академия больших данных made mail.ru group

Кратчайшие пути. Потоки.

Мацкевич Степан

Алгоритмы и структуры данных

План

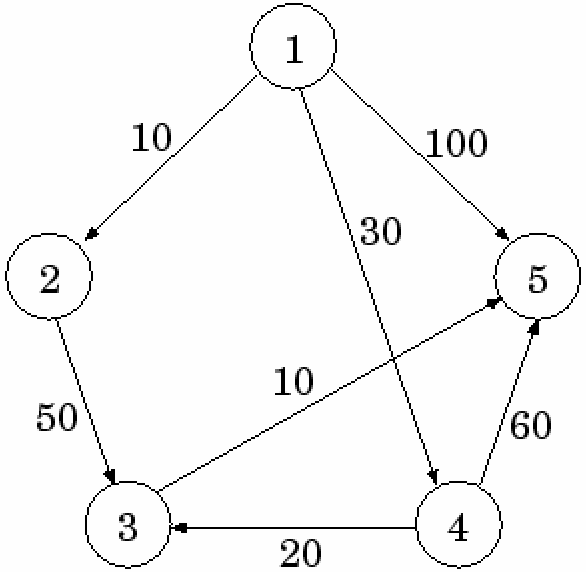
- **Кратчайшие пути**
  - Алгоритм Дейкстры
  - Алгоритм A*
  - Алгоритм Беллмана-Форда
- **Потоки**
- **Паросочетания**

Взвешенный граф

Взвешенный граф – это граф, дугам которого поставлены в соответствие веса, так что дуге (x_i, x_j) сопоставлено некоторое число $c(x_i, x_j) = c_{ij}$, называемое *длиной* (или *весом*, или *стоимостью*) дуги.

Может быть ориентированным и неориентированным.

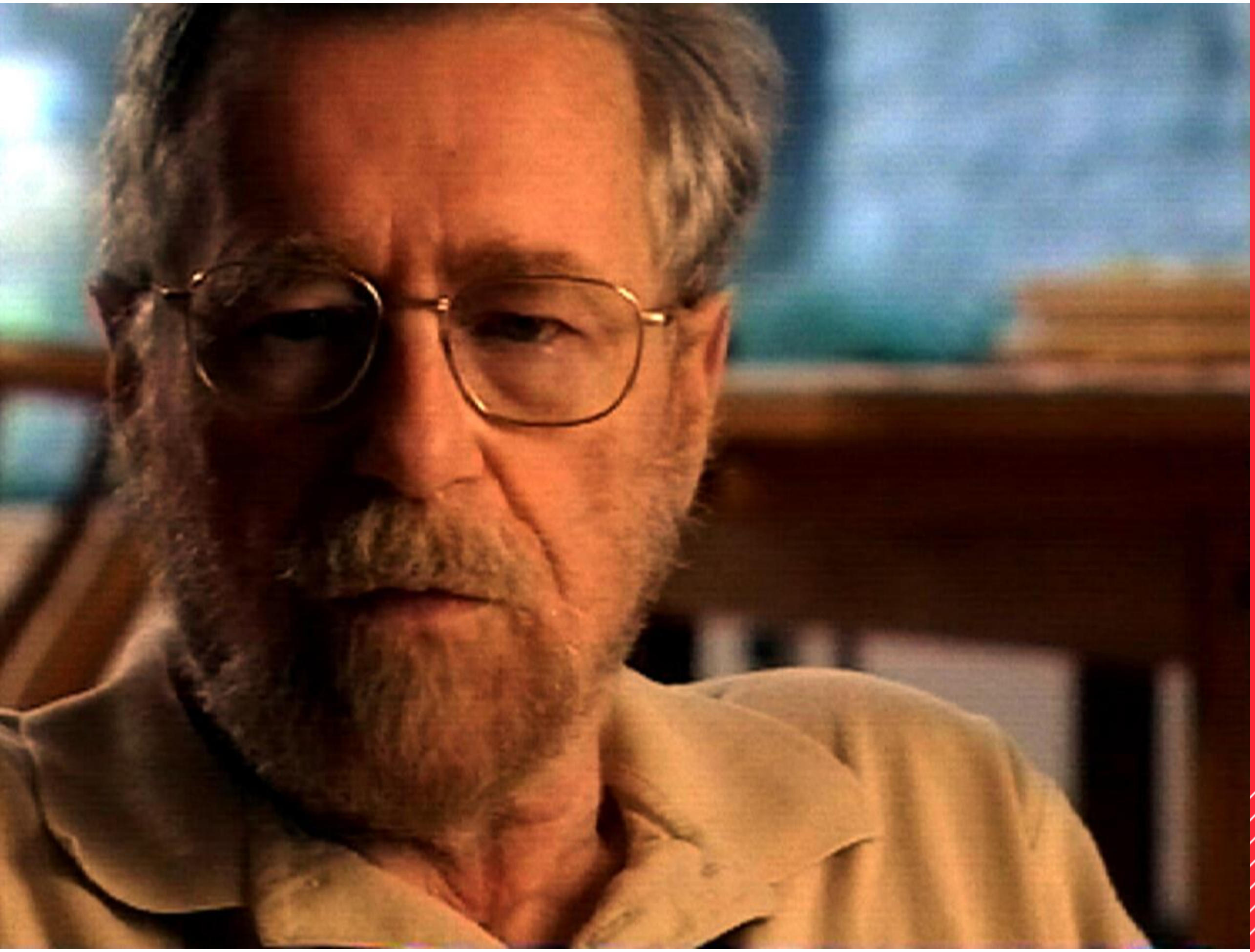
Длина пути во взвешенном графе – это сумма длин (весов) тех ребер, из которых состоит путь.



Задачи поиска кратчайших путей

G – ориентированный взвешенный граф.

- **SPSP** (Single Pair Shortest Path problem) – поиск кратчайшего пути между двумя вершинами.
  Алгоритмы Дейкстры, A*, IdaStar.
- **SSSP** (Single Source Shortest Paths problem) – поиск кратчайших путей из выделенной вершины до всех остальных.
  Алгоритмы Дейкстры, Беллмана-Форда.
- **APSP** (All Pairs Shortest Paths problem) – поиск кратчайших путей между всеми парами вершин.
  Алгоритмы Флойда, Джонсона.



Алгоритм Дейкстры

Алгоритм Дейкстры

G – ориентированный взвешенный граф, s – стартовая вершина,
$w(u, v) \geq 0$ – веса ребер неотрицательны.

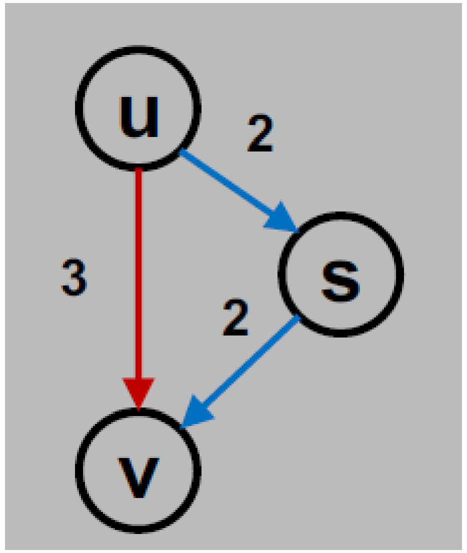
Храним для каждой вершины u два значения:

- $d[u]$ –длина некоторого пути, ведущего в вершину,
- $\pi[u]$ – предок вершины u в этом пути.

