Алгоритмы и Структуры Данных

Коченюк Анатолий

2 декабря 2021 г.

Глава 1

Алгоритмы на графах и строках и фане.

1.1 Графы и обход в ширину

кружочки и стрелочки

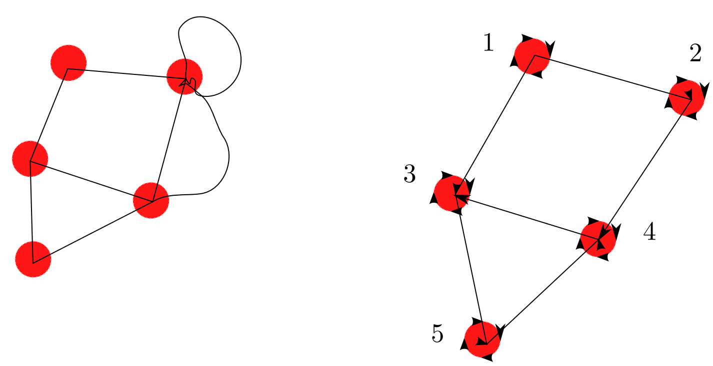

Рис. 1.1: fex

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 петлами

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

```
dfs(v):
    mark[v] = True
    for vu \in out(v):
        if !mark[u]:
            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. ■

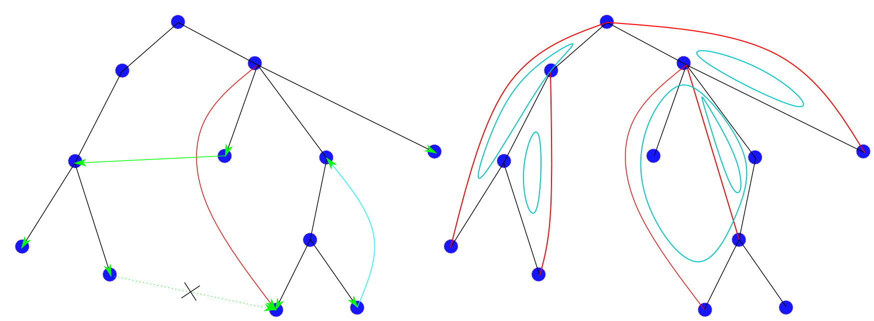

Рис. 1.2: dfstree

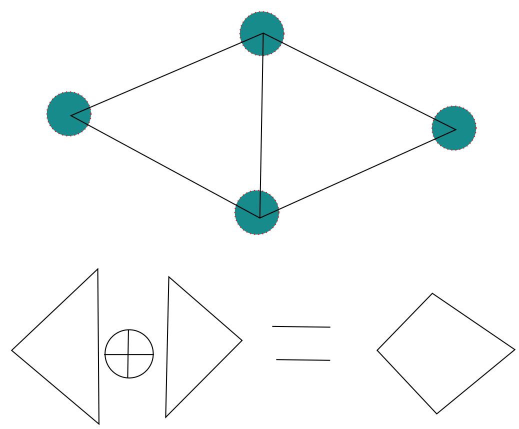

Рис. 1.3: xor

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.

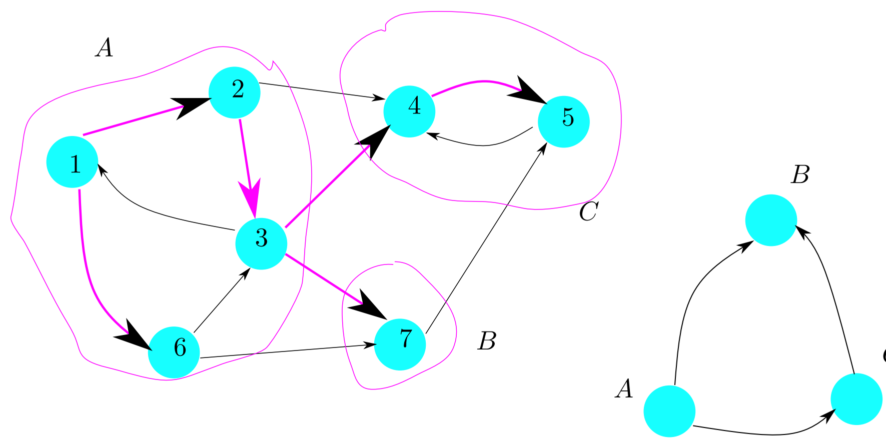

Рис. 1.4: dvureb

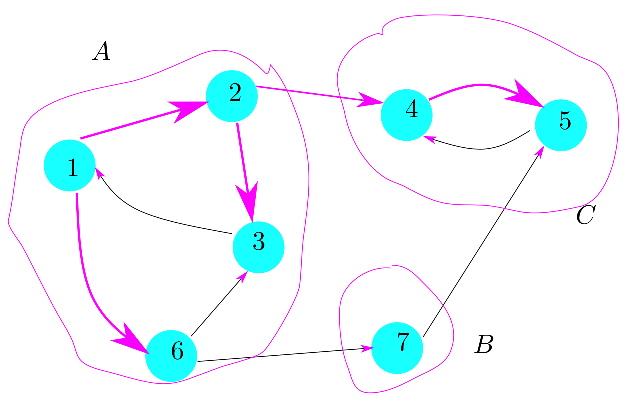

Рис. 1.5: neprimer

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 компоменты 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

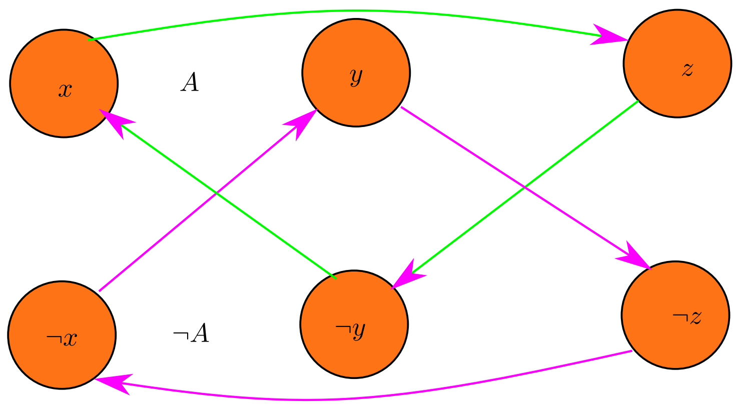

Рис. 1.6: vershinki

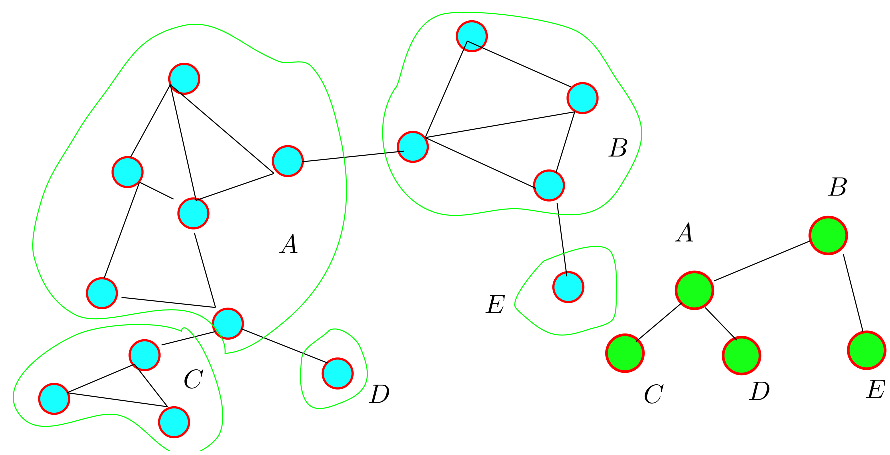

Рис. 1.7: лфлщштшигвкфзр

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]:
```

