Алгоритмы и структуры данных, лекции

Глава 1

Паросочетания и потоки

Мы начнём с паросочетаний и потоков.

1.1 Паросочетания

Паросочетание графа — это набор его ребер, не имеющих общих вершин.

1.1.1 Паросочетание в двудольном графе

Паросочетания в двудольных графах ищутся гораздо проще, чем в произвольных. Алгоритмы решают эту задачку понемногу. Берется маленькое (пустое) паросочетание и расширяется с помощью дополняющих цепочек.

Дополняющая цепочка — это путь, который начинается и заканчивается в вершинах вне паросочетания, и ребра в нём чередуются (принадлежит - не принадлежит пар. сочетанию). Из такой цепочки можно получить паросочетание большего размера (на 1).

Алгоритм начинает строить цепочки и удаляет дополняющие цепочки такой заменой.

Теорема 1. В графе нет дополняющего пути тогда и только тогда, когда паросочетание максимально.

Это утверждение сформулировано для произвольных графов: для двудольных и не очень. А в чём проблема? Проблема в поиске дополняющих путей.

В двудольном графе это делается просто. Это алгоритм Куна.

Двудольный граф можно хранить не как нормальный граф. "Алгоритм КУНА. Это не связано с аниме никак!" "Код, на самом деле, за пять копеек." Асимптотика: $O(nm)$.

Алгоритм никогда не освобождает вершины — если вершшина попала в пар. соч., она там останется, можно dfs из неё не запускать. Чуть сложнее: если dfs однажда не нашел доп. пути из одной вершины, он никогда его не найдёт — можно его не запускать.

Немного о полных сочетаниях.

Теорема 2. В двудольном графе $G_{n,n}$ существует полное паросочетание в том и только том случае, когда для любого подмножества A вершин одной доли выполнено $|N(A)| \geqslant |A|$.

Паросочетания позволяют решать кучу прикольных задач.

1.1.2 Вершинное покрытие

Вершинное покрытие в графе — это набор вершин, таких, что никаие две вершины не лежат на одном ребре. Это понятие двойственное понятию паросочетания. Мы ищем минимальное по мощности вершинное покрытие. Это NP-полная задача в произвольном графе. В двудольном, однако, всё очень мило.

Размер покрытия в графе всегда не превосходит размер вершинного покрытия: $M \leqslant S$.

1.1.3 Парасочетание в недвудольном графе

Так же понемногу будем искать дополняющие пути и достраивать наше паросочетание.

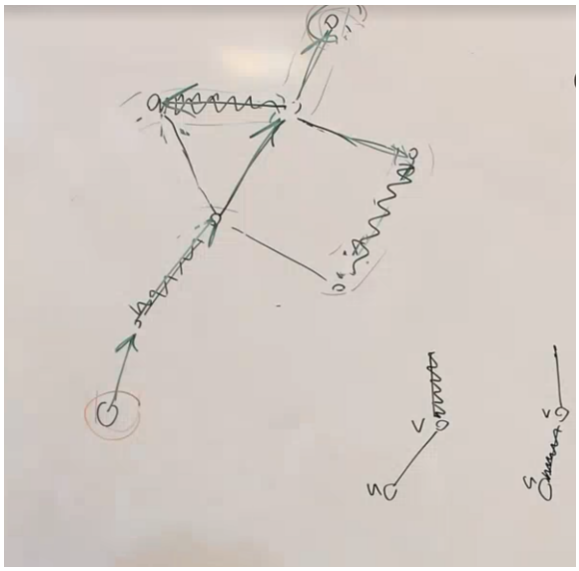
Как уже было доказано, M — max парсоч $\iff$ не. т дополняющих путей.

Как искать дополняющие пути? Может быть заюзать `dfs`?

```
dfs(v, state)
    ...
for n
    if state:
        (u,v) ∈ M
    else:
        (u,v) ∈ M
    dfs(n, !state)
```

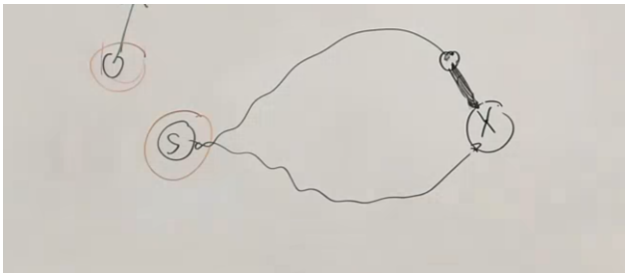
Но так нельзя!

