МГТУ им. Н.Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

Лекция 5. Графы 1.

Алгоритмы и структуры данных

Мацкевич С.Е.

План лекции 5 «Графы 1»

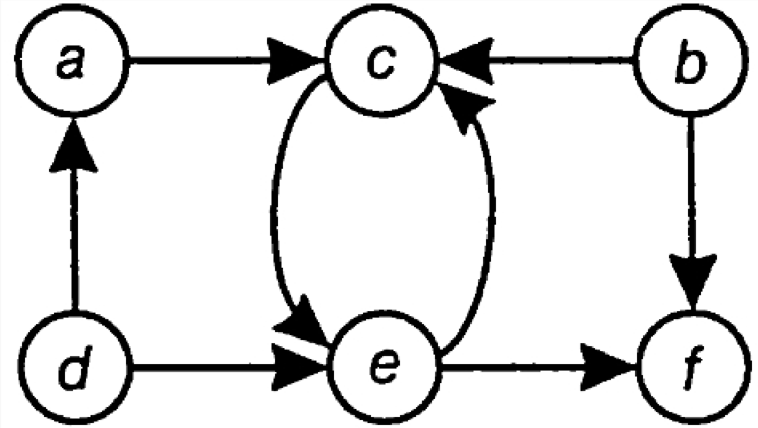
1. Терминология.
2. Обход в глубину.
3. Времена входа-выхода. Лемма о белых путях.
4. Поиск циклов.
5. Проверка связности.
6. Топологическая сортировка.
7. Поиск сильносвязных компонент. Алгоритм Косарайю.
8. Обход в ширину.

Терминология. Ор.граф.

<u>Определение.</u> $G = (V, E)$, где $E \subset V \times V$, называется *ориентированным графом*.
V – множество *вершин*,
E – множество *ребер*.

<u>Определение.</u> *Петля* – ребро (v, v).

<u>Определение.</u> Пусть (v, u) – ребро.
Тогда v – *предок* u,
v и u – *смежные*,
v *инцидентна* (v, u),
u *инцидентна* (v, u).



Терминология. Псевдограф.

Определение. $G = (V, E, beg, end)$,
где V, E – множества,
$beg: E \to V$,
$end: E \to V$,
называется *псевдографом*.

В псевдографе возможны кратные ребра. В том числе кратные петли.

Определение. *Полный граф* – граф, в котором E содержит всё $V \times V$ без петель.

Терминология. Неор.граф.

Определение. $G = (V, E)$, где $E \subset \{\{v, u\}, v, u \in V\}$,
называется *неориентированным графом*.
V – множество *вершин*,
E – множество *ребер*.

Определение. $G = (V, E, ends)$,
где V, E – множества,
$ends: E \to \{\{v, u\}, v, u \in V\}$,
называется *неориентированным псевдографом*.

Степень вершины

Определение. *Степень вершины* $\deg v$ – число ребер, инцидентных v, причем петля добавляет степень 2.

Лемма. $\sum_{v \in V} \deg v = 2|E|$.

Доказательство. Индукция по числу ребер.

Следствие 1. Число вершин нечетной степени – четно.

Следствие 2. Число ребер в полном графе равно

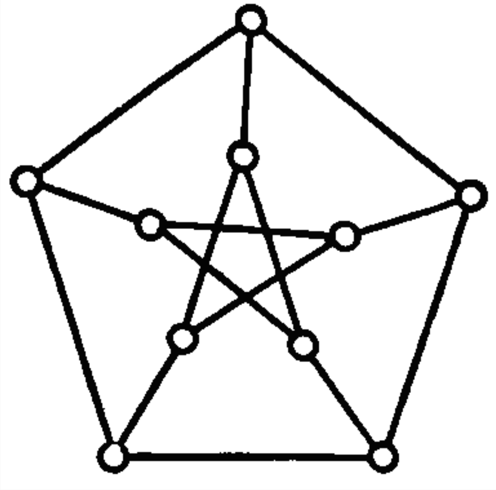
$$\frac{|V||V-1|}{2}.$$

Регулярный граф

<u>Определение.</u> *Регулярный граф* – граф, в котором степени всех его вершин равны.

В таком графе

$$|E| = \frac{k|V|}{2}.$$



Пути и циклы

Определение. *Путь* – последовательность
$$v_0, e_1, v_1, e_2, v_2, \dots, e_k, v_k,$$
где $e_i \in E$, $v_i \in V$, $e_i = (v_{i-1}, v_i)$,
k – *длина* пути.