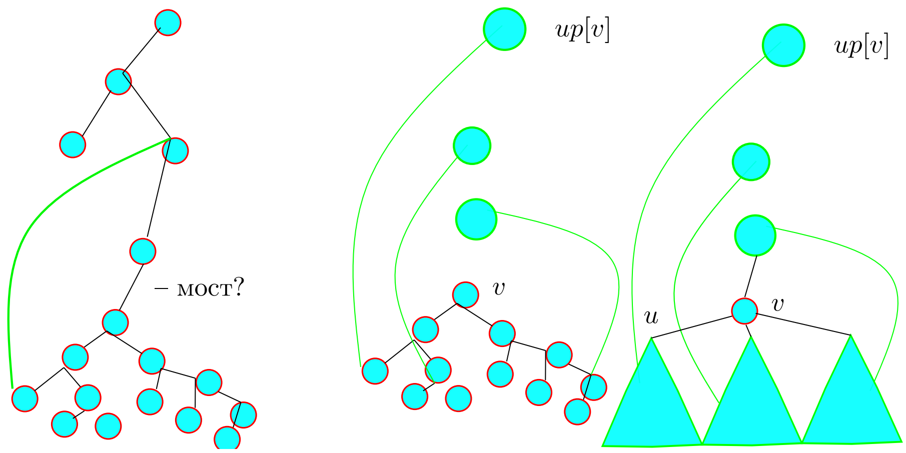

Рис. 1.8: sadfs

```
        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])
```

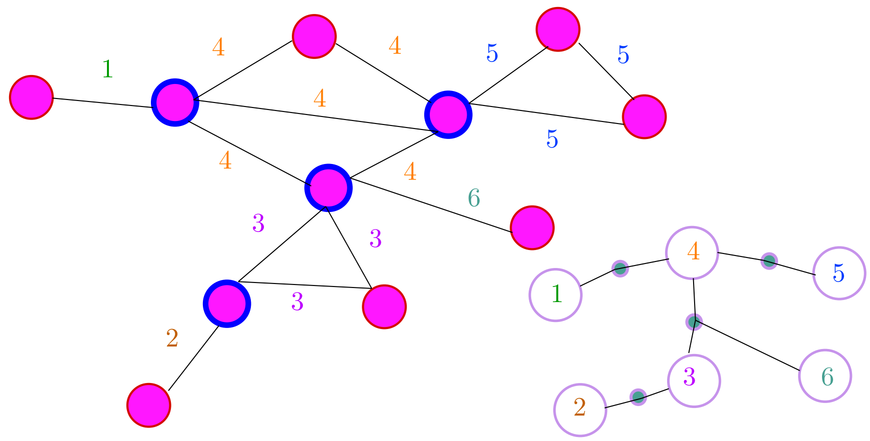

Рис. 1.9: лфкештлгрщсргтфкшыщмфе

```
                if up[u] >= t_in[v]:
                    ok = True
            else:
                up[v] = min(up[v], t_in[u])

        if ok || p = None && c > 1:
            v - точка сочленения
```

```
    dfs(v, p):
        t_in[v] = T++
        up[v] = t_in[v]
        mark[u] = True
        ok = False
        for u in out[v]:
            if u = p:
                continue
            if !mark[u]:
                buf.push(vu)
                dfs(u,v)
                up[v] = min(up[v], up[u])
                if up[u] >= t_in[v]
                    while True:
                        e = buf.pop()
                        comp.add(e)
                        if r = (vu):
                            break
            else:
                up[v] = min(up[v], t_in[u])
                if t_in[u] < t_in[v]:
                    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..

```
dfs(v):
    vu -- любое непомеченное ребро из v
    if vu = None:
        return
     пометить vu
     dfs(u)
     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