На таком графе не сработает:



Проблема в том, что в какую-то вершину мы можем прийти с неправильной четностью.

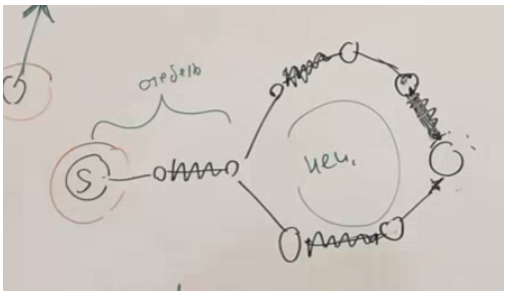
Утверждение: если такого не случилось, то все будет ок.
Пусть есть стартовая вершина. Назовем вершину сомнительной, если до нее есть четный и нечетный пути.
Если в графе нет сомнительных вершин, такой граф можно переделать в двудольный и легко в нем постить паросочетание.



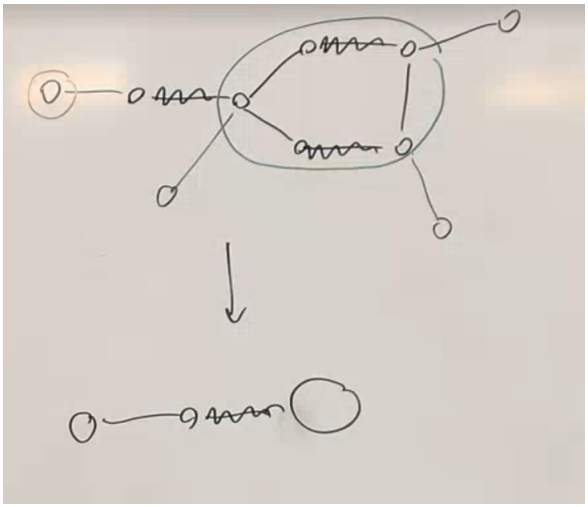
Что делать? Давайте запустим такой dfs. Может быть три случая:

1. Нашли дополняющий путь;
2. Не нашли дополняющий путь и нет сомнительных вершин $\Longrightarrow$ паросочетание макисмальное;
3. Нашли сомнительную вершину.

Если случилось третье, будем веселиться. При помощи того же dfs-а найдем соцветие (blossom). Внутри соцветия цикл нечетной длинны.



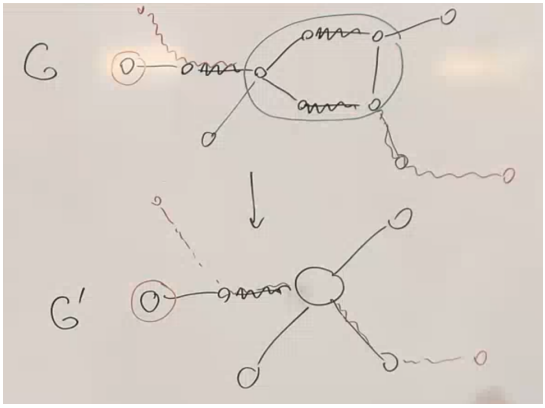
Алгоритм называется blossom cut. Возьмем все вершины соцветия и заменим на большую вершину.



Утверждение: если в исходном графе G был дополняющий путь, то и в полученном графе G'.

Доказательство. Докажем в две стороны

($\Leftarrow$) Либо дополняющий путь вообще не проходит через большую вершину соцветия, тогда вообще все просто, ничего точно не поломается.
Если дополгяющий путь проходит через большую вершину соцветия, то выберем кусок цикла нужной четности и соееденим кусочки дополняющего пути.



(⇒) Все сложно. Могут быть всякие сложные пути и после сжатия не понятно, что с ними делать. Конструктивно доказать сложно.

Докажем при помощи теоремы о дополняющем пути. От противного.

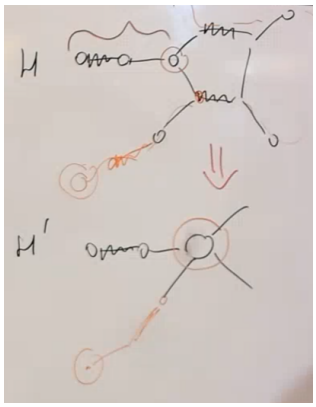
1. Возьмем граф G вместе с соцветием и инвертируем ему стебель, получим граф H с валидным парсочем.