Определение. *Циклический путь* (*цикл*) в ориентированном графе – путь, в котором $v_0 = v_k$.

Определение. *Циклический путь* (*цикл*) в неориентированном графе – путь,
в котором $v_0 = v_k$ и $e_i \neq e_{i+1}$, $e_1 \neq e_k$.

Простые пути и циклы

Определение. *Простой* (*вершинно-простой*) *путь* – путь, в котором каждая из вершин встречается не более одного раза.

Определение. *Реберно-простой путь* – путь, в котором каждое ребро встречается не более одного раза.

Простой цикл и реберно-простой цикл определяются аналогично.

Связность

Определение. Вершины u и v в неориентированном графе *связны*, если существует путь $u \rightsquigarrow v$.

Теорема. Связность – отношение эквивалентности.

Доказательство. Надо доказать:
Рефлексивность… (очевидно)
Симметричность… (очевидно)
Транзитивность… (очевидно) ☺

Определение. *Компонента связности* – класс эквивалентности отношения связности.

Определение. Неориентированный граф – *связен*, если он состоит из одной компоненты связности.

Слабая связность

Определение. Вершины u и v в ориентированном графе G *слабо связны*, если они связны в графе G', полученном из G удалением ориентации ребер и повторяющихся ребер.

Следствие. Слабая связность – отношение эквивалентности.

Определение. *Компонента слабой связности* – класс эквивалентности отношения слабой связности.

Сильная связность

Определение. Вершины u и v в ориентированном графе G *сильно связны*, если существуют пути $u \rightsquigarrow v$ и $v \rightsquigarrow u$.

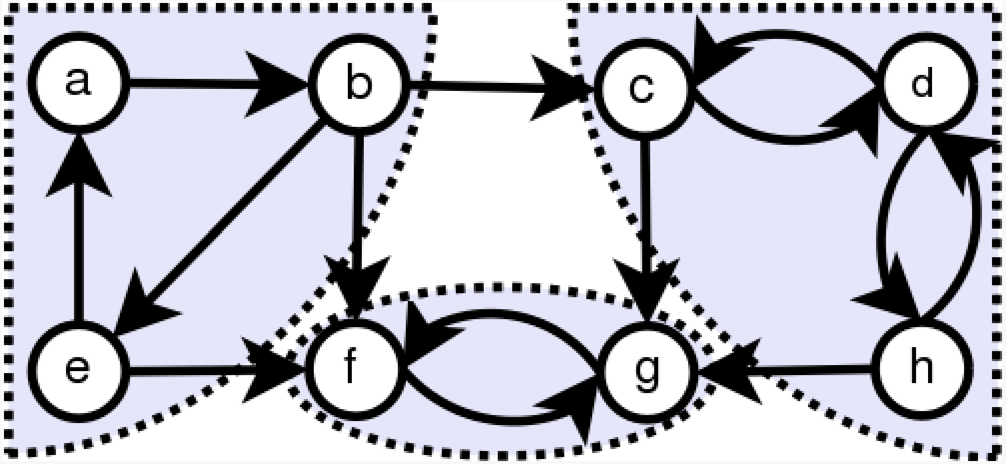
Теорема. Сильная связность – отношение эквивалентности.

Определение. *Компонента сильной связности* (*КСС*) – класс эквивалентности отношения сильной связности.

Определение. Ориентированный граф – *сильно связен*, если он состоит из одной компоненты сильной связности.