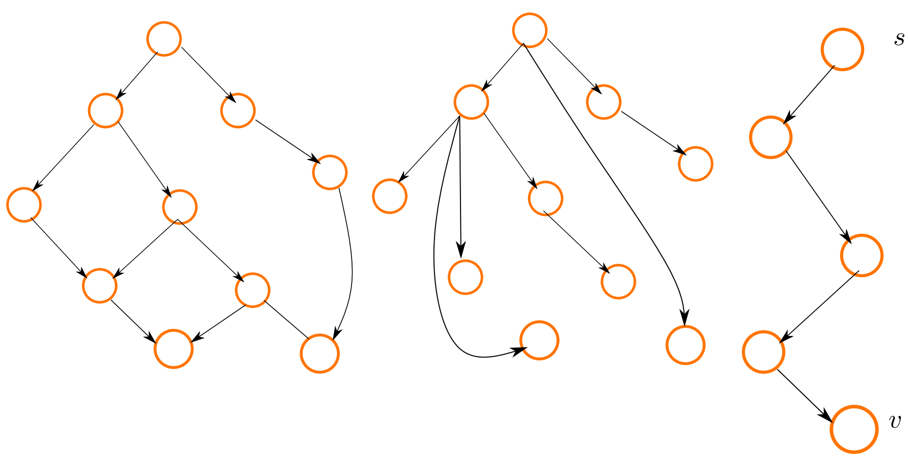

Рис. 1.10: krasiv

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

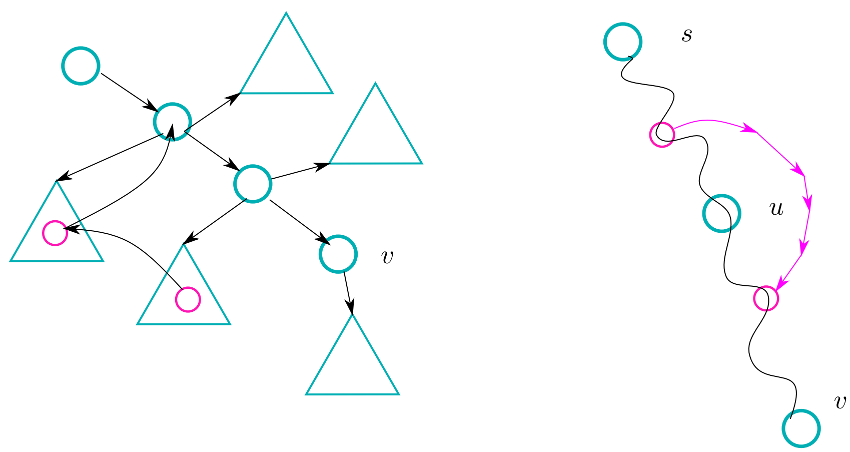

Рис. 1.11: cicles

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 $> v$

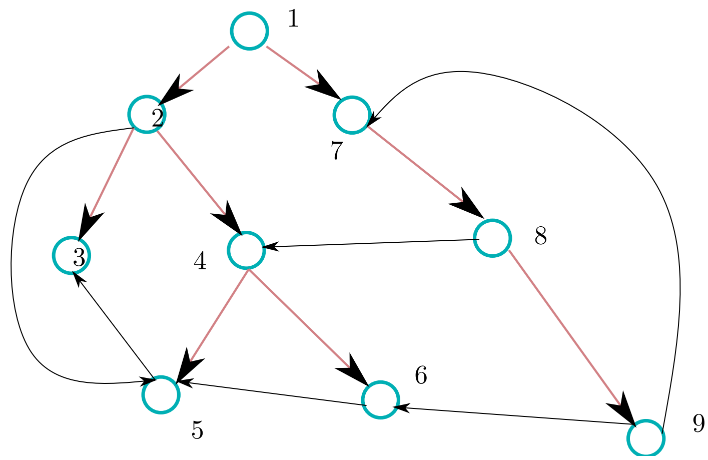

Рис. 1.12: semi

$sdom[v]$ – 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
```

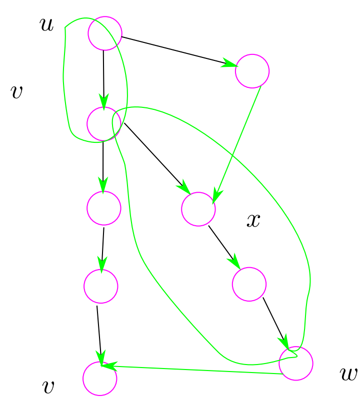

Рис. 1.13: semi2

1.6 Минимальное остовное дерево

Хотим выбрать остовное дерево минимальное по сумме весов рёбер.

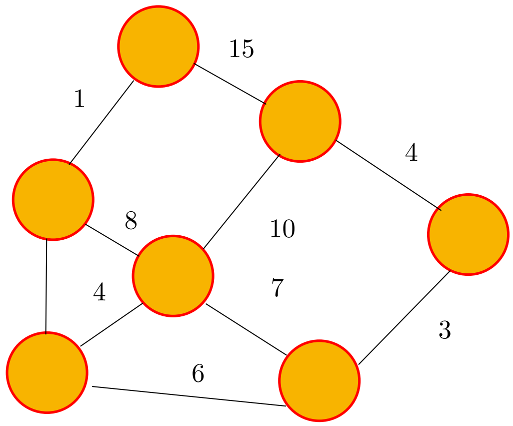

Рис. 1.14: ostree

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 $\min_{\substack{u \in A \\ v \in B}} \implies uv \in MST$ – minimal spanning tree.

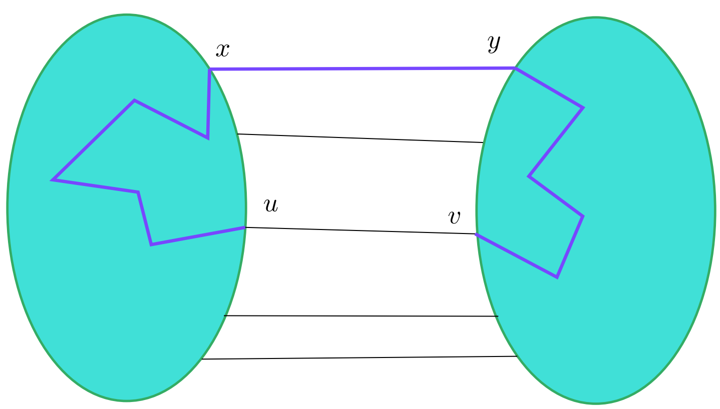

Рис. 1.15: spliceproof

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. А 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 резрезе, добавляем его.

```
    for uv in E.sorted():
        if find(u) != find(v):
            T = T + uv
            union(u,v)

```

Утверждение 5 (Алгоритм Прима)**.**

```
    A = {s}
    repeat(n-1)
        uv -- min: u in A, v not in A
        A = A U {v}
        T = T U {uv}

```