2. Возьмем граф H вместе с соцветием и инвертируем ему стебель, получим граф H' с валидным парсочем.

3. Если в G есть дополняющий путь, то и в G' тоже есть дополняющий путь (очевидно, так как и то и другое паросочетание не максимального размера).

   Если в H' есть дополняющий путь, то и в H тоже есть дополняющий путь (очевидно, так как и то и другое паросочетание не максимального размера).

   Если в G' есть дополняющий путь, то и в H' есть дополняющий путь (конструктивно).



■

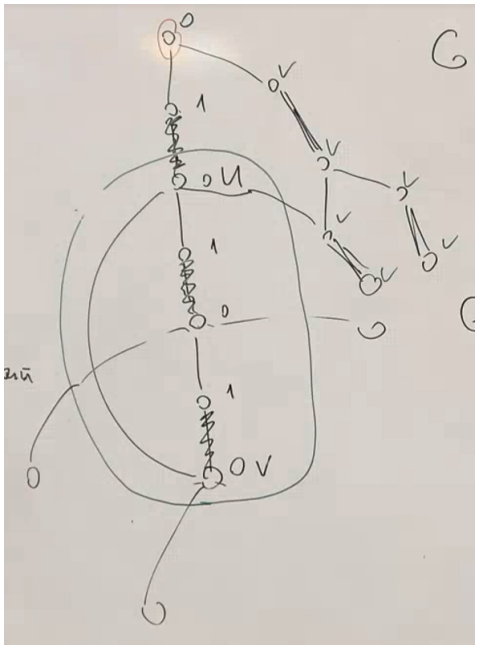
Алгоритм

```
M = ∅
while True:
    dfs
    if нашли доп путь:
        разжать соцветие
        M++
        continue
    if нет додоп пути, не соцветий:
        break
    if соцветие:
        сжать соцветие
```

Итого, время работы алгоритма — $\mathcal{O}(n^2m)$.

Как реализовать побыстрее? *Немного пострадать.*

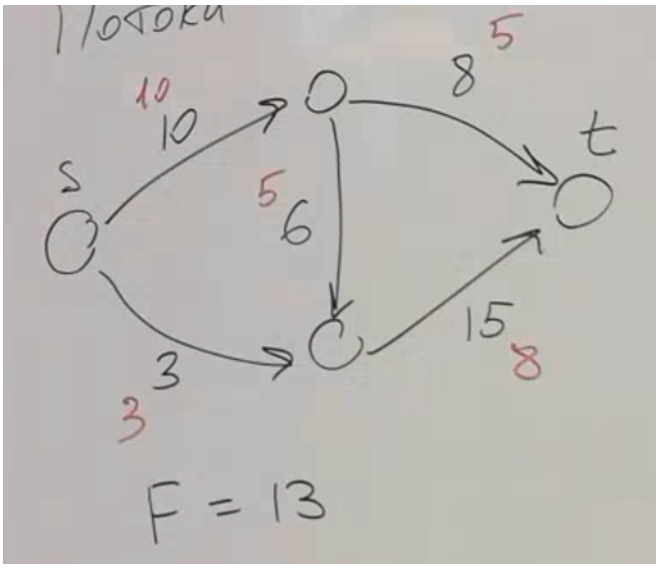
Пооптимизируем DFS. Прямо проходя в DFS–е можно сжимать вершины. Для этого можно быстро мерджить списки ребер.



В общем, если постараться, можно получить решение $\mathcal{O}(nm\alpha(m,n))$. А вообще, пацаны умеет делать за $\mathcal{O}(m\sqrt{n})$.

1.2 Потоки

Пусть у нас есть ориентированный граф. По этому графу течет некоторая *жижа*. Есть исток и есть сток, для каждого ребра известна пропусканая способность.



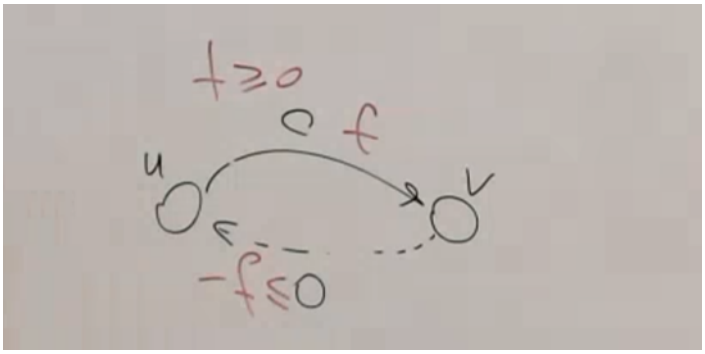
Обозначим C_{uv} — пропусканая способность ребра, f_{uv} — величина потока.