Релаксация ребра

Релаксация ребра – это проверка того, дает ли продвижение по данному ребру новый кратчайший путь к конечной вершине.

```
bool Relax( u, v ) {
    if( d[v] > d[u] + w( u, v ) ) {
        d[v] = d[u] + w( u, v );
        pi[v] = u;
        return true;
    }
    return false;
}
```

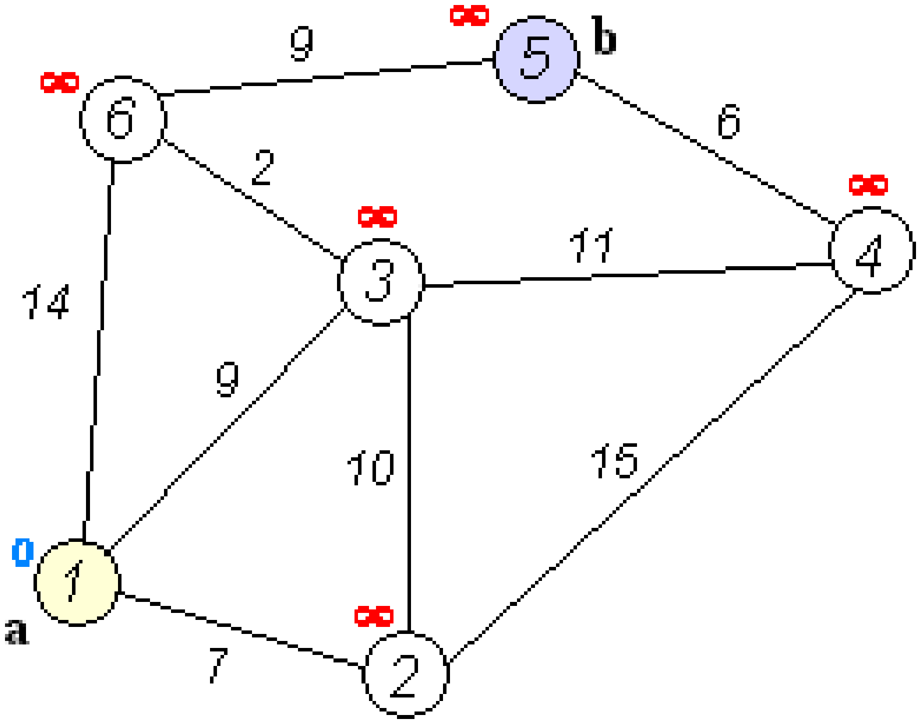


Алгоритм Дейкстры

Алгоритм Дейкстры похож на BFS, но вместо обычной очереди используется очередь с приоритетом.

```
void Dijkstra( G, s ) {
    pi[V] = -1;
    d[V] = INT_MAX;
    d[s] = 0;
    priority_queue<int> q; q.push( s );
    while( !q.empty() ) {
        u = q.top(); q.pop();
        for( ( u, v ) : ребра из u ) {
            if( d[v] == INT_MAX ) {
                d[v] = d[u] + w(u, v);
                pi[v] = u;
                q.push( v );
            } else if(Relax( u, v )) {
                q.DecreaseKey( v, d[v] );
        }   }
    }
}
```

Пример



Оценка времени работы

Будем использовать в качестве реализации очереди с приоритетом двоичную кучу.

- Добавление узла: $O(\log V)$. Не более V операций.
- Уменьшение значения ключа: $O(\log V)$. Не более E операций.

Всего $T = O\big((E+V) \cdot logV\big)$.

Используемая память $M = O(V)$.

Время работы можно улучшить за счет использования другой структуры для очереди с приоритетом.

Важная тонкость

Как найти узел v в двоичной куче, чтобы вызывать

`q.DecreaseKey(v, d[v])` ?

Решение 1. Хранить позицию i в куче для каждого узла v. В момент изменения позиции в куче обновлять эту позицию, хранящуюся в описании узла.

Решение 2. Использовать вместо бинарной кучи множество `set<pair<d, v>>`.
Во время релаксации удалять старую пару `<d[v], v>`, добавлять новую.

Корректность

<u>Утверждение.</u> G – ориентированный граф с неотрицательными ребрами. Алгоритм Дейкстры на таком G работает корректно.

<u>Доказательство.</u> Докажем, что в момент извлечения узла v из очереди выполняется $d[v] = \rho(s, v)$.

Индукция по числу вершин.

Рассмотрим v – вершину из кучи с минимальным d[v] в некоторый момент, v извлекается алгоритмом Дейкстры.

Рассмотрим кратчайший путь $s \rightsquigarrow v$. Пусть x – последняя на этом пути вершина, которая была извлечена на пути в v. u – следующая после x на этом пути

$$d[x] = \rho(s, x),$$
$$d[v] \leq d[u],$$
$$d[u] \leq d[x] + w(x, u) = \rho(s, x) + w(x, u) = \rho(s, u).$$

Итого:

$$d[v] \leq d[u] \leq \rho(s, u) \leq \rho(s, v).$$

Потенциал. Алгоритм A*.

Потенциал

Введем функцию на вершинах графа – потенциал:

$$\pi: V \to \mathbb{R}$$

Определим измененный вес ребра, добавив разность потенциалов:

$$w'(u, v) = w(u, v) + \pi(v) - \pi(u).$$

Как изменятся длины путей?

$$w'(p) = \sum w'(x_{i-1}, x_i) = \sum \big(w(x_{i-1}, x_i) + \pi(x_i) - \pi(x_{i-1})\big) =$$

$$= \sum w(x_{i-1}, x_i) + \pi(x_k) - \pi(x_0) =$$

$$= w(p) + \pi(x_k) - \pi(x_0)$$

Потенциал

То есть длина пути изменится на разность потенциалов в конечной и начальной точке.

<u>Следствие 1.</u>

Кратчайшие пути останутся кратчайшими в новой весовой функции w^′.

И наоборот, кратчайшие пути относительно w^′ будут кратчайшими относительно w.

Можно применять алгоритмы поиска кратчайших путей для измененной весовой функции на разность потенциалов.

Пример 1

$$\pi[v] = -\rho(s, v)$$

Такой потенциал обнуляет ребра, лежащие на кратчайших путях из вершины s.

$$w^{'}(u, v) = w(u, v) - \rho(s, v) + \rho(s, u) \geq 0$$

Кроме того, на обновленном графе можно применить алгоритм Дейкстры уже из любой вершины, достижимой из s.

Пример 2

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

Такой потенциал обнуляет ребра, лежащие на кратчайших путях к вершине t.

$$w'(u, v) = w(u, v) + \rho(v, t) - \rho(u, t) \geq 0$$

Кроме того, на обновленном графе можно применить алгоритм Дейкстры уже из любой вершины, достижимой из s.

Более того, алгоритм будет «сразу» находить кратчайший путь к вершине t.

Алгоритм A*

$$\pi[v] = \varepsilon(v, t)$$

– эвристика, оценивающая длину пути от v до t. $\varepsilon(t, t) = 0$.

Будем записывать $\varepsilon(u) = \varepsilon(u, t)$.

Чтобы алгоритм Дейкстры работал корректно, необходима неотрицательность ребер:

$$w'(u, v) = w(u, v) + \varepsilon(v, t) - \varepsilon(u, t) \geq 0.$$

Это условие называется «монотонностью».