```
repeat(n-1)
    uv = Q.remove_min()
    A = A U {v}
    T = T U {uv}
    for uv:
        if w in A:
            Q.remove(vw)
        else
            Q.add(vw)
```

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

$O(m \log n)$

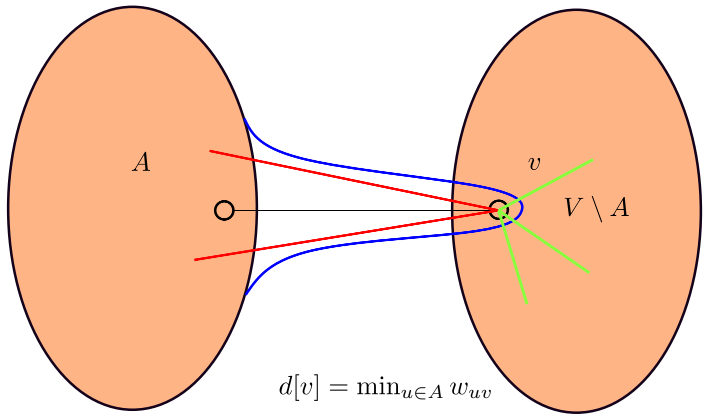

Рис. 1.16: prima

В полном графе лучше использовать массив для $n^2 + m$. Можно Фиббоначиеву 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.

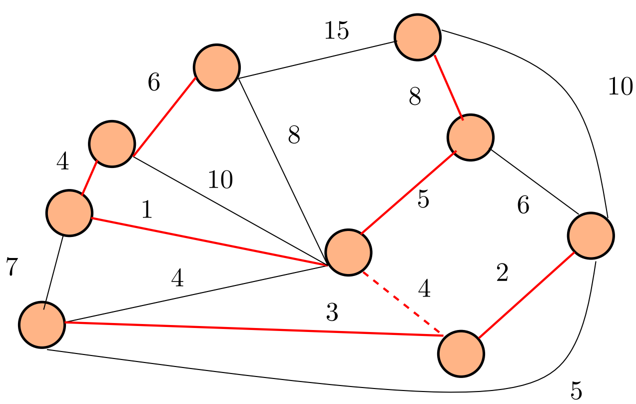

Рис. 1.17: barufka

Помечаем для каждой вершины её минимальное ребро. ...

Возьмём ориентированный граф

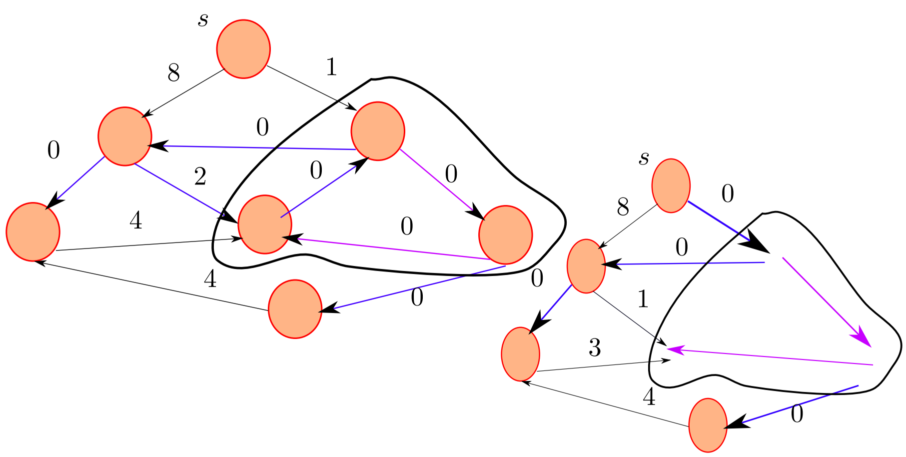

Рис. 1.18: orgraph

> **Утверждение 7** (Алгоритм Эдмундса)**.** Выберем v. Рассмотрим все входящие рёбра. Если вычесть из всех одинковое Δ, то миниму не поменяется. Такая безопасная операция. Еслив вычитать веса минимальных получится много нулей, что крута.

```
            for v = 0 .. n-1
                Delta = min w_{uv}
                for uv:
                    w_{uv} -= Delta
```

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

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

$O\left(nm\right)$

Хотим быстрее.

Умеют за $O\left(m + n\log n\right)$, 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 большуую вершину. А дальше продолжить процесс от неё.

нужна структура данных: минимум, вычитание из отрезка, и мёрж. Splay 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 фиббоначивое й 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 вцелом всё работает..?..

1.7 Кратчайшие пути

Есть граф, хотим дойти из одной вершины в другую кратчайшим путём.

1. $w_{uv} = 1$
2. $w_{uv} \geqslant 0$
3. w_{uv}

Решим для ориентированного. Неориентированный – ориентированный с рёбрами в обе стороны.

```
bd[0] = s
d[s] = 0 // d[v] = -1, v != s
for k = 0 .. n-2: // k -> k+1
    for v in bd[k]
        for all vu:
            if d[u] = -1:
                d[u] = k+1
                bd[k+1].add(u)
```

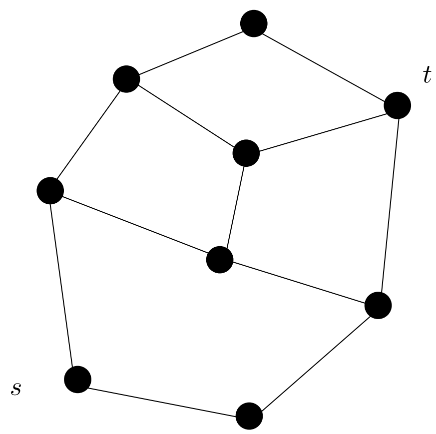

Рис. 1.19: short

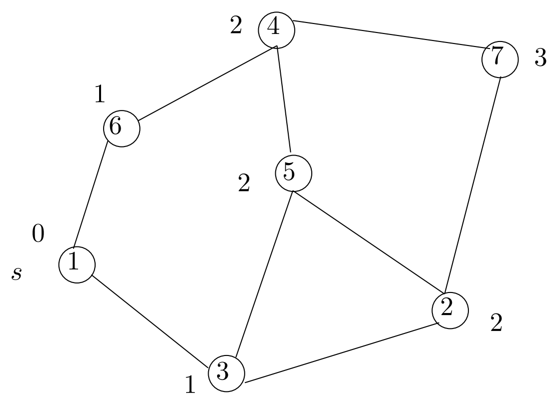

Рис. 1.20: shortex

$O(m)$

Вместо списка списков можно использовать одну очередь.

```
    q.add(s)
    d[s] = 0 // d[v] = -1, v != s
    while !q.empty():
        v = q.remove()
            for allvu:
                if d[u] = -1:
                    d[u] = d[v] + 1
```