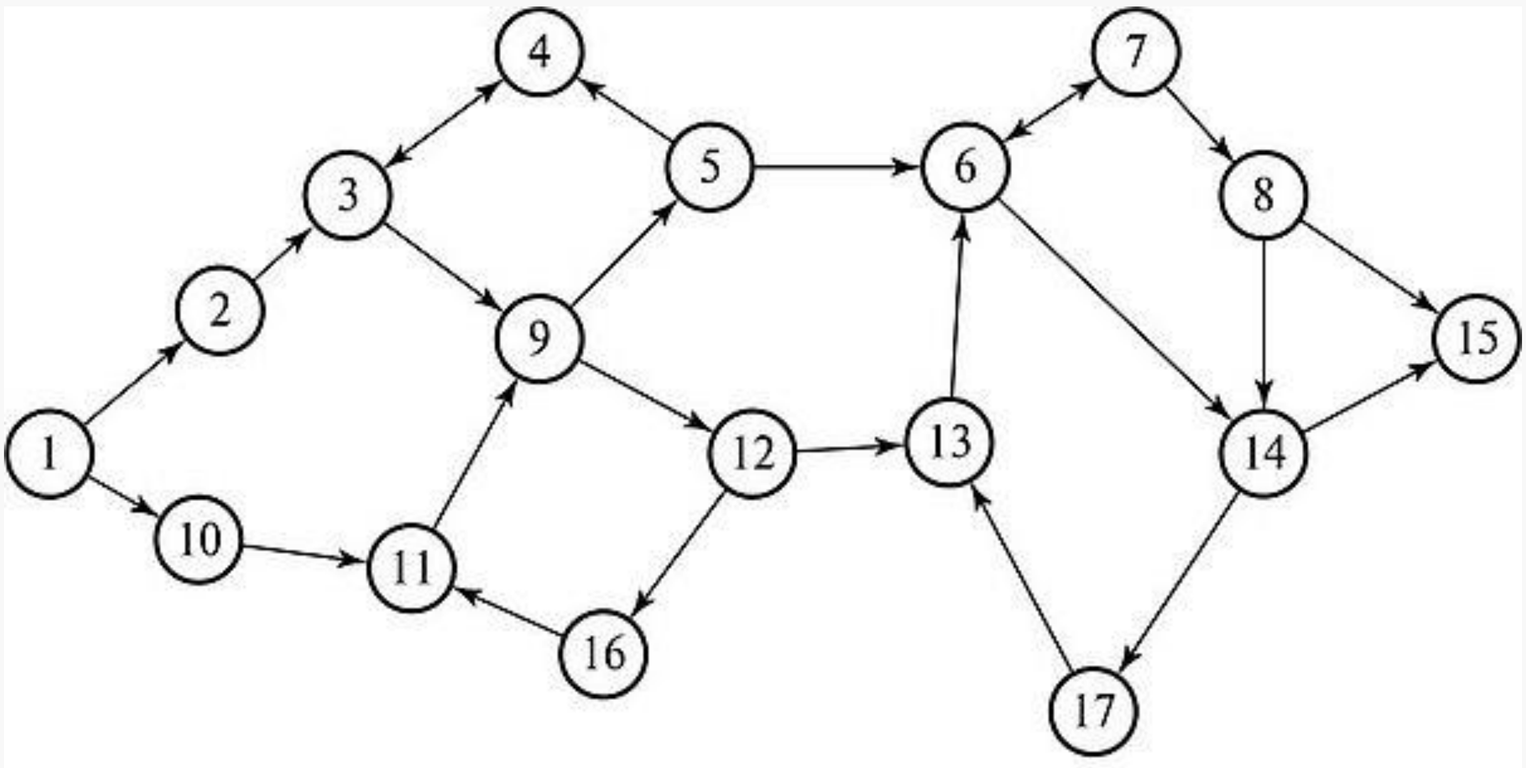
КСС

Интересно искать сильно связные компоненты графа.



КСС

Сколько КСС в этом графе?



Обход в глубину

DFS – Depth First Search – обход ориентированного или неориентированного графа, при котором рекурсивно обходятся все вершины, достижимые из текущей вершины.

1) Выбираем непосещенную вершину u.
2) Запускаем dfs(u):
   - Помечаем u,
   - Запускаем dfs(v) для всех $(u, v) \in E$.
3) Повторяем 1) и 2) пока есть непосещенные вершины.

Обход в глубину

```
vector<bool> visited;
void dfs( int u ) {
    visited[u] = true;
    for( v: (u, v) ∈ E )
        if( !visited[v] )
            dfs( v );
}
int MainDFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            dfs( i );
}
```

Цвета вершин

Белая – в ней еще не были.

Серая – проходимся текущим вызовом dfs.

Черная – пройдена, итерации завершены.

Подграф предшествования.

$G_P = (V, E_P)$, где $E_P = \{(p[u], u)\}$, $p[u]$ – вершина, от которой был вызван dfs(u).

G_P – лес обхода в глубину, состоящий из нескольких деревьев.