Определение. Эвристика **монотонна**, если

$$\varepsilon(u) \leq w(u, v) + \varepsilon(v).$$

Похоже на неравенство треугольника.

Эвристика A*

Определение. Эвристика ε допустима, если

$$\varepsilon(u) \leq \rho(u, t).$$

Утверждение. Монотонная эвристика допустима.

Доказывается по индукции.

Рассмотрим а. Дейкстры с потенциалами

```
void Dijkstra( G, s ) {
    pi[V] = -1;
    d[V] = INT_MAX;
    d[s] = 0;
    priority_queue<int> q; q.push( s );
    while( !q.empty() ) {
        u = q.top(); q.pop();
        for( ( u, v ) : E ) {
            if( d’[v] == INT_MAX ) {
                d’[v] = d’[u] + w’(u, v); pi[v] = u;
                q.push( v );
            } else if( d’[v] > d’[u] + w’(u, v) ) {
                d’[v] = d’[u] + w’(u, v); pi[v] = u;
                q.DecreaseKey( v, d’[v] );
            }
        }
    }
}
```

Модифицируем

```
void Dijkstra( G, s ) {
    pi[V] = -1;
    d[V] = INT_MAX;
    d[s] = 0;
    priority_queue<int> q; q.push( s );
    while( !q.empty() ) {
        u = q.top(); q.pop();
        for( ( u, v ) : E ) {
            if( d[v] == INT_MAX ) {
                d[v] = d[u] + w(u, v); pi[v] = u;
                q.push( v );
            } else if( d[v] + e[v] – e[s] > d[u] + w(u, v) + e[v] – e[s] ) {
                d[v] = d[u] + w(u, v); pi[v] = u;
                q.DecreaseKey( v, d[v] + e[v] – e[s] );
            }
        }
    }
}
```

Получился A*

```
void AStar( G, s ) {
    pi[V] = -1;
    d[V] = INT_MAX;
    d[s] = 0;
    priority_queue<int> q; q.push( s );
    while( !q.empty() ) {
        u = q.top(); q.pop();
        for( ( u, v ) : E ) {
            if( d[v] == INT_MAX ) {
                d[v] = d[u] + w(u, v); pi[v] = u;
                q.push( v );
            } else if( d[v] > d[u] + w(u, v) ) {
                d[v] = d[u] + w(u, v); pi[v] = u;
                q.DecreaseKey( v, d[v] + e[v] );
            }
        }
    }
}
```

Алгоритм Беллмана-Форда

Есть отрицательные веса?

G – ориентированный взвешенный граф.

Если в графе нет циклов отрицательного веса, достижимых из s, то алгоритм Беллмана-Форда находит все кратчайшие пути из s до остальных вершин (SSSP).

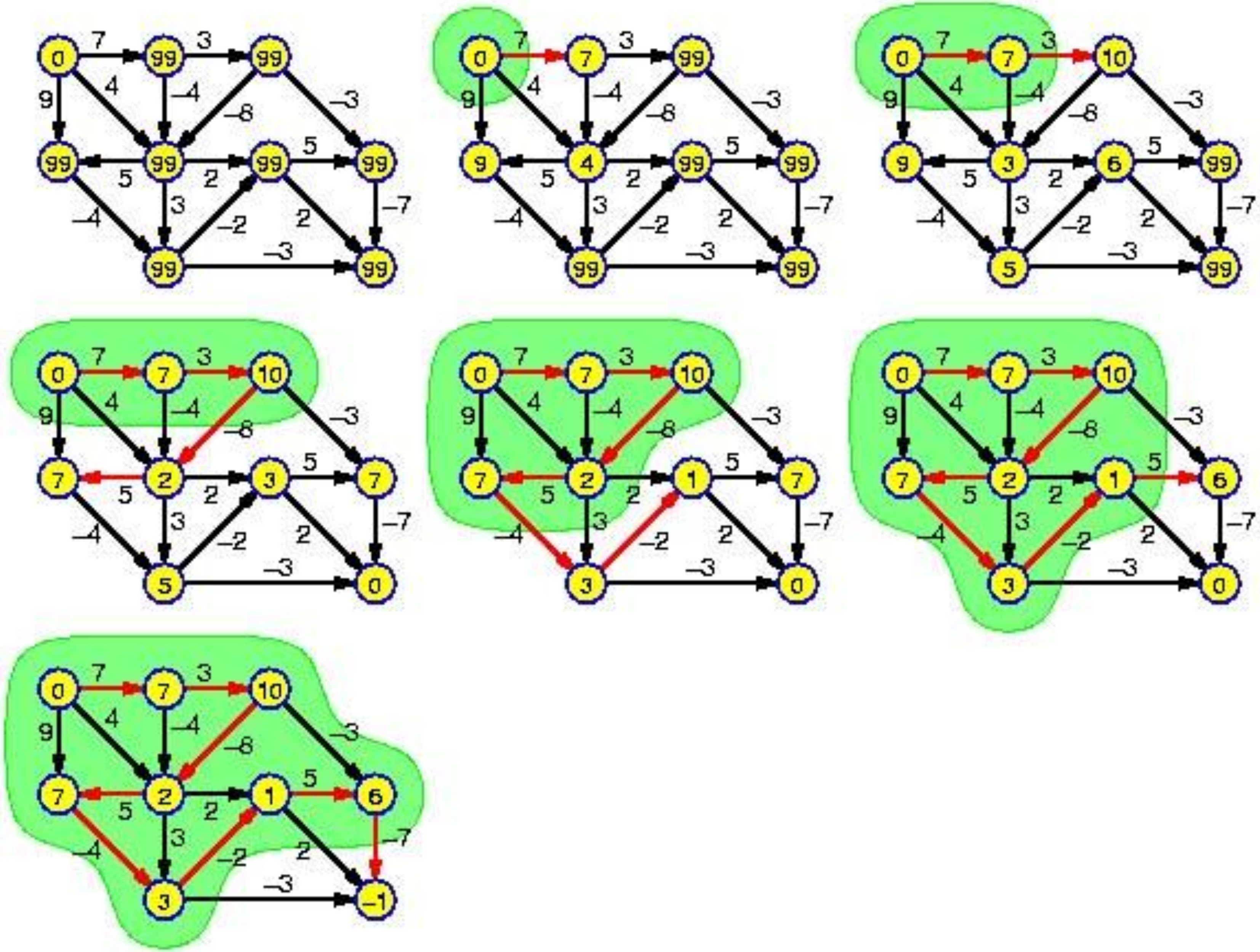
Если в графе есть достижимый из s цикл отрицательного веса, то алгоритм Беллмана-Форда сообщит о его наличии. (и может выдать его).