```
                    q.add(u)
    for k = 0 .. n-2: // k -> k+1
        for v in bd[k]
            for all vu:
                if d[u] = -1:
                    d[u] = k+1
                    bd[k+1].add(u)
```

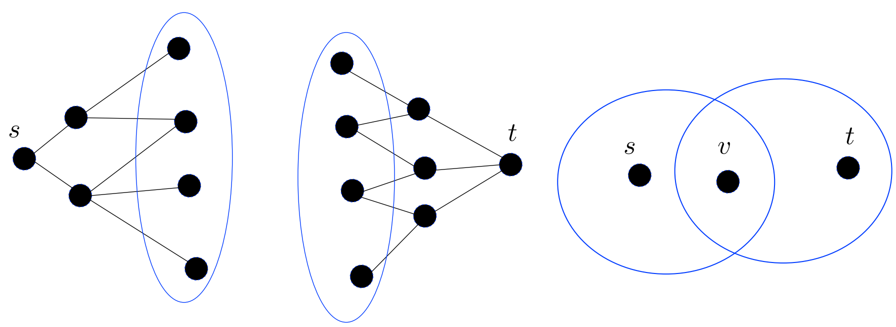

Рис. 1.21: both sides

Можно идти с двух сторонинайти вершину, где мы встретимся, так можно тоже найти кратчайший путь

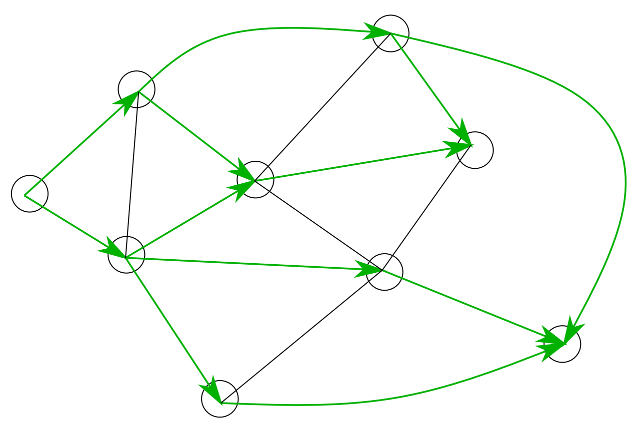

Рис. 1.22: poslozhnee

Можно построить граф, куда мы будем добавлять ребро, если при нм увеличивается расстояние. Он ацикличен и любой путь в нём кратчайший. Друг человека.

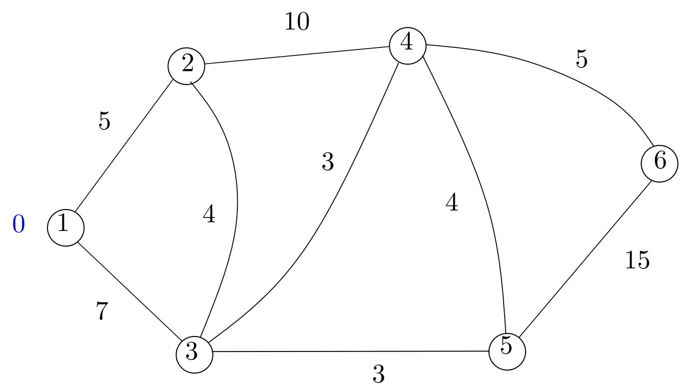

Рис. 1.23: kak len'

```
A = {s}, d[s] = 0
repeat (n-1)
    v: v not in A, d[u] + w_{uv} -> min
    d[v] = d[u] + w_{uv}
    A = A U {v}
```

Инвариант: $v \in A \implies d[v] = dist(s, v)$

```
v not in A => d[v] = min(d[u] + w_{uv})
d[s] = 0, A = empty
for i = 0 .. n-1:
    PQ.add(i)
while !PQ.empty():
    v = PQ.remove_min()
    A = A U {v}
    for all vu:
        d[u] = min(d[u], d[v] + w_{vu})
        PQ.update(u)
```

Двоичная куча – $O(m \log n)$

Массив – $O(n^2 + m)$

Фиббоначиева куча – $O(n \log n + m)$

Утверждение 9 (A*)**.** $\rho(u, v) \leqslant dist(v, u)$

$\rho(u, t) \leqslant \rho(v, t) + w_{vu}$

$h[v] = \rho(v, t)$

$\min d[v] \to \min\left(d[v] + h[v]\right)$

```
w'_{vu} = w_{vu} + h[v] - h[u]
```

Утверждение 10. Кратчайший путь не содержит циклов (если все циклы неотрицательные).

Утверждение 11 (Форда-Беллмана)**.** Д.П.

d[v,k] – кр. путь из S в v, состоит из k рёбер

$d[v, 0] = \begin{cases} 0, v = s \\ +\infty m v \neq s \end{cases} \qquad d[v, k] = \min\left(d[u, k-1] + \omega_{uv}\right)$

```
for  k = 1 .. n-1:
    for v = 0 .. n-1:
        d[v,k] = min(...)

```

По времени плохо $O(nm)$. По памяти можно хранить только предыдущую строку, а для минимума считать $d[v, k]$ – из k или меньше рёбер.

```
d[s] = 0, d[v] = infinity forall v != u

for k  0 ... n-1
    for uv in E:
        d[v] = min(d[v], d[u] + w_{uv})
```

На k-ой итерации $d[v]$ - длина какого-то пути $\leqslant$ кр пути из $\leqslant k$ рёбер.

Если на какой-то итерации ничего не срелаксировалось, можно сделать брейк.

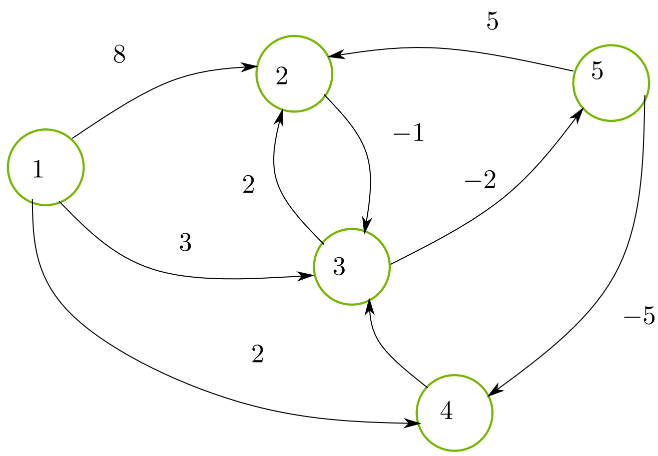

Рис. 1.24: ford-bellman

Утверждение 12. $d[ij]$ – кратчайший путь между i и j, такой что все вершины на этом пути имеют номера $\leqslant k$

$$d[ij] = \begin{cases} 0 & i = j \\ \omega_{ij} & \text{есть ребро} \\ \infty & \text{нет ребра} \end{cases}$$

1. Вершины k нет, тогда $d[ij]$ при $k-1$
2. Вершина k есть, тогда $d[ik] + d[kj]$

Как понять, а не дали ли нам случайно отрицательный цикл? на диагонали появятся отрицательные числа

Если нет неотрицательных циклов $\implies$ можно подобрать $\varphi(v)$ $\quad \omega'_{uv} = \omega_{uv} + \varphi(u) - \varphi(v) \quad \omega'_{uv}$

Это преобразование сохраняет кратчайшесть пути.

$\sum w' = \sum w + ((\varphi_1 - \varphi_2) + (\varphi_2 - \varphi_3) + \ldots) = \sum w + \varphi(s) - \varphi(t)$

На самом деле отсутствие отриц. циклов эквивалентно тому, что мы такие можем подобрать.

1.8 Строки и алгоритмы

Определение 8. Строка – набор из буковок

$sLabaabcab$

Буквы из Σ

Буквы строки индексируются $s[i]$ – i-ая буква

Можно взять префикс или суффиксс строки $s[0..i-1] \quad s[i..n-1]$

Подстрока $s[l..r-1]$

1.8.1 Поиск подстроки

T – большая строка. s - маленькая строка. Ищем подстроку в T равную s