Типы ребер графа относительно dfs

1. **<u>Ребра дерева</u>** $\in G_P$.
2. Ребра (u, v), соединяющие u с предком v – **<u>обратные</u>** ребра.
   В неориентированном графе предок – не родитель.
3. Ребра (u, v), соединяющие u с потомком v – **<u>прямые</u>** ребра.
   В неориентированном графе нет разницы между прямыми и обратными.
4. Остальные ребра (u, v) – **<u>перекрестные</u>** ребра.

Переход в белую вершину в dfs – ребро дерева.

Переход в серую вершину в dfs – обратное ребро.

Переход в черную вершину в dfs – прямое или перекрестное.

Последние можно различить по времени входа и выхода.

Времена входа и выхода

```
dfs( u ) {
    entry[u] = time++;
    … // обработка вершины.
    leave[u] = time++;
}
```

В дереве dfs вершина u – предок v титтк

$$entry[u] < entry[v] \quad \text{и} \quad leave[u] > leave[v].$$

Простая лемма

Лемма. Не существует момента поиска в глубину такого, в котором существует ребро из черной вершины в белую.

Доказательство. От противного. Пусть такое ребро (u, v) и момент *time* существуют.

Рассмотрим момент *leave[u]*. Этот момент – первый, в котором вершина u – черная. Т.е.

$$leave[u] \leq time.$$

Следовательно, вершина v в момент *leave[u]* – белая, т.к. она белая в момент *time*.

Но это означается, что на момент выхода из вершины u есть необработанное ребро (u, v). Противоречие.

Лемма о белых путях

Лемма (о белых путях). Пусть есть некоторый обход dfs в графе G.

entry[u] и *leave[u]* – моменты входа и выхода из вершины u. Тогда между этими моментами:

1. Черные и серые вершины $G\backslash u$ не поменяют свой цвет.
2. Белые вершины $G\backslash u$ либо останутся белыми, либо станут черными. Причем черными станут те, которые были достижимы из u по белым путям и только они.

Доказательство. Черная вершина останется черной.

Серая вершина останется серой, т.к. находится в стеке рекурсии.

Достижимая белая вершина станет черной. Иначе на пути к ней в момент *leave[u]* будет ребро из некоторой черной вершины в некоторую белую.

Если вершина стала черной, значит, она была достижима из u по белому пути.

Проверка наличия циклов

Задача. Есть ориентированный или неориентированный граф G. Проверить наличие циклов в графе и, если циклы есть, найти какой-нибудь цикл.

Решение. Если в некоторый момент некоторого обхода dfs нашли обратное ребро (ведущее из текущей вершины в серую), то цикл существует. Иначе цикла нет.

Доказательство. =>. Нашли серую вершину, следовательно, цикл есть.

<=. Если есть цикл C, то пусть v – первая вершина C, обрабатываемая dfs. Рассмотрим entry[v]. В этот момент все остальные вершины цикла C – белые. Существует ребро (u, v) цикла C. Из v в u есть белый путь по циклу C. По лемме о белых путях вершина u станет черной до выхода из v. Следовательно, в момент entry[u] вершина v будет серой.

Для неориентированного графа рассуждение аналогичное, но важно наличие как минимум двух ребер цикла (u_1, v) и (u_2, v).

Поиск цикла

Время работы проверки наличия цикла – O(V + E).

Задача. Найти какой-нибудь цикл в графе с циклами.

Решение.

Пусть в момент time в dfs была найден переход (u, v) в серую вершину v.

Цикл восстанавливается по предкам (стек вызовов в момент *entry[u]*):

$$v, u, p[u], p\left[p[u]\right], \ldots, v.$$

Проверка связности

Задача. Проверить, является ли неориентированный граф G связным.

Решение.

Запустим dfs(v).

Если после выхода все вершины посетили ⇔ связность.

После выхода из dfs(v) все вершины можно не проверять на visited[u], если использовать переменную для подсчета числа обработанных вершин.

Время работы $O(V + E)$.

Топологическая сортировка

Определение. *Топологическая сортировка* ациклического графа $G = (V, E)$ – такое упорядочивание всех вершин V, что если $(u, v) \in E$, то u располагается до v.
Формально: упорядочивание

$$\phi: V \to \{1, \dots, n\}, \phi(u) < \phi(v).$$

Топологическая сортировка для графов с циклами невозможна.

Топологическая сортировка

Теорема. Пусть G – ациклический ориентированный граф. Тогда существует топологическая сортировка, т.е.

$$\exists \phi: V \to \{1, \dots, n\},$$

т.ч. для любого $(u, v) \in E$: $\phi(u) < \phi(v)$.