Беллмана-Форда

V-1 раз релаксируем все ребра.

```
bool BellmanFord( G, s ) {
    for( int i = 0; i < V; ++i ) {
        for( (u, v) : E ) Relax( u, v );
    }
    // Детектирование цикла.
    for( (u, v) : E ) {
        if( Relax( u, v ) )
            return false;
    }
    return true;
}
```

BELLMAN–FORD ALGORITHM



Беллмана-Форда. Анализ

Утверждение. Если в G нет циклов отрицательного веса, достижимых из s, то алгоритм Беллмана-Форда вернет true и

$$d(v) = \rho(s, v).$$

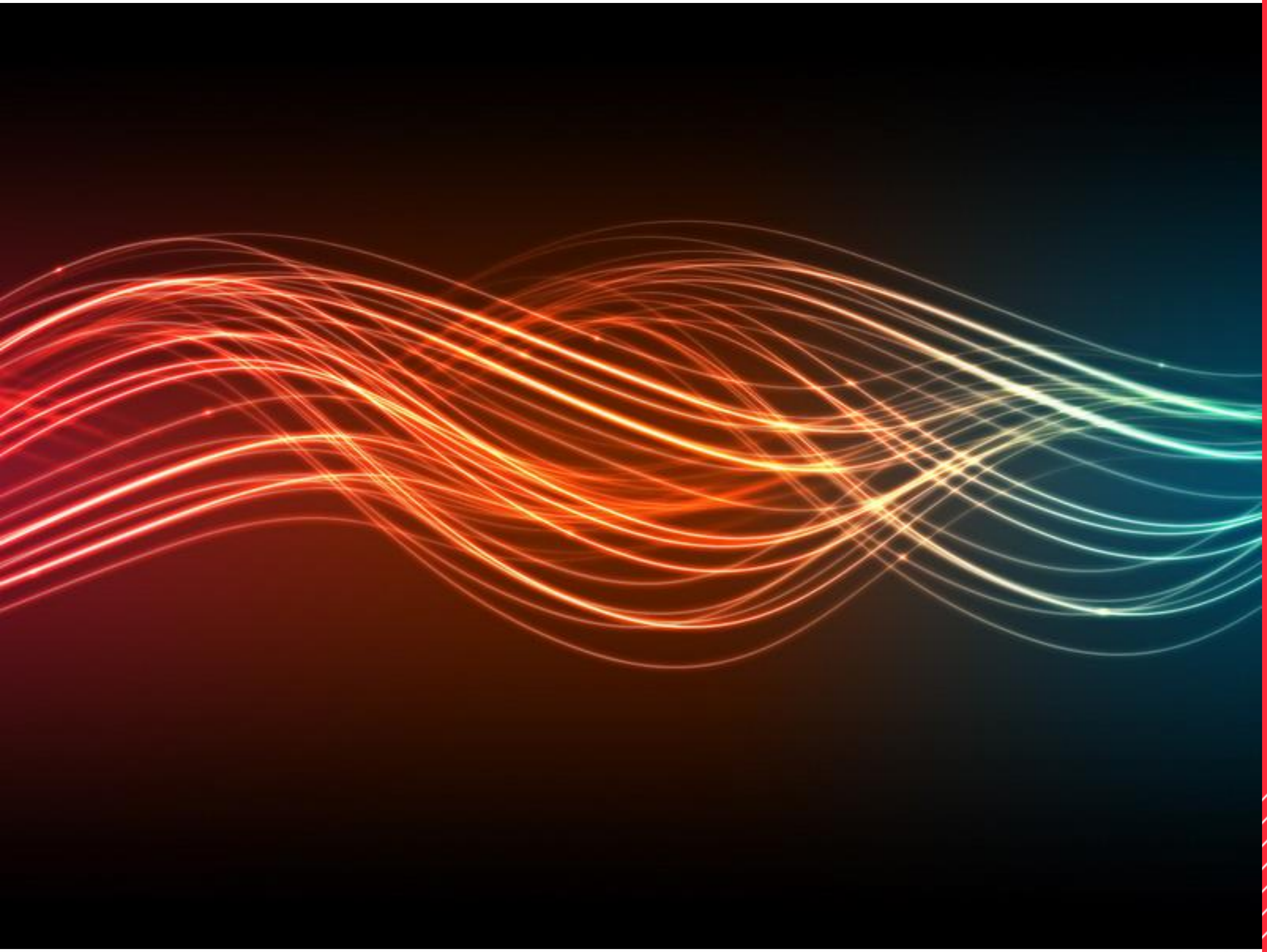
Доказывается по индукции.

Время работы: V итераций по E релаксаций.

$$T = O(V \cdot E)$$

Поиск цикла отрицательного веса

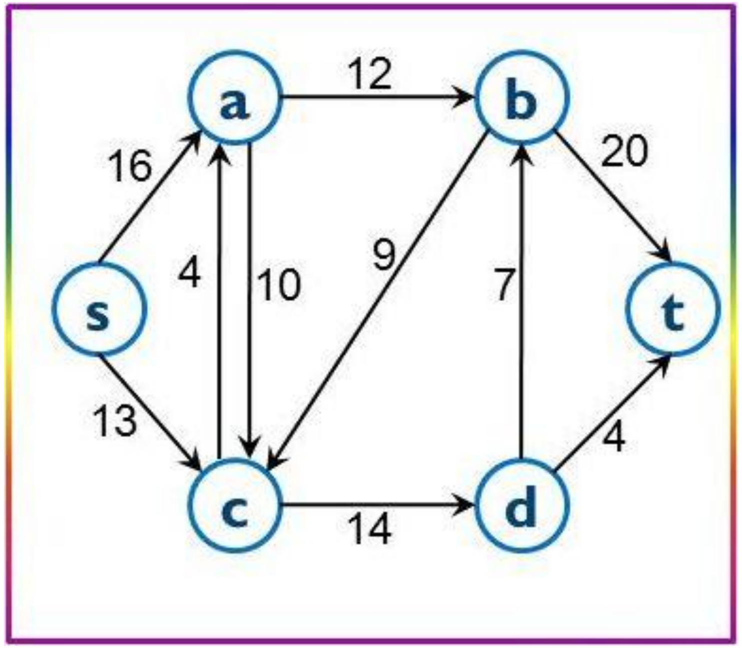
Если алгоритм вернул false, то цикл можно найти, пройдя по предкам от ребра, релаксировавшегося на V-ом шаге. Путь по предкам не дойдет до s, но зациклится на цикле отрицательного веса.



Потоки в сетях

Сеть

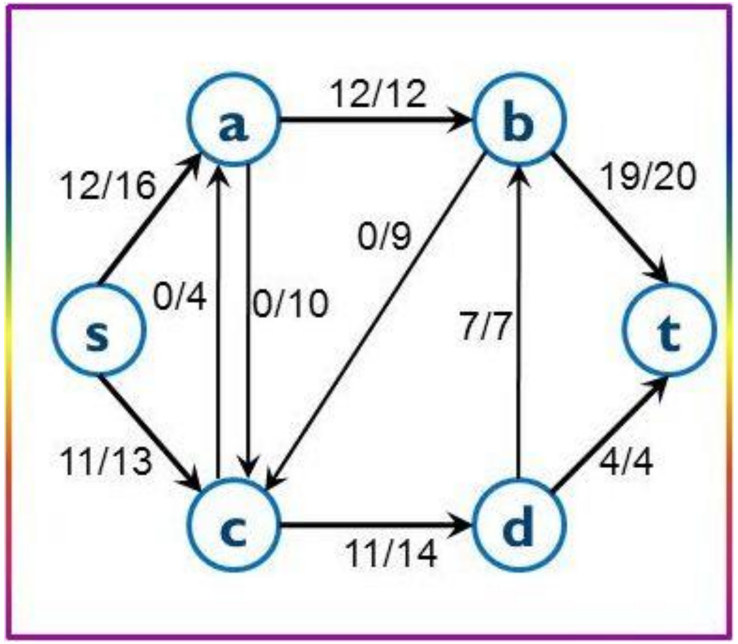
<u>Определение.</u> **Сеть** (flow network) – ориентированный граф $G(V, E)$, в котором каждое ребро $(u, v) \in E$ имеет положительную **пропускную способность** (capacity) $c(u, v) > 0$.
В сети выделены две вершины: s – **исток**, t – **сток.**
Если $(u, v) \notin E$, то предполагается, что $c(u, v) = 0$.