```
for i = 0 .. n-m:
    if T[i .. i+m-1] = S
        print(i)

O(n*m)
```

Теорема 1 (Алгоритм Рабина-Карпа)**.** Хотим найти одно вхождение. Большинство времени тратим на проверку, что две строки неравны

```
for i = 0 .. n-m:
    if hash(T[i .. i+m-1]) != has(S):
        continue
    if T[i .. i+m-1] = S:
        print(i)
        break

```

Если научимся быстро считать хэши, то наступит счастье.

$hash(s) = (s_0 x^{n-1} + s_1 x^{n-2} + \ldots + s_{n-1} x^0)\%M$

M – случайное большое простое. x – случайное

$P(hash(A) = hash(B)) \quad A \neq B \quad \leqslant \frac{n}{M}$

$\sum a_i x^i = \sum b_i x^i \quad \sum (a_i - b_i) x^i = 0 (mod\ M)$

x – корень многочлена, таких максимум n

$H = T_i x^{m-1} + T_{i+1} x^{m-2} + \ldots + T_{i+m-1} x^0$

$H' = Y_{i+1} x^{m-1} + \ldots + T_{i+m-1} x^1 + T_{i+m} x^0$

$H' = H * x - T_i x^m + T_{i+m}$

```
H = hash(T[0 .. m-1])
Hs = hash(S)
for i = 0 .. n-m:
    if H = Hs:
```

```
            if T[i .. i+m-1] = S:
                print i
                break
        H = (H*x - T[i]x^m + T[i+m]) % M
```

Все вычисления везде по модулю.

```
hash(l,r) = hash(S[l .. r-1]) -- O(1) хочется
P[i] -- полином для iго- префикса
P[i] = hash(S[0 .. i-1]) = s_0x^{i-1} + ... + s_{i-1}x^0
P[i] = P[i-1]*x + s_{i-1}

hash(S[l..r-1]) = s_l * x^{r-l-1} + .. + s_{r-1}x^0
P[r] = s_0x^{r-1} + .. + s_{r-1}x^0
hash(l,r) = P[r] - P[l] * x^{l-r}
```

Теорема 2 (Кнута-Морриса-Пратта)**.** Есть текст. Идём слева направо, ищем строчку s. Дошли до i-ой позиции, встретили сколько-то символов из начала строки s. Встречаем новый символ. Если такой же в s – радуемся. Если нет, то я явно столько символов s мы не нашли, мы нашли меньше и надо найти сколько.

Префикс-функция. $pref(s)$ – максимальная строка, которая является и префиксом и суффиксом и при этом меньше всей строки.

```
P[i] = pref(S[0 .. i-1])

S = abbababb
P =-00012123
```

```
p[0] = 1
for i = 0 .. n-1:
    k = p[i-1]
    while k != -1 and s[k] != s[i-1]:
        k = p[k]
    p[i] = k+1

O(n) -- n раз можем понизить значение k
```

Приписываем s к T слева и дописываем между ними левый символ. Ищем префикс функцию в ней и если она равна длине строки s, то мы нашли вхождение.

Теорема 3. $z[i]$ – максимальное k, что $s[0..k-1] = s[i..i+k-1]$

```
for i = 1 .. n-1:
    z[i] = 0
    while s[i+z[i]] = s[z[i]]:
        z[i]++
```

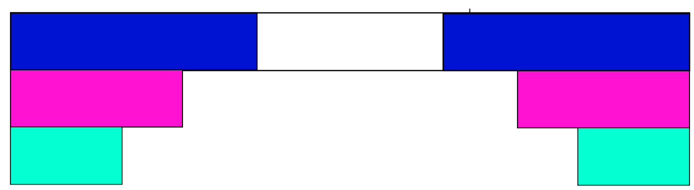

Рис. 1.25: prefpref

```
    for i = 1 .. n-1:
        if i > 1:
            z[i] = min(z[i-l], l + z[l]-i)
            z[i] = max(z[i], 0)
        while s[z[i] = s[i+z[i]]:
            z[i]++
        if i = 1 or i + z[i] > l + z[l]:
            l = i

        z[i] = min(z[i-l], l + )
        while s[i+z[i] = s[z[i]:
            z[i]++

O(n) -- в вайле увеличиваем сумму l + z[l]
```

1.9 Минимизация автоматов

Количество допускающихся строк можно посчитать динамикой примерно как в Форде-Беллмане.

```
    for i = 1 .. n:
        for c in Sigma:
            k = i
            while k >= 0 and s[k] != c:
                k = p[k]
            delta[i,c]=k+1
```

Но это медленно

```
    for i = 1 .. n:
        p[i] = delta[p[i-1], s[i-1]
```

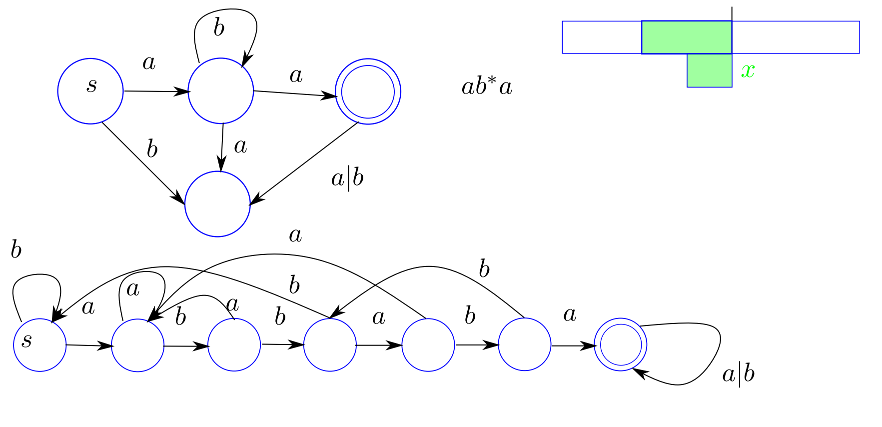

Рис. 1.26: aluto1