Доказательство. Определим

$$\phi(v) = |V| + 1 - leave[v].$$

$entry[v]$ не считаем, только $leave[v]$, начиная с 1.

Докажем, что для так определенного $\phi(v)$ выполняется $\phi(u) < \phi(v)$ для любого $(u, v) \in E$.
Рассмотрим момент $entry[u]$. v – не серая, т.к. нет циклов.

1) v – белая. Тогда она будет обработана внутри dfs(u).
2) v – черная. Значит, она уже обработана.

Топологическая сортировка

Алгоритм топологической сортировки напрашивается из теоремы:

- Запустить DFS, считать $leave$.
- $\phi(v) = |V| + 1 - leave[v]$

Время работы $T = O(V + E)$.

Алгоритм Косарайю

Алгоритм Косарайю (1978г) – алгоритм поиска сильно связных компонент.

Пусть $G = (V, E)$.

Алгоритм:

1) Построим $H = G^T$ – граф, являющийся инвертированным к G.
2) DFS(H)
3) DFS(G), перебирая вершины в MainDFS в порядке убывания $leave_H$.

Деревья, полученные запусками dfs на шаге 3 – компоненты сильной связности.

Алгоритм Косарайю

Теорема. Деревья, полученные в п. 3 алгоритма Косарайю, – КСС.

Доказательство. =>

Пусть вершины s и t из одной КСС.

Тогда существуют пути $s \rightsquigarrow t$ и $t \rightsquigarrow s$. Значит, вершины s и t попадут в одно дерево в шаге 3).

Это следует из следующего рассуждения:

Пусть v – первая вершина из цикла $s \rightsquigarrow t \rightsquigarrow s$ в обходе DFS(G). Тогда в момент $entry[v]$ вершины s и t достижимы из v по белым путям. По лемме о бп они будут обработаны в dfs(v).

Алгоритм Косарайю

Пусть C – КСС. Обозначим leave[C] – максимальное время выхода leave[v], $v \in C$.

Лемма. Пусть C, C' – две различные КСС, и есть ребро (u, v) между ними, $u \in C, v \in C'$. Тогда $leave[C] > leave[C']$.

Доказательство леммы. а) Первой была достигнута КСС C – вершина *w*. Тогда в момент входа в C вся компонента C' – белая и достижима из C. По лемме о белых путях в момент $leave[w] > leave[C']$.

б) Первой была достигнута КСС C'. В этом случае вся компонента C' будет пройдена до обхода C, т.к. не существует пути из C' в C. То есть $leave[C] > leave[C']$.

Алгоритм Косарайю

Продолжение доказательства теоремы. <=

Рассмотрим дерево T – дерево обхода dfs на этапе 3.

Докажем, что T – компонента сильной связности.

Пусть T содержит два или более различных КСС и пусть C – первая КСС в обходе dfs. Существуют ребро (v, u), пройденное dfs. $v \in C$ и $u \in C_2$. C, C_2 – различные КСС.

$leave_H[C] > leave_H[C_2]$ по построению T.

Но по лемме $leave_H[C] < leave_H[C_2]$, т.к. ребро $(u, v) \in H$.

Противоречие.

Обход в ширину

BFS – Breadth First Search – обход в ширину.

Обход, при котором вершины обходятся в порядке увеличения расстояния от стартовой вершины.

Обход, при котором вершины обходятся «по слоям».

Как и в обходе в ширину деревьев используется очередь.

Обход в ширину

```
vector<bool> visited;
void bfs( int u ) {
    std::queue<int> q;
    q.push( u ); visited[q] = true;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( !visited[w] )
                q.push( w );
    }
}
int MainBFS() {
    visited.assign( n, false );
    for( int i = 0; i < n; ++i )
        if( !visited[i] )
            bfs( i );
}
```

Поиск кратчайших путей

```
vector<int> r, pi;
void bfs( int s ) {
    std::queue<int> q;
    q.push( s ); r[s] = 0; pi[s] = -1;
    while( !q.empty() ) {
        v = q.front(); q.pop();
        for( w: (v, w) ∈ E )
            if( r[w] > r[v] + 1 ) {
                r[w] = r[v] + 1;
                pi[w] = v;
                q.push( w );
            }
    }
}
```

Реберная двусвязность

Определение. Вершины u и v в неориентированном графе G реберно двусвязны, если между этими вершинами существуют два реберно непересекающихся пути.