$f_{uv} \geqslant 0$, $f_{uv} \leqslant c_{uv}$.

$\forall v \neq s, v \neq t \ : \ \sum\limits_{u} f_{uv} = \sum\limits_{w} f_{vw}$.

$F = \sum\limits_{u} f_{uv} - \sum\limits_{w} f_{vw}$

Решать такую задачу в такой формулировке не очень удобно. Удобнее мыслить в симеетричной формулировке, хочется избавиться от двух сумм. Давайте считать, что для каждого ребра есть фиктивный обратный поток.



$f_{vu} = -f_{uv}$, $c_{vu} = 0$, $f_{uv} \leqslant c_{uv}$

$\sum\limits_{u} f_{vu} = 0$, $\forall v$ кроме s и t.

$F = \sum\limits_{u} f_{su} = \sum\limits_{w} f_{wt}$.

$\sum\limits_{v}\sum\limits_{u} f_{vu} = \sum f_{su} + \sum f_{tu}$.

На самом деле, не всегда можно просто так скаладывать потки $F = F_1 + F_2$. — не всегда валидный поток (может переполниться ребро)

Теорема 3. (Как бы утверждение) Любой поток можно декомпозировать на маленькие потоки

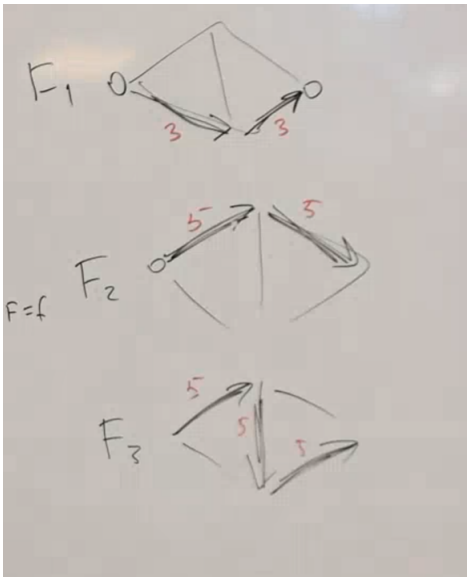
$$F = F_1 + F_2 + \ldots + F_k$$

F_i – путь $S \to t$ или цикл

Доказательство. План: взять путь и вычесть его. Как найти какой-нибудь путь?

1. Пойдём из S
2. Возьмём любое ребро по которому течёт жижа.
3. Рано или поздно либо дойдём до t, либо зациклимся.

Хорошо, мы нашли путь. Давайте удалим его, в терминах потоков надо взять минимальную пропускную способность ребра и вычесть из всех ребер.



Может ситуация, что из S все рёбра нулевые, в t все рёбра нулевые, а где-то в середине что-то происходит. Это значит, что остались только циклы. Для их нахождения нужно взять любое ненулевое ребро и запускаться уже от него.

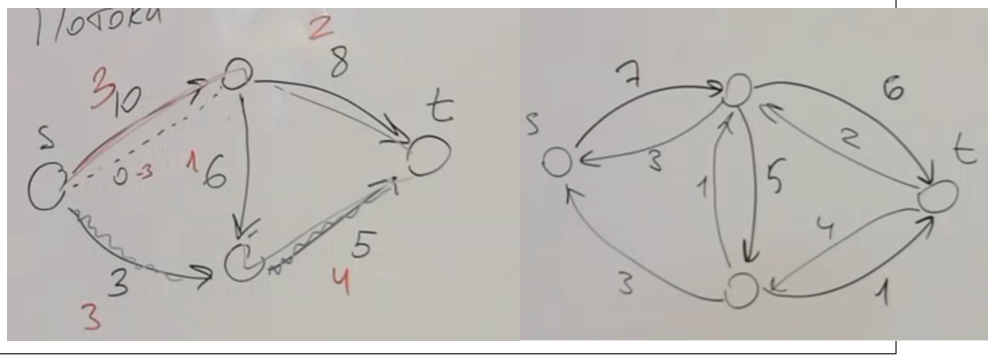
Всего будет $O(m)$ путей и циклов, т.к. мы каждый раз обнулям хотя бы одно ребро. ■

Вообще, мы хотим найти максимальный поток. Мы сперва возьмем пустое разбиение, а потом будем его постепенно дополнчть.