Поток (в сети)

<u>Определение.</u> **Потоком** (flow) в сети G называется действительнозначная функция $f: V \times V \to \mathbb{R}$, удовлетворяющая условиям:
1) $f(u, v) = -f(v, u)$ – антисимметричность,
2) $f(u, v) \leq c(u, v)$ – ограничение пропускной способности.
3) $\sum_v f(u, v) = 0$ для всех вершин u,
кроме s и t – закон сохранения потока.

<u>Определение.</u> **Величина** потока f определяется как $|f| = \sum_v f(s, v)$.



<u>Свойство.</u> Если ребер (u, v), (v, u) нет,

то $f(u, v) = 0$.

Утверждения

Обозначение 1. $f^{+}(v)$ - сумма положительных $f(v,u)$ - исходящий поток,
$f^{-}(v)$ - сумма положительных $f(v,u)$ - входящий поток.

Обозначение 2. $f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$

Утверждение 1. $f^{+} = f^{-}$

Утверждение 2. $f(u,V) = 0$, для любой $u \in V \backslash \{s,t\}$

Утверждение 3. $f(X,Y) = -f(Y,X)$

Утверждение 4. $f(X,X) = 0$

Утверждение 5. Если $X \cap Y = \emptyset$, то

$$f(X \cup Y, Z) = f(X,Z) + f(Y,Z),$$
$$f(Z, X \cup Y) = f(Z,X) + f(Z,Y).$$

Величина потока

Утверждение. $|f| = f(V, t)$.

Доказательство.

$$|f| = f(s, V) = f(V, V) - f(V\backslash s, V) = -f(V\backslash s, V) =$$
$$= f(V, V\backslash s) = f(V, t) + f(V, V\backslash\{s, t\}) = f(V, t).$$

Поток через разрез

<u>Определение.</u> **Разрез** (cut) сети $G(V,E)$ – разбиение множества V на две части S и T, такие что $s \in S, t \in T$.

<u>Утверждение.</u> Величина потока равна потоку через разрез.

$$|f| = f(S,T).$$

<u>Доказательство.</u>

$$|f| = f(s,V) = f(S,V) - f(S\backslash s,V) = f(S,V) = \cdots = f(S,T)$$

Остаточная сеть

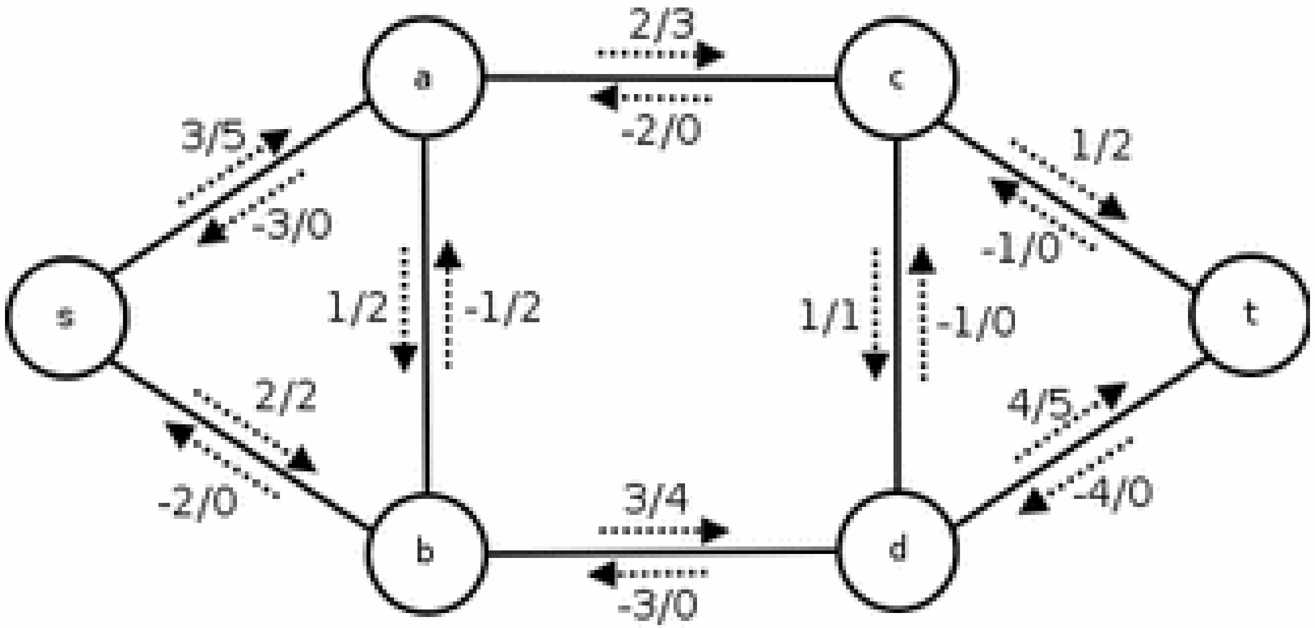
Определение. **Остаточная пропускная способность ребра**

$$c_f(u, v) = c(u, v) - f(u, v).$$

Утверждение. Остаточная пропускная способность – неотрицательна.

Определение. **Остаточная сеть** G_f – граф G, в котором оставлены/добавлены только те, ребра, остаточная пропускная способность которых строго положительна.

Остаточная сеть



Дополняющий путь

<u>Определение.</u> **Дополняющий (увеличивающий) путь** в остаточной сети – это путь из s в t, проходящий по ребрам с ненулевой остаточной пропускной способностью $c_f(u, v)$.

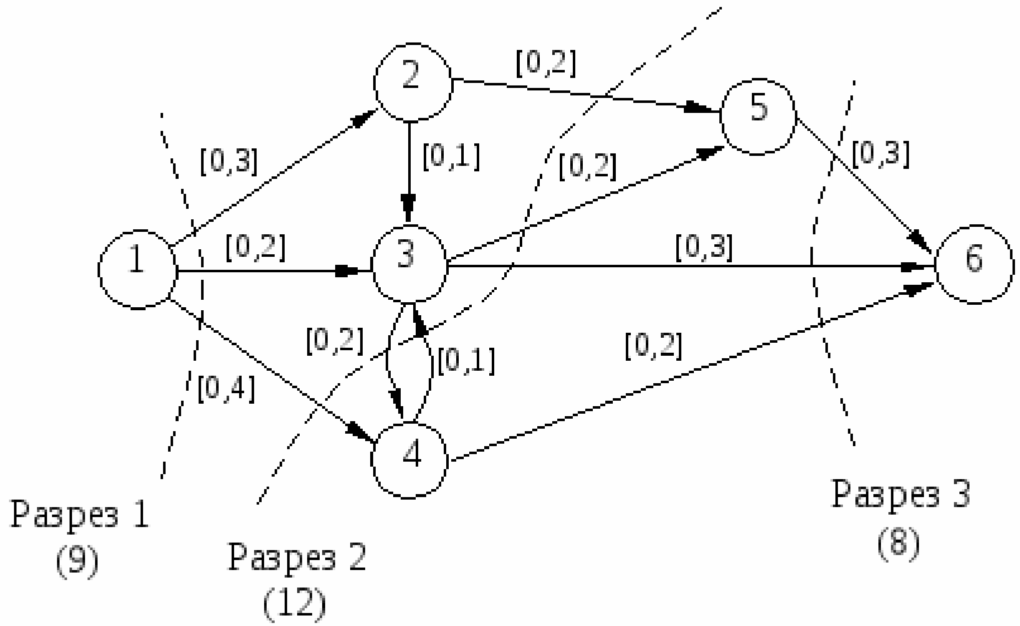
Общую величину потока можно увеличить на значение потока по увеличивающему пути.