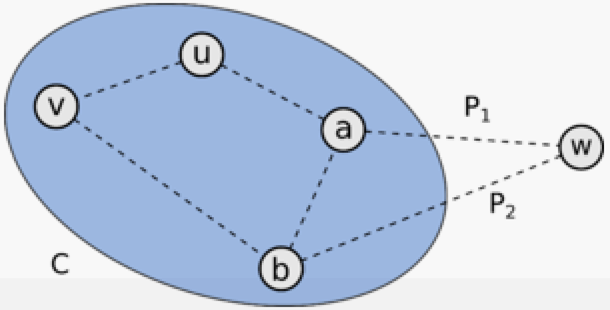
Утверждение. Реберная двусвязность – отношение эквивалентности.

Реберная двусвязность

Д-во. Транзитивность: u с v, v с w. Докажем реберную двусвязность u и w.

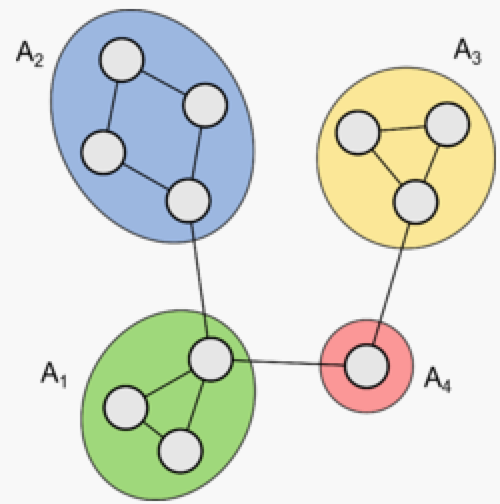
Пусть из w в v есть два реберно непересекающихся пути P1 и P2 соответственно. Обозначим за C объединение реберно непересекающихся путей из u в v. C – реберно простой цикл. Пусть вершины a и b – первые со стороны w вершины на пересечении P1 и P2 с C соответственно.

Рассмотрим два пути wau и wbu, такие, что части au и bu идут в разные стороны по циклу C. Наличие двух таких рёберно непересекающихся путей очевидно, а значит u и w рёберно двусвязны.



Реберная двусвязность

Следствие. Вершины неориентированного графа разбиваются на компоненты реберной двусвязности.



Мост

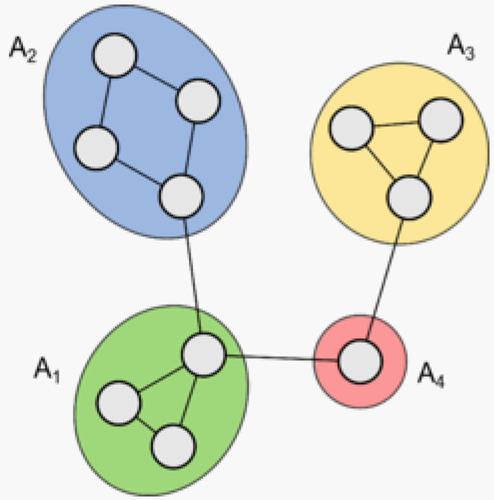
Определение 1. Мост – ребро, связывающее две различные компоненты реберной двусвязности.

Определение 2. Мост – ребро, при удалении которого компонента связности распадается.

Определение 3. Мост – ребро (u, v), лежащее на любом пути, соединяющем u и v.

Утверждение. Все три определения эквивалентны.

Доказывается 1 => 2 => 3 => 1.



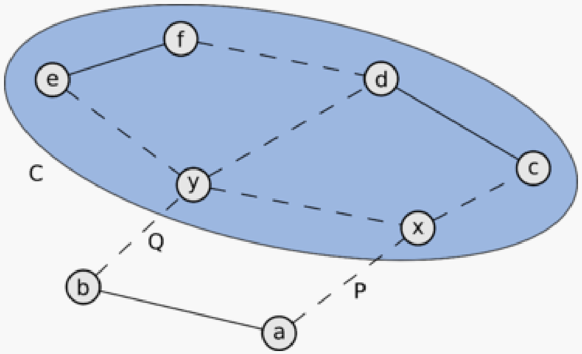
Вершинная двусвязность

Определение. Два ребра в неориентированном графе G вершинно двусвязны, если существуют вершинно непересекающиеся пути, соединяющие их концы.