Определение 1. Ребро называется насыщенным, если вся его пропуская задействована.

Определение 2. Блокирующий поток — такой поток, что любой путь содержит насыщенное этим потоком ребро. Иными словами, в данной сети не найдётся такого пути из истока в сток, вдоль которого можно беспрепятственно увеличить поток.

Определение 3 (Остаточная сеть)**.** Пусть у нас был валидный поток. Для каждого ребра посчитаем $c'_{uv} = c_{uv} - f_{uv} \geqslant 0$. Остаточная сеть — подграф исходного, где на прямых ребрах написаны c_{uv}, а на обратных ребрах остаточная пропускная способность c'_{uv}.



Идея алгоритма (схема Форда–Фалкерсона)

- Возьмём поток F, воздмём максимальный поток $F_{\max}$ и посмотрим на их разность $\Delta F = F_{\max} - F$
- ΔF — поток в остаточной сети C'_F $\Delta f_{uv} = f_{\max uv} - f_{uv} \leqslant C_{uv} - f_{uv} = C'_{uv}$.
- Нет пути $S \to t$ в сети C'_F тогда и только тогда когда $F = F_{\max}$

```
int dfs(v, min Δ):
    if mark(v):
        return 0
    mark[v] = True
    for (uv): C_vu − f_uv > 0
        Δ = dfs(u, min(min Δ, C_uv − f_uv))
        if Δ > 0:
```

```
                f_vu += Δ
                f_uv -= Δ
                return Δ
        return 0
```

Время работы алгоритма — $\mathcal{O}(F \cdot m)$.

1.2.1 Разрезы

Задача 1. Пусть есть граф. Мы хотим удалить некоторые ребра, чтобы не было пути из S в T. Определим $C = \sum c_{uv}$, для таких u, v, что нужно удалить ребро uv.

Надо найти $C_{\min}$.

На самом деле это двойственная задача к поиску макимального потока.

Как искать минимальный разрез? Найдем максимальный поток, рассмотрим его дополняющую сеть. ... ДОПИСАТЬ

1.2.2 Алгоритм Эдмондса–Карпа

Сделаем все то же самое, но dfs заменим на bfs.

Пусть $d[v]$ — растояние (число ребер) от S до v в остаточной сети.

Теорема 4. $d[v]$ не убывает, если выбирать кратчайшие лополняющие пути.

Доказательство. Возьмем кратчайший путь $S \to T$. Ребра с минимальной дельтой на пути могли пропасть из остаточной сети. Какие могли добавться? Те, у которых были обратные ребра.

Заметим, что $d[v]$ не убывает.

Ребро может пропасть из дополняющей сети $d[u] = d[u] + 1$.
Ребро может восстановавливаться в дополняющей сети $d'[u] = d'[v] + 1$.

Таким образом $d'[u] \geqslant d[u] + 2$.

Поэтому ребро может быть добавлено и удалено не больше $\mathcal{O}(n)$ раз, а посколько ребер m, то всего таких запусков bfs может быть $\mathcal{O}(nm)$. ■

Итого, наш алгоритм будет работать за $\mathcal{O}(nm^2)$. На самом деле, время работы асимптотически не больше, чем $\mathcal{O}(m \cdot F)$.

Анонс на дальше (что умеют люди):

1. $\mathcal{O}(nm)$ – но стращно,такое смотреть небудем
2. Плавно дойдём до $\mathcal{O}(nm \ln n)$
3. $\mathcal{O}(nm \ln C)$
4. $\mathcal{O}(n^3)$

Определение 4 (Масштабирование)**.** Почему наш алгоритм работал долго? ПОтому что он искал плохие пути. Давайте запретим ему искать единичные пути. Запустим ту же штуку, но на фазе k заставим его брать пути длиной хотя бы 2^k

$k = \log C \dots 0$ искать пути $\Delta \geqslant 2^k$

Доказательство. $k+1 \to k$:

$F_{\max} - F \leqslant 2^{k+1} \cdot m$

Путей с $\Delta = 2^k \quad \leqslant 2m$

Финальная асимптотика: $O(\ln C \cdot m^2)$ – слабополиномиальный алгоритм.