```
        for c in Sigma:
            if c = s [i]:
                delta[i,c] = i + 1
            else:
                delta[i,c] = delta[p[i], c]

    O(n*|E|)
```

> **Задача 6.** Хочется построить автомат, чтобы он принимал только конкретный заданный набор строк

Минимизация ДКА

```
    for u in T:
        for v not in T:
            Q.add(u,v)
            A[u,v] = 'x'

    while !Q.empty():
        (u,v) = Q.remove()
        for c in Sigma:
            for u' : delta[u', c] = u:
                for v' : delta[v', c] = v:
                    if A[u', v'] != 'x':
                        A[u', v'] = 'x':
                        Q.add(u'v')

        O(n^2 * Sigma)
```

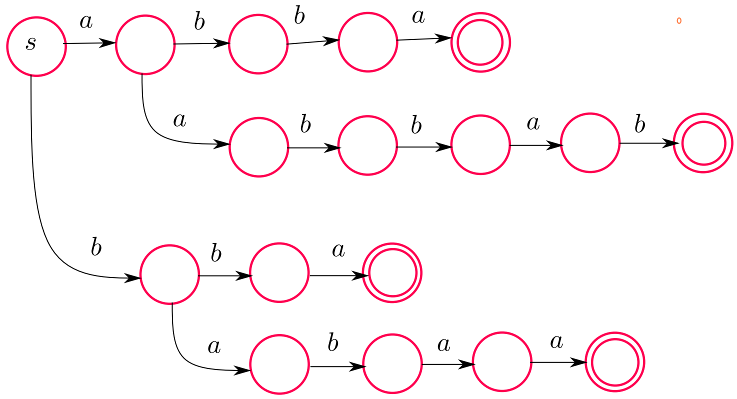

Рис. 1.27: aluto

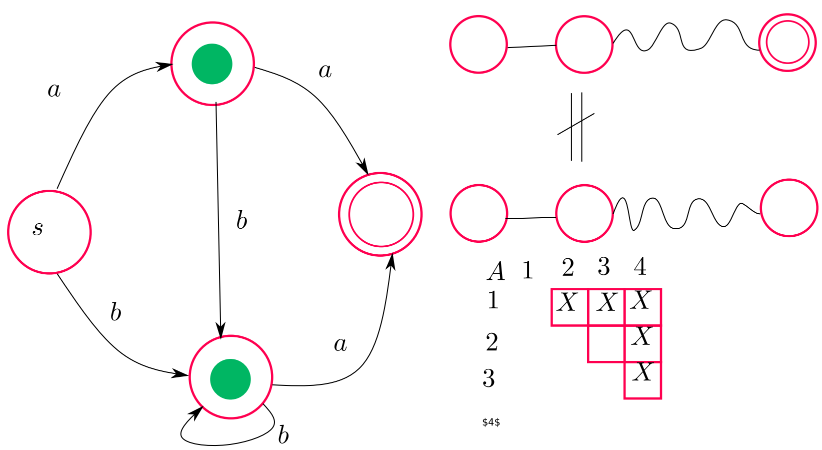

Рис. 1.28: aluto3

Задача 7. Проверить эквивалентность ДКА

Возьмём два автомата и будем считать, что один. Несвязный с двумя стартовым состояниями, но неважно Запускаем алгоритм сверху и проверяем эквивалентный ли начальные состояния.

zzz...

```
    Q.add(T)
    Q.addне( T)
```

```
    while !Q.empty:
        A = Q.remove()
        for c in Sigma:
            for c in A:
                for u: delta[u,c] = v
                    B = set[u]
                    S.insert(B)
                    M[B].add(u)
            for B in S:
                B1 = M[B], B2 = B \ B1
                if B1 = empty || B2 = empty: continue
                if B in Q:
                    заменить B на B1 и B2
                else:
                    Q.add(min(B1, B2))
```

1.10 Ахо-Корсик

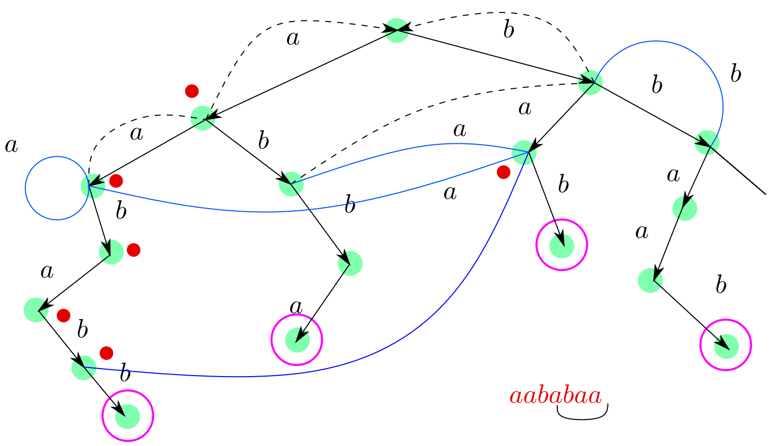

Рис. 1.29: karas

$suflink[v]$ = ссылка на max суффикс V, который есть в боре

$\delta[v, c]$ – переход из v по символу c

```
    for v in BFS:
        suflink[v] = delta[suflink[parent[v]], char(v)]
        for c in Sigma:
            if delta[v,c] = null:
                delta[v,c] = delta[suflikn[v], c]
```

Длина $suflink$ от v всегда меньше, чем v. Для динамики надо, чтобы родитель и его $suflink$ посчитались, чем вершина. Для этого запустим bfs и будем идти по нему

$O\left(\left(\sum S_i\right) \cdot |\Sigma|\right)$

Хотим проверить если ли в тексте одно из слов из конкретного набора (ругательства например). Помечаем терминальными вершины, из которых суффиксная ссылка ведёт в терминальное.

Замечание. Суффиксные ссылки образуют дерево. Из каждой вершины есть ссылка на вершину меньшего размера

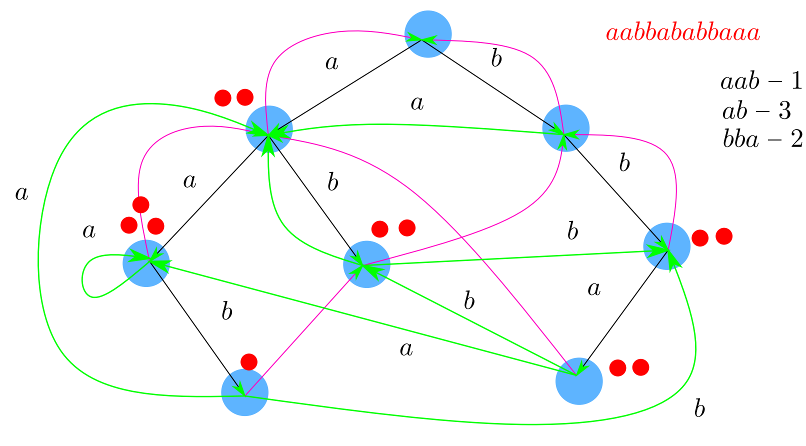

Рис. 1.30: karasik

А как делать без построения автомата (он большой, а хочется наверное просто suflink и хватит)