Утверждение. Вершинная двусвязность – отношение эквивалентности на ребрах.

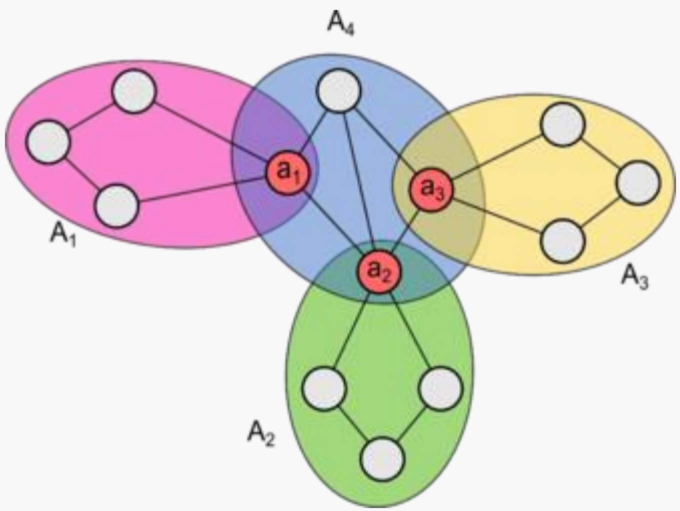
Вершинная двусвязность

Д-во. Транзитивность: Пусть имеем ребра: ef вершинно двусвязно с cd, cd вершинно двусвязно с ab, при этом все они различны. Ребра ef и cd лежат на вершинно простом цикле C. Будем считать, что существуют непересекающиеся пути P : a –> c, Q : b –> d (ситуация, когда они идут наоборот, разбирается аналогично). Пусть x — первая вершина на P, лежащая также на C, y — первая вершина на Q, лежащая на C. Проделав пути от a до x и от b до y, далее пойдем по циклу C в нужные (различные) стороны, чтобы достичь e и f. То есть ef вершинно двусвязно с ab.



Вершинная двусвязность

Следствие. Ребра неориентированного графа разбиваются на компоненты вершинной двусвязности.

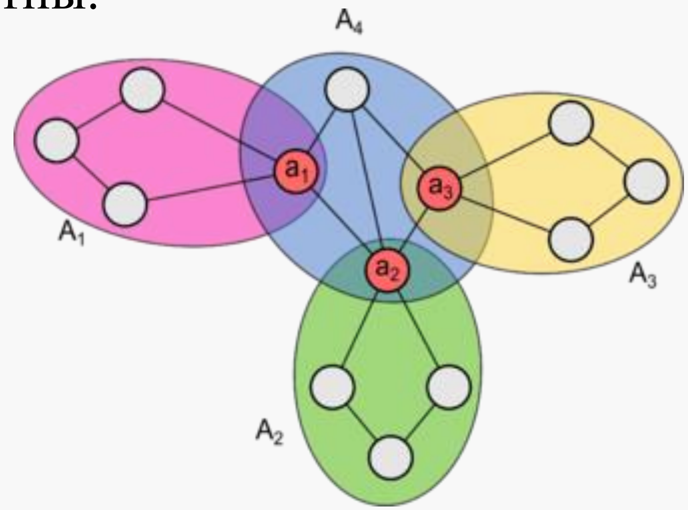


Точка сочленения

Определение 1. Точка сочленения – вершина, при удалении которой вместе с инцидентными ей ребрами, компонента связности распадается.

Определение 2. Точка сочленения – вершина, принадлежащая двум или более компонентам вершинной двусвязности.

Утверждение. Определения эквивалентны.



Поиск мостов

Утверждение. Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину v или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является.

Д-во. В самом деле, мы этим условием проверяем, нет ли другого пути из v в to, кроме как спуск по ребру (v, to) дерева обхода в глубину.

Поиск мостов

```
void dfs( int v, int p = -1 ) {
    used[v] = true;
    entry[v] = lowest[v] = time++;
    for( int i = 0; i < g[v].size(); ++i ) {
        int to = g[v][i];
        if( to == p )
            continue;
        if( used[to] )
            lowest[v] = min( lowest[v], entry[to] );
        else {
            dfs( to, v );
            lowest[v] = min( lowest[v], lowest[to]);
            if( lowest[to] > entry[v] )
                IS_BRIDGE( