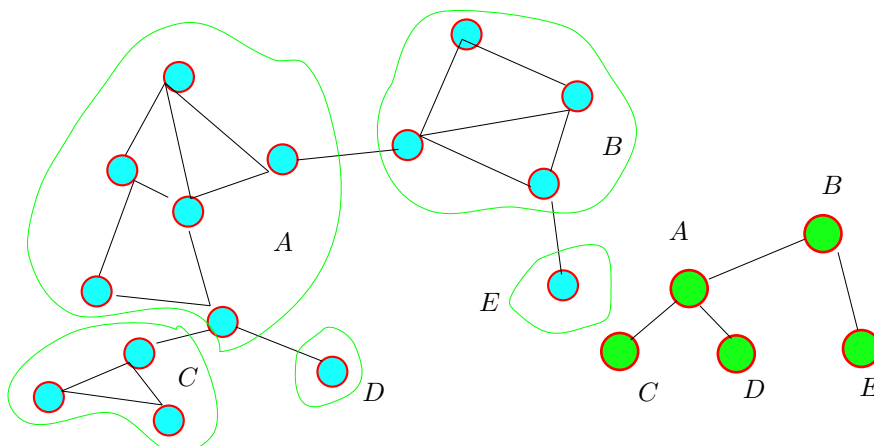


Рис. 1.7: лфлщштшигвкфзр

Утверждение 2. dfs обязательно пройдёт по всем мостам.

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

```

1  dfs(v, p):
2      t_in[v] = T++
3      up[v] = t_in[v]
4      mark[v] = True
5      buf.push(v)
6      for u in out[v]:

```

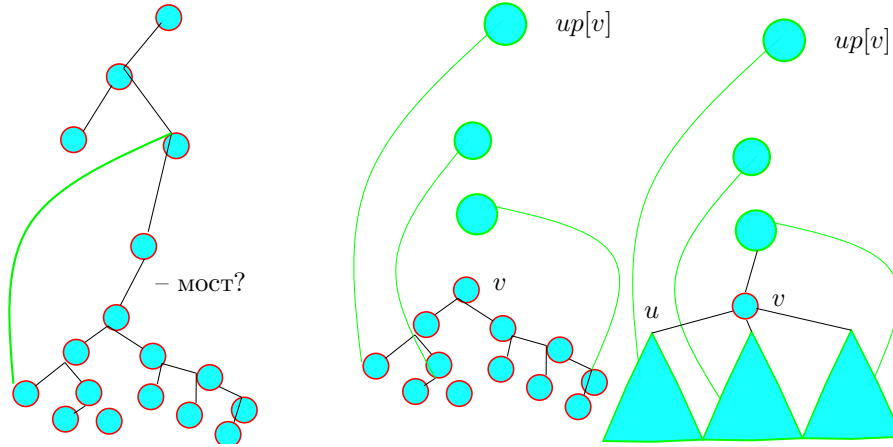


Рис. 1.8: sadfs

```

7         if u == p:
8             continue
9         if not mark[u]:
10             dfs(u, v)
11             up[v] = min(up[v], up[u])
12         else:
13             up[v] = min(up[v], t_in[u])
14         if up[v] == t_in[v]:
15             while True:
16                 x = buf.pop()
17                 comp.add(x)
18                 if x == v:
19                     break

```

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

верно для всех вершин, кроме корня

```

1     def dfs(v, p):
2         t_in[v] = T++
3         up[v] = t_in[v]
4         mark[v] = True
5         ok = False
6         c = 0
7         for u in out[v]:
8             if u == p:
9                 continue
10            if not mark[u]:
11                dfs(u, v)
12                c++
13            up[v] = min(up[v], up[u])