Пусть **c** – минимальная остаточная пропускная способность по ребрам увеличивающего пути. Тогда поток

$$f_p(u, v) = \begin{cases} c, (u, v) \in p \\ -c, (v, u) \in p \\ 0, \text{иначе} \end{cases}$$

Теорема Форда-Фалкерсона

<u>Теорема.</u> Следующие утверждения эквивалентны.
1) Поток f максимален.
2) Остаточная сеть G_f не содержит дополняющих путей.
3) Для некоторого разреза (S, T) выполнено равенство $|f| = c(S, T)$.



Алгоритмы Форда-Фалкерсона и Эдмондса-Карпа

Алгоритм Форда-Фалкерсона

1. Строим остаточную сеть $G_f = G$.
2. while($\exists$ увеличивающий путь p в G_f) {

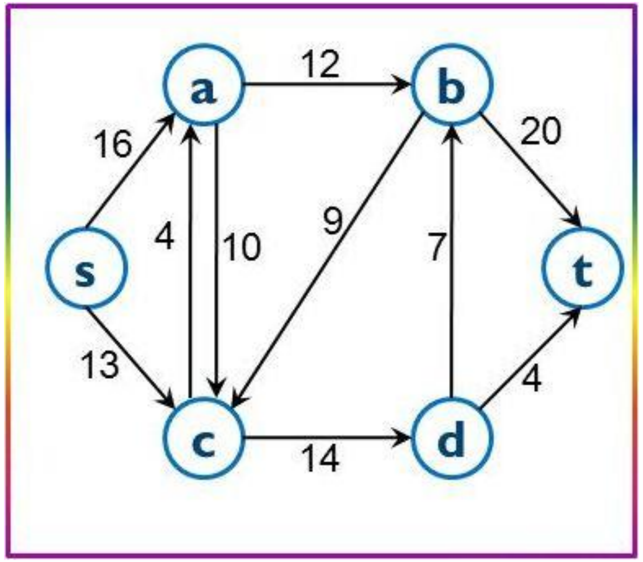
    вычисляем c_p – пропускную способность пути p.

    f += f_p; где f_p – поток, пропущенный по p.

    $|f|$ += c_p;

    обновляем G_f.

   }



Алгоритм Форда-Фалкерсона. Оценка.

Одна итерация:

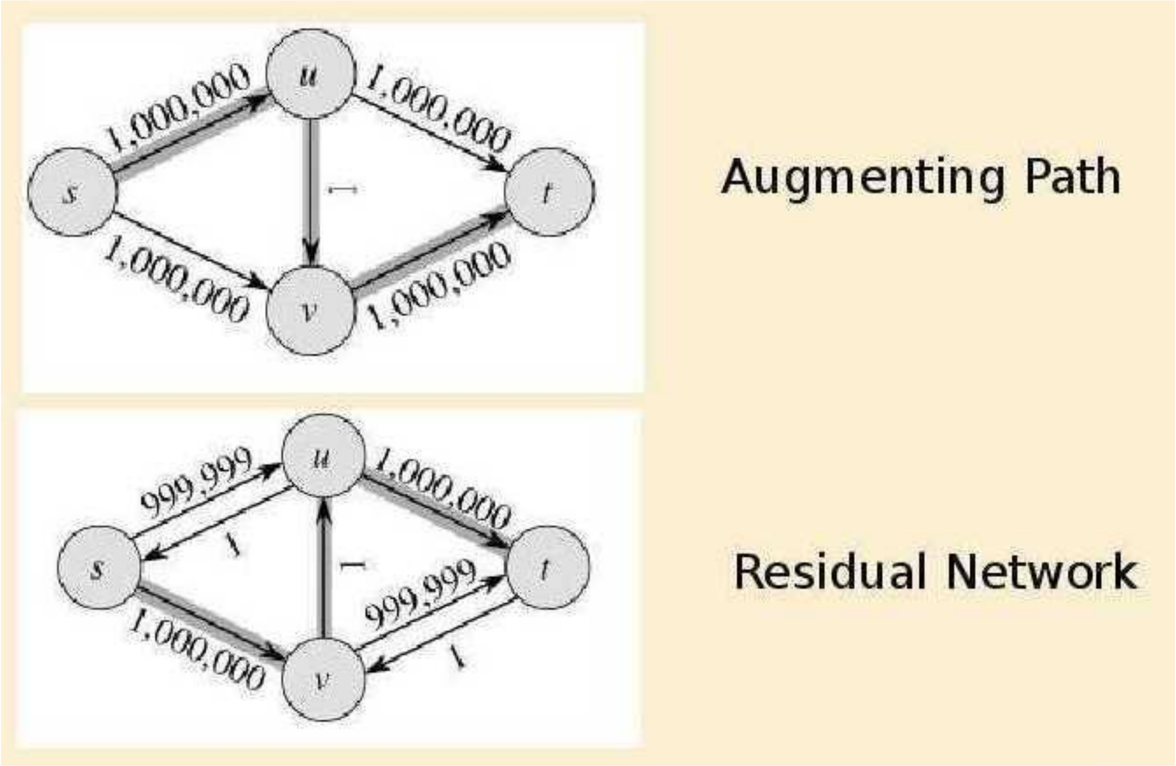
- поиск увеличивающего пути p – $O(E)$,
- увеличение потока f и обновление остаточной сети – $O(V)$.

Если ребра имеют целочисленную пропускную способность, то за одну итерацию поток увеличивается не менее чем на 1.

Общее число итераций в этом случае – не более $|f|$

$$T = O(E \cdot |f|).$$

Пример «плохого случая»