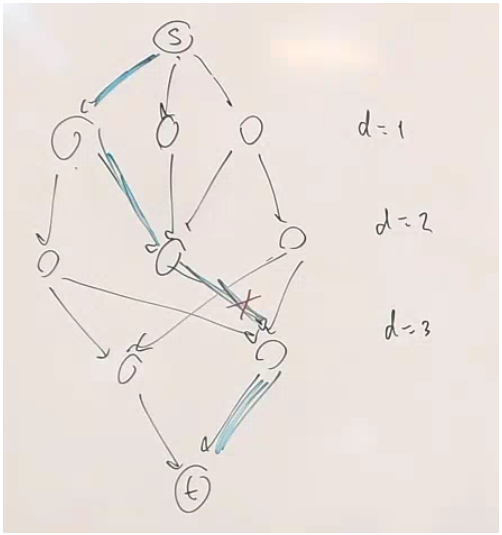
■

Алгоритм Диница

Определение 5. Алгоритм Диница.

Проведём bfs , оставим только те рёбра, которые лежат на кратчайших путях (с меньшего на больший слой). Уберём все тупики.

Найдём так кратчайший путь (идём по любому ребру), запустим на нём поток величины равной минимальной пропускной способности на пути. Таким образом хотя бы одно из рёбер уйдёт. Повторим. Остановимся когда s станет тупиком.




```
while есть путь s → t:
    bfs()
    while dfs():
        Δ = min
        f += Δ
```

```
            удалить тупики
    можно( удалять лениво:
        если dfs не смог дойти до t,
        возвращаемся из рекурсии и
        убираем за собой ребро)
```

Внешний *while* выполняется не больше n раз. Внутренний не больше m. Суммарная асимптотика: $\mathcal{O}(n^2m)$. n на каждый dfs. Удаление тупиков – суммарно n, не страшно.

Если добавить масштабирование, то асимптотика станет $\mathcal{O}(nm \ln C)$

Алгоритм Гольберга-Торьяна

План: не терять все пути, которые мы нашли раньше.

1. Нашли какой-то кратчайший путь
2. Нужно найти на нём минимальное C', только значение.
3. Прибавить этот C' ко потоку
4. Нужно найти минимальное ребро, чтобы его убрать.
5. Помним предыдущий путь. На нём есть ребро идущее в тупик. Уберём его. Будем делать так пока таких не будет, а тогда мы сможем пройти в t.
6. В целом будем идти до ближайшего осколка предыдущего пути.
7. Из этих путей будем дерево.

План закончился, теперь алгоритм:

1. У каждой вершины будет не более одного исходящего ребра – дерево.
2. s принадлежит дереву. Пойдём по помеченным рёбрам, пока не упёрлись. Дошли до корня дерева s.
3. Если дошли до t – радуемся и на этом пути делаем все действия (минимальная пропускная спосность, ко всем на пути прибавляем Δ, убираем минимальные)
4. Если не дошли – идём куда-нибудь. Идём до корня уже той вершины. Заодно помечаем ребро п которому мы “куда-нибудь” перешли. Повторяем.

Псевдо-псевдо-кот:

```
    while есть путь s → t:
        bfs()
        while не удалили s:
            v = root(s)
            if v == t:
```

```
6                   Δ, e = min(s..v)
7                   add(s..v, -\Delta)
8                   cut(e)
9               else:
10                  if v -- тупик:
11                      for all wv:
12                          cut(wv)
13                  else:
14                      (vu) -- любое ребро (d(u) = d(v) + 1)
15                      link(vu)
```

- Большоой while – n раз
- bsf – m
- корень суммарн $m \ln n$. Каждый if тоже $m \ln n$

Время работы алгоритма — $\mathcal{O}(nm \ln n)$.

Алгоритм Малхотры – Кумара – Махешвари

Идея: удалять за раз не ребро, а целую вершину. Какую вершину проще всего удалить? У которой минимальная пропускная способность...
$c(v) = \min(\sum c_{uv}, \sum c_{uw})$

1. Берём вершину. Течём из неё по одному ребру. Если не хватает, пихаем сколько получится. Пихаем в другие, чтобы в сумме было $\Delta/$
2. Повторяем на вершинах ниже.
3. От s делаем по обратным рёбрам.

Время работы алгоритма — $\mathcal{O}(n \cdot (n \cdot n + m)) = O(n^3)$