```

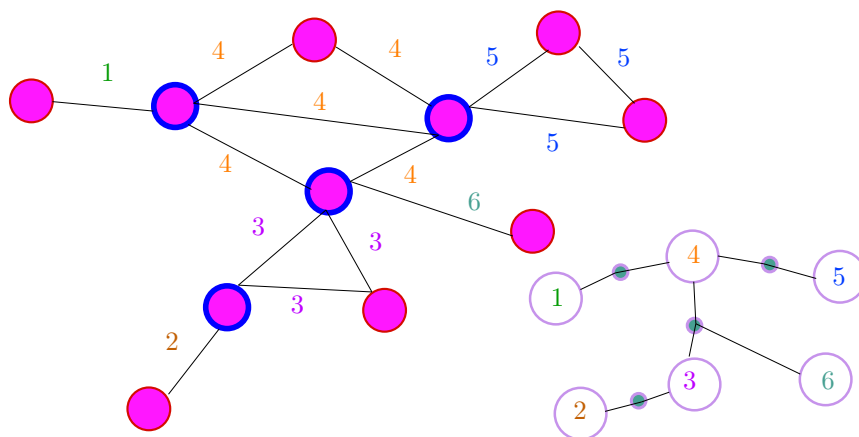


Рис. 1.9: лфкештлгрщсргтфкшыщмфе

```

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  def 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 цикл 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

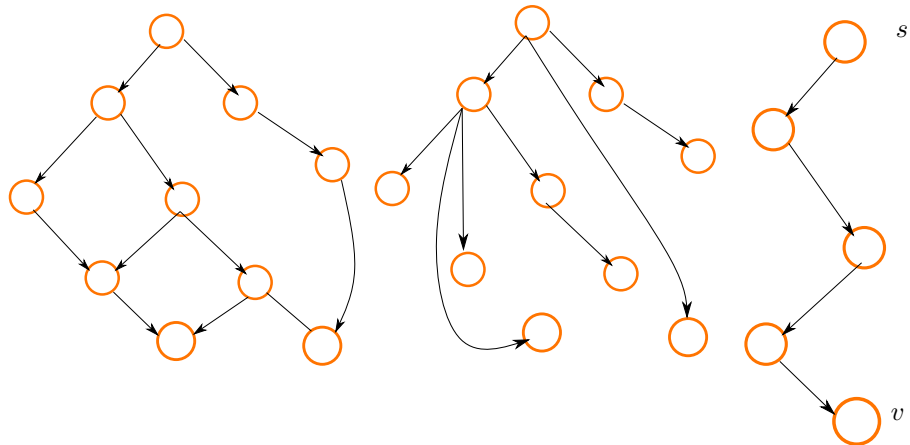


Рис. 1.10: krasiv

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 доминаторам

Если граф ацикличен, то идём по топсорту и подвешиваемся к лца своим ближайшим предкам

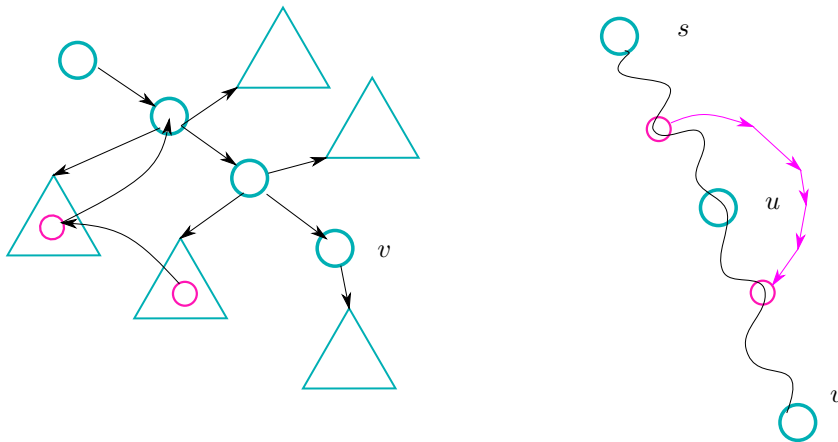


Рис. 1.11: cicles

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 $> v$

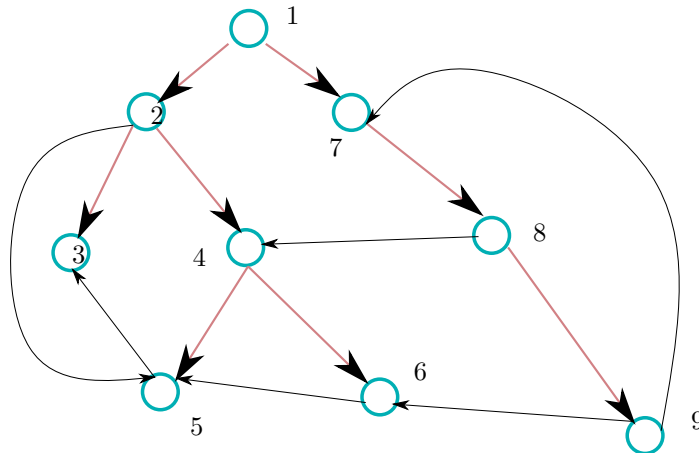


Рис. 1.12: semi

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

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

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

$u \rightarrow \dots \rightarrow x \rightarrow \dots \rightarrow w \rightarrow v$

u – полудоминатор $x \implies u$ полудоминатор v