```
for v in BFS:
    k = suflink[parent[v]]
    while k != null && delta[k,v] != null:
        k = subflink[k]

    if k = null:
        suflink[v] = root
    else
        suflink[v] = delta[k,x]
```

На каждой итерации фора длина строки увеличивается, а в вайле уменьшается. Увеличивается она длину строки раз, значит и уменьшается не больше, чем столько $O\left(|S_i|\right)$

1.11 Суффиксный массив

$s = abbaab$

Выпишем суффиксы в лексикографическом порядке – p

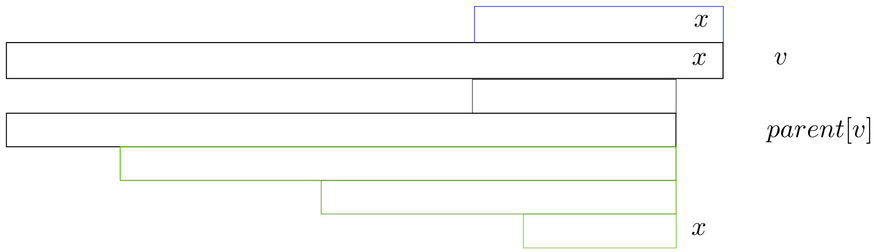

Рис. 1.31: noauto

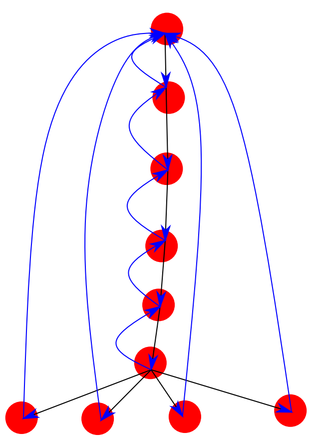

Рис. 1.32: break

$s[p[i]+j]$ – j символ i суффикса

1.11.1 Зачем

Искать в s подстроку T_i. Любая подстрока это префикс какого-то суффикса. Найдём в суффиксном массиве бинпоиском нужную подстроку.

1.11.2 Как

Таблица 1.1

6 ·
3 aab 4 ab 0 abbaab 5 b 2 baab 1 bbaab

Утверждение 13 (Простая идея)**.** Взять все суффиксы и отсортировать. $O(n \log n \cdot |comp|)$

Худший случай $O\left(n^2 \log n\right)$

На практике нормальный метод, можно жить, если достаточно рандомные строки.

Задача – научиться нормально сравнивать суффиксы. Нужно найти общий префикс и сравнить следующие два символа.

С хэшами $O\left(n \log^2 n\right)$

Сортировать строки неприятно. В конце строки ещё надо ифать случай, что строка заканчивается. Пусть каждая строка заканчивается долларом, который меньше, чем любой символ. По техническим причинам после доллара есть смысл писать дальше строку по циклу. А потом ещё запишем ещё один символов по циклу, чтобы довести до 2^n

Символы после доллара не влияют на порядок, потому на первом же сравнение оканчивается.

Таблица 1.2: caption

$abbaab$ aab$abba ab$abbaa abbaab$a b$abbaab baab$abb bbaab$ab

На итерации k сортируем строки $s[i..i + 2^k - 1]$ (цикл)

1. $k = 0 \quad s[i..i]$ – $O(n \log n)$ любой сортировкой
2. $k \to k + 1$ $A < B \iff A_1 < B_1 \vee (A_1 = B_1 \wedge A_2 < B_2)$

   Осталось научиться сравнивать A_1 и B_1

Таблица 1.3: ex

$	6	0	ab	0	12	01	6	0
a	0	1	bb	1	22	11	3	1
a	3	1	ba	2	21	12	0	2
a	4	1	aa	3	11	12	4	2
b	1	2	ab	4	12	20	5	3
b	2	2	b$	5	20	21	2	4
b	5	2	$a	6	01	22	1	5
	p	c					p	c

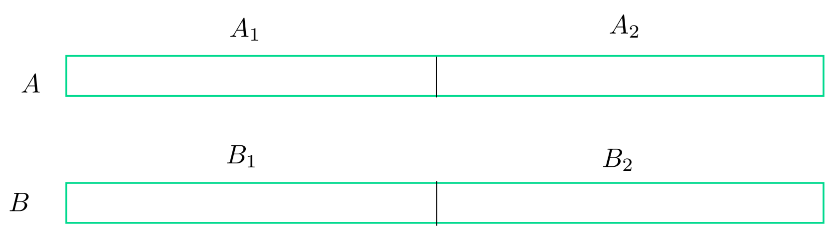

Рис. 1.33: k-k+1

$O\left(\log n \cdot n \log n\right) = O\left(n \log^2 n\right)$

Но в моменте, когда мы сравниваем пары целых чисел, можно использовать radix sort.

$O\left(\log n \cdot n\right) = O\left(n \log n\right)$

Вместо сортировки можно получать список отсортированный по левой половине сразу. Для этого к отсортированным строкам 2^k допишем по циклу следующую половину

```
while 2^k < n:
    for i = 0 .. n-1:
        p[i] = (p[i] - 2^k)%n
    count_sort(p,c)
    c'[p[0]=0
    for i  = 1 .. n-1:
        if c[p[i]] = c[p[i-1]] and c[p[i]] + 2^k] = c[p[i-1] + 2^k]:
            c'[p[i]] = c'[p[i-1]]
        else:
            c'[p[i]] = c'[p[p-1]]+1
    c = '
    k++
```

$lcp(i, j) = lcp(s[i..n], s[j..n])$

$lcp[i] = lcp\left(p[i], p[i+1]\right)$

$lcp(i, j) = \min_{k=c[i]..c[j]-1} l[k]$

как искать $l[i]$

```
fori = 0 .. n-1:
```

```
        x = c[i] -- позция в суф. мас.
        if k > 0:
            k--
        while s[p[i-1]+k] = s[p[x]+k]:
            k++
        l[x-1] = k
```