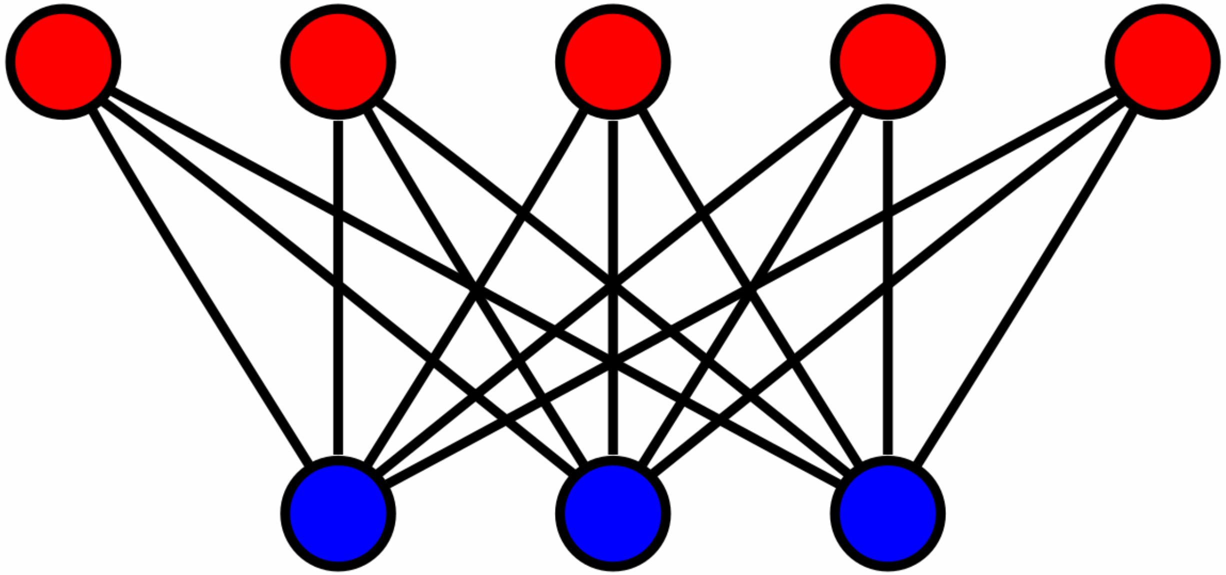
Алгоритм Эдмондса-Карпа

Оптимизация алгоритма Форда-Фалкерсона, в которой увеличивающий путь = кратчайший путь между s и t по количеству ребер.

$$T = O(VE^2).$$

Одна итерация:

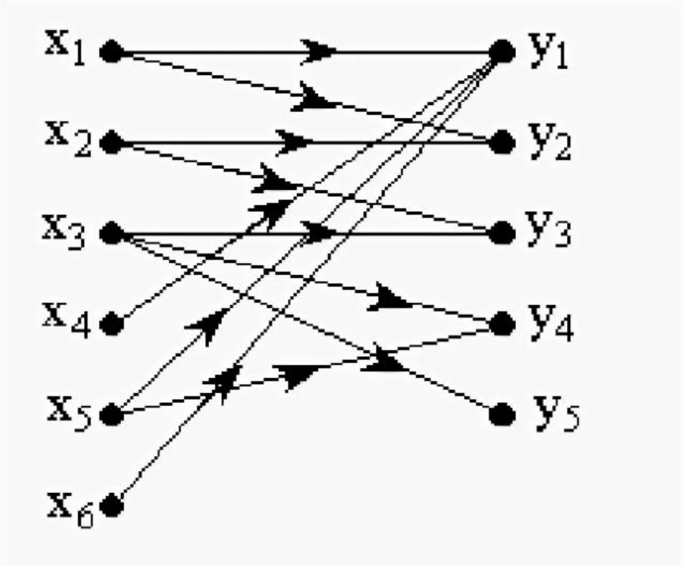
1. Ищем кратчайший по числу ребер дополняющий путь в остаточной сети - BFS.
2. Добавление найденного пути в поток.
3. Обновление остаточной сети.

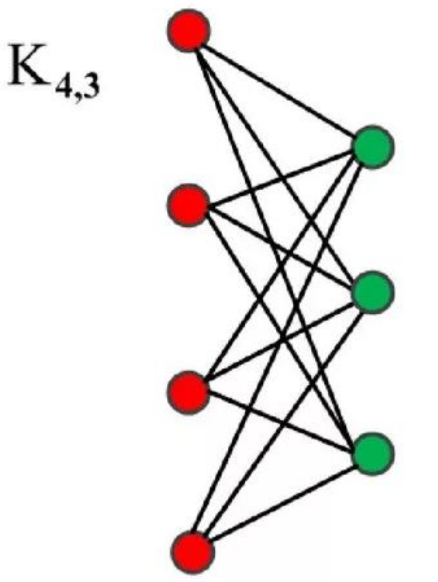


Немного о паросочетаниях

Двудольный граф

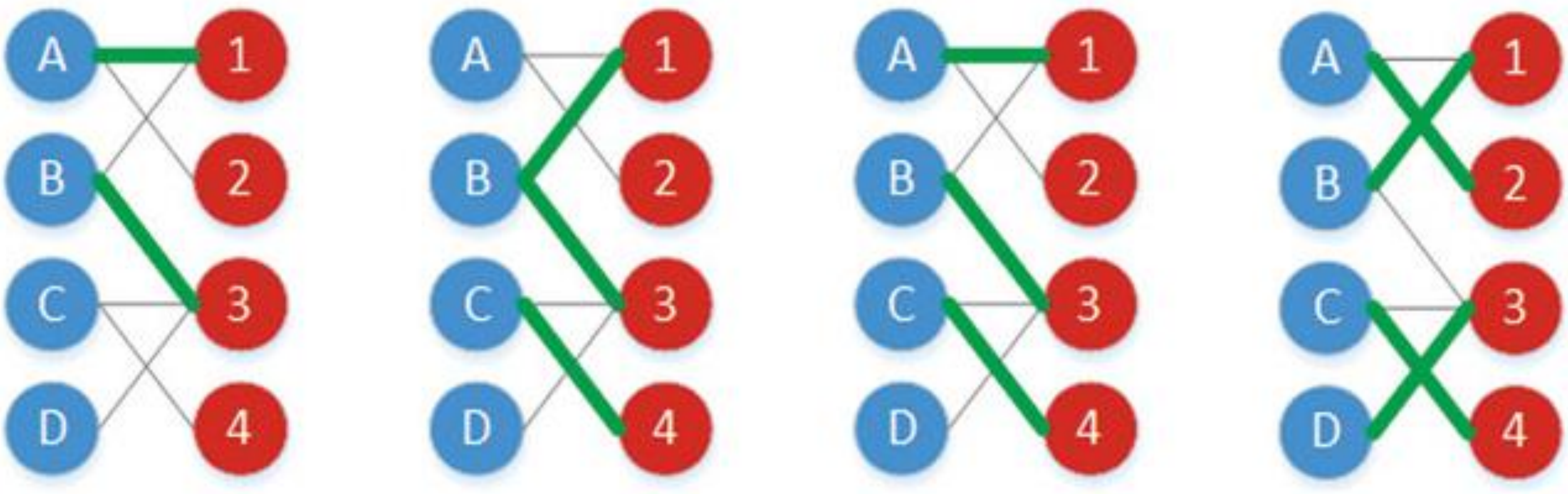
Определение. **Двудольный граф** (или **биграф**) – граф, множество вершин которого можно разбить на две части таким образом, что каждое ребро графа соединяет какую-то вершину из одной части с какой-то вершиной другой части, то есть не существует ребра, соединяющего две вершины из одной и той же части.





Паросочетание (matching)

<u>Определение.</u> **Паросочетание** M в двудольном графе – произвольное множество ребер двудольного графа, такое что никакие два ребра не имеют общей вершины.



Паросочетание | Не паросочетание | Не максимальное паросочетание | Максимальное паросочетание

Прочие определения

Определение. Вершины двудольного графа, инцидентные рёбрам паросочетания M, называются **покрытыми** (matched), а неинцидентные — **свободными** (unmatched).

Определение. **Максимальное паросочетание** (maximal matching) — это паросочетание M в графе G, содержащее максимальное количество ребер.