```
1 sdom(v) = min(  
2   u: есть ребро u->v sdom[u]  
3   sdom[x]: x in ветке [w, lca(u,w)], w->v)  
4
```

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

$$dom[s] \leq sdom[s]$$

(а) Пусть для всех $u \in [v, \dots, sdom[v]]$ $sdom[u] \geq sdom[v]$

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

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

Алгоритм: ищем минимум по полудоминатору между вершиной и её полдоминатором.

```

1  for v = ...
2      u = min_sdom [v, ..., sdom[v]]
3      if sdom[u] >= sdom[v]
4          dom[v] = sdom[v]
5      else
6          dom[v] = dom[u] # u < v -- нужно считать доминаторы по
                             возрастанию номеров. Но с LinkEval хочется считать по уменьшению. На
                             самом деле приходится запоминать равенство гиперссылкой и потом
                             высстанавливать

```

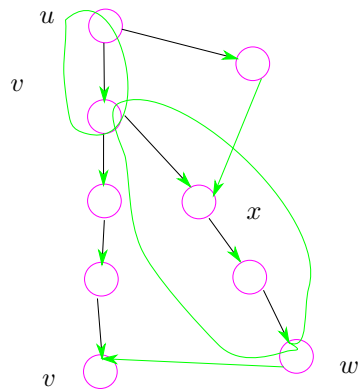


Рис. 1.13: semi2

1.6 Минимальное остовное дерево

Хотим выбрать остовное дерево минимальное по сумме весов рёбер.

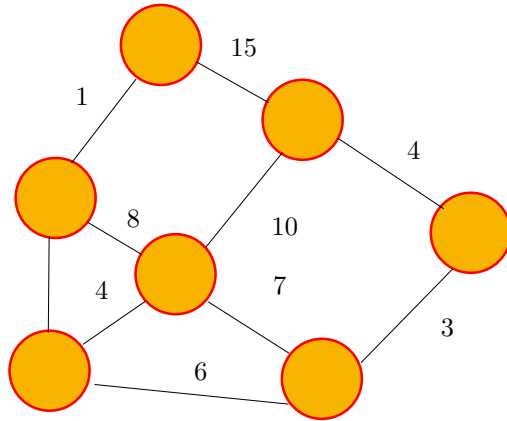


Рис. 1.14: ostree

Определение 7. Разрез – взяли все вершины графа и поделили их на две части.

Посмотрим на рёбра между двумя частями разреза. Пусть uv – минимальное такое ребро $\min_{u \in A, v \in B} \Rightarrow uv \in MST$ – minimal spanning tree.

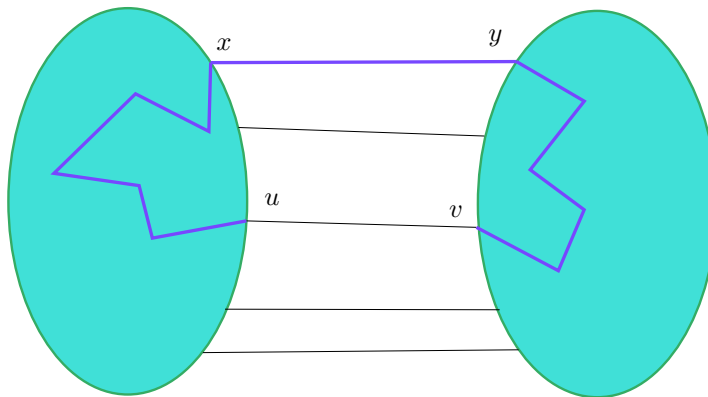


Рис. 1.15: spliceproof

Лемма 3 (О разрезе). *Доказательство.* Если не было этого ребра, то в графе тоже было остовное дерево. Добавим обратно и в остовном дереве образуется цикл. В котором будет uv и ещё одно ребро между A и B . второе будет по выбору uv больше по весу, чем uv , значит оно не минимальное, т.к. замена в дереве этого ребра на uv делает остовное дерево с меньшим весом. A значит, оно лежит в дереве. ■

Утверждение 4 (Алгоритм Караскала). берём минимальное по весу ребро. Если у нему не прилегает выделенных рёбер, добавляем его в дерево. Если есть, то берём целиком связанные рёбра с одной из сторон как часть разреза. Это ребро минимальное по выбору, оно минимальное ребро на разрезе, добавляем его.

```

1  for uv in E.sorted():
2      if find(u) != find(v):
3          T = T + uv
4          union(u,v)
5

```

Утверждение 5 (Алгоритм Прима).

```

1  A = {s}
2  repeat(n-1)
3      uv -- min: u in A, v not in A
4      A = A U {v}
5      T = T U {uv}
6

```

```

1  repeat(n-1)
2      uv = Q.remove_min()
3      A = A U {v}
4      T = T U {uv}
5      for uv:
6          if w in A:
7              Q.remove(vw)
8          else
9              Q.add(vw)

```

```

1  repeat(n-1)
2      v = Q.remove_min() // min d[v]
3      A = A U {v}
4      for vw:
5          if w not in A:
6              d[w] = min(d[w], w_{vw})
7              Q.update(w)

```

$O(m \log n)$

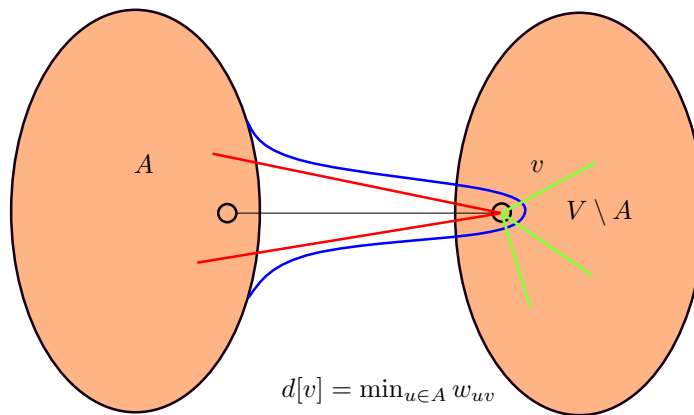


Рис. 1.16: prima

В полном графе лучше использовать массив для $n^2 + m$. Можно Фибоначчиеву кучу и тогда $n \log n + m$

Но люди умеют страшнее. Например $m\alpha(m, n)$. Рандомизованно за m могут.

Утверждение 6 (Баруфка). Берём граф.

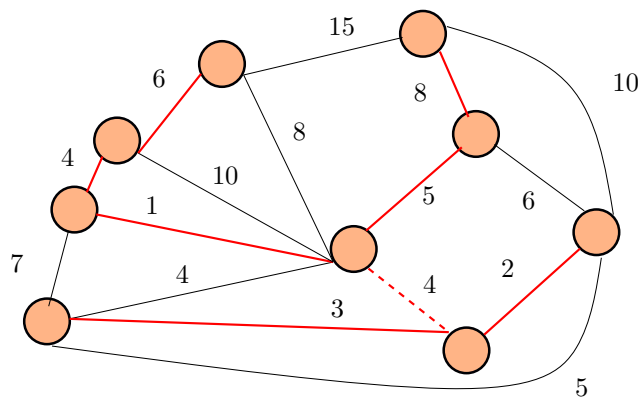


Рис. 1.17: barufka

Помечаем для каждой вершины её минимальное ребро. ...

Возьмём ориентированный граф

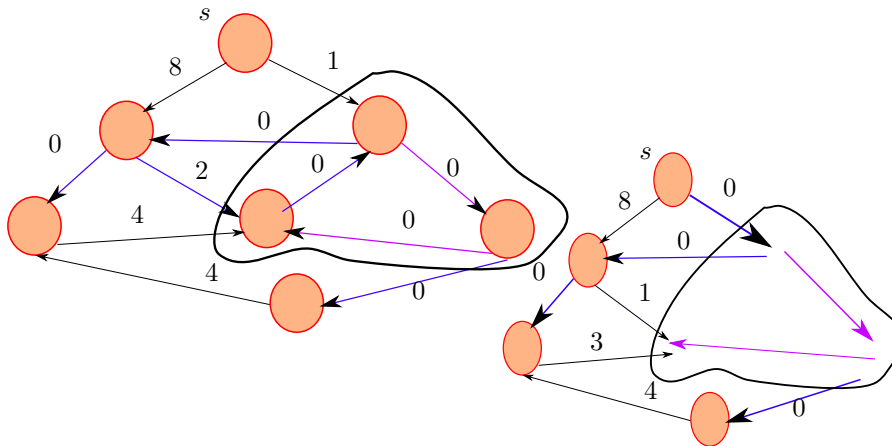


Рис. 1.18: orgraph

Утверждение 7 (Алгоритм Эдмундса). Выберем v . Рассмотрим все входящие рёбра. Если вычесть из всех одинаковое Δ , то минимум не поменяется. Такая безопасная операция. Если вычитать веса минимальных получится много нулей, что круто.

```

1  for v = 0 .. n-1
2      Delta = min w_{uv}
3      for uv:
4          w_{uv} -= Delta
5

```

Возьмём вершины недостижимую из S . Не можем дойти до s , где-то зациклимся.

```

1  if все вершины достижимы из S по нелевым рёбрам:
2      строим дерево из 0, радуемся
3  if нет (else):
4      найдём цикл, сожмём цикл и запустимся рекурсивно
5

```

$$O(nm)$$

Хотим быстрее.

Умеют за $O(m + n \log n)$, быстрее вроде не могут, доказывать, что быстрее нельзя тож))

Утверждение 8 (Как делать нормально (махание руками в качестве бонуса)). Возьмём любую вершину v . Возьмём все входящие рёбра, вычтем минимум, получим нулевое ребро, пойдём по нему. Повторим, будем двигаться. Нужен минимум и вычитание из отрезка – дерево отрезков.

Если мы дошли до s , построили большой кусок дерева, радость, можно брать другую ветку.

Более грустная ситуация – зациклились. Цикл нужно сжать в одну большую вершину. А дальше продолжить процесс от неё.

нужна структура данных: минимум, вычитание из отрезка, и мёрж. Sprlay дерево сойдёт)

Было в цикле k вершин. за $k \log n$ смёрджили, уменьшили к-во вершин на k . Всего соответственно мёрджили не больше, чем $n \log n$

Поиск минимумов – $n \log n$

Где же m ?.. где-то обманули. На самом деле при сжатии цикла появляется дофига петель (рёбер между вершинами цикла). Эти петли нужно отсеивать. Таки $m \log n$.

Класть себе петли плохо. Храним минимальное ребро среди нужных и поддерживаем, что в фиббоначиевой куче лежат правильные рбра. Нужна операция перекладывания из одной фиббоначиевой кучи в другую за единицу. Мы так не можем (следствие – сортировка за 1), но если очень-очень хотим, то можем. Там некоторая магия с тем, что какие-то кучи валидные, а какие-то нет и в целом всё работает..?..