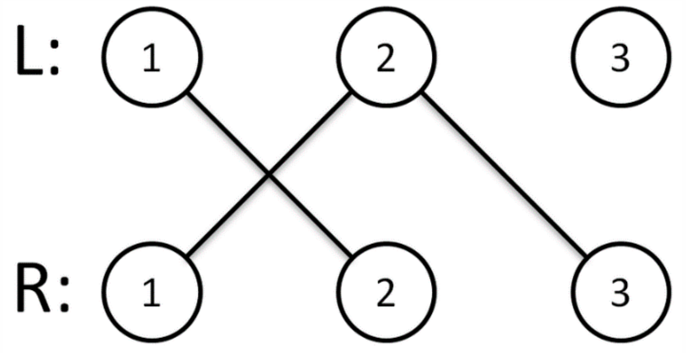
Определение. Паросочетание M графа G называется **совершенным** (или полным) (perfect matching), если оно покрывает все вершины графа.

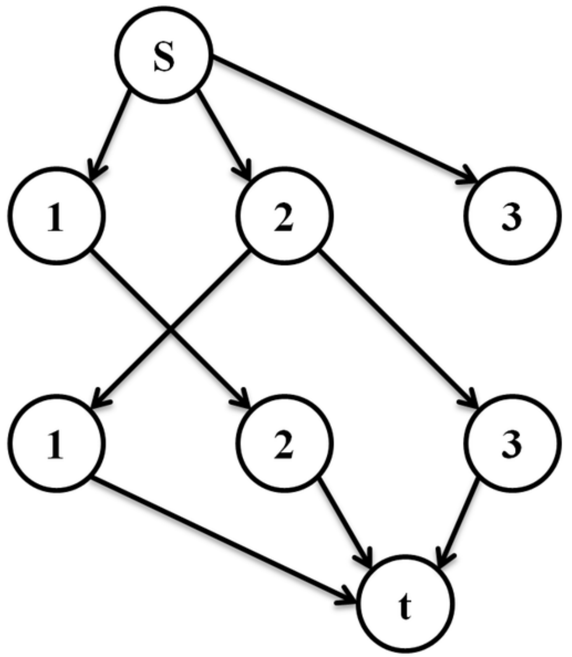
Алгоритм Форда-Фалкерсона для поиска максимального паросочетания

Построим сеть по двудольному графу.

Ребрам исходного графа назначим пропускную способность = 1.

Добавим сток, исток. Соединим ребрами пропускной способности 1.





Алгоритм Форда-Фалкерсона для поиска максимального паросочетания

Насыщенные ребра максимального потока = максимальное паросочетание.

Поиск увеличивающего пути – dfs.

<u>Оценка.</u>

Поиск в глубину запускается от вершины s не более чем n раз, т.к. размер паросочетания не может быть больше n, а каждый увеличивающий путь увеличивает паросочетание на 1.

Сам dfs работает за O(n+m), каждое инвертирование и перезапись паросочетания так же занимает O(n) времени. Тогда все время алгоритма ограничено O(n (n + m)).

Цепи и алгоритм Куна

Цепи

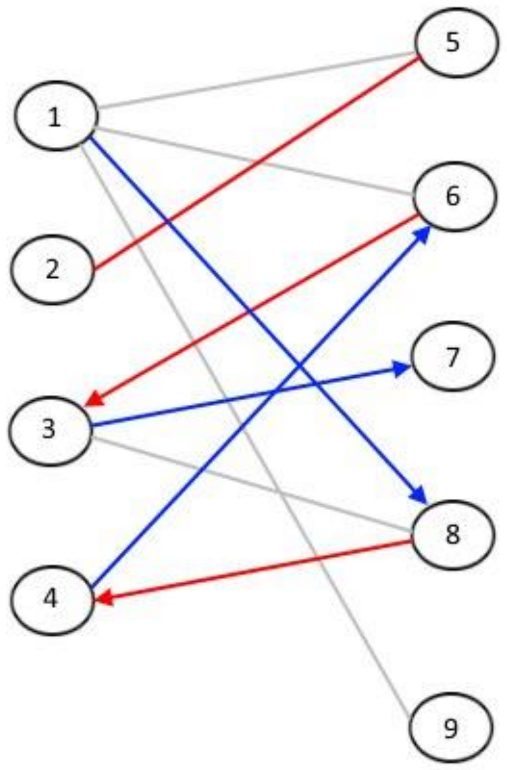
Определение. **Чередующаяся цепь** (alternating path) — путь в двудольном графе, для любых двух соседних рёбер которого верно, что одно из них принадлежит паросочетанию M, а другое нет.

Определение. **Дополняющая цепь** (или увеличивающая цепь, augmenting path) — чередующаяся цепь, у которой оба конца свободны.

Можно аналогично определить уменьшающую и сбалансированную цепь.

Дополняющая цепь

Красные рёбра принадлежат паросочетанию M, а синие не принадлежат.
Тогда чередующаяся цепь: 1–8–4–6–3–7.



Дополняющая цепь VS максимальное паросочетание

Теорема Бержа (о максимальном паросочетании и дополняющих цепях). Паросочетание M в двудольном графе G является максимальным тогда и только тогда, когда в G нет дополняющей цепи.

Доказательство. =>. От противного. Пусть существует дополняющая цепь. Инвертируем все ребра в ней – получим бОльшее паросочетание.

<=. От противного. Пусть M – не наибольшее. Пусть M' – другое паросочетание, $|M'| > |M|$. Рассмотрим их симметрическую разницу. В таком графе все вершины имеют степень не больше 2. Найдется компонента, в которой ребер M' больше. Это будет увеличивающий путь для M.

Поиск дополняющей цепи

Обход в глубину.

Изначально стоим в текущей ненасыщенной вершине v левой доли.

Просматриваем все рёбра из этой вершины, пусть текущее ребро — это ребро (v, to).

Если вершина to ещё не насыщена паросочетанием, то, значит, мы смогли найти увеличивающую цепь: она состоит из единственного ребра (v, to); в включаем это ребро в паросочетание и прекращаем поиск.

Иначе, — если to уже насыщена каким-то ребром (p, to), то попытаемся пройти вдоль этого ребра: тем самым мы попробуем найти увеличивающую цепь, проходящую через рёбра (v, to), (to, p). Для этого просто перейдём в нашем обходе в вершину p — теперь мы уже пробуем найти увеличивающую цепь из этой вершины.

Алгоритм Куна

Идея: Возьмём пустое паросочетание, а потом — пока в графе удаётся найти увеличивающую цепь, — будем выполнять чередование паросочетания вдоль этой цепи, и повторять процесс поиска увеличивающей цепи.

Как только такую цепь найти не удалось — процесс останавливаем, — текущее паросочетание и есть максимальное.

Увеличивающую цепь можем искать обходом в глубину из каждой вершины.

Но так асимптотика будет $O(n^2(n + m))$

Алгоритм Куна

Можно запустить поиск дополняющего пути из каждой вершины левой доли по одному разу.

Оказывается: если для некоторой вершины поиск увеличивающего пути не дал результата, то и в другой раз после обхода других вершин также не даст.

Алгоритм Куна = Поиск дополняющего пути из каждой вершины левой доли по одному разу с сохранением информации о посещенных вершинах dfs, не участвующих в дополняющем пути.

Алгоритм Куна. Оценка

L запусков обхода в глубину.

Сохранение информации о посещенных вершинах dfs позволяет экономить проходы.

Всего этот алгоритм исполняется за время $O\big(M \cdot (n + m)\big)$, где M – размер максимального паросочетания.

Спасибо за внимание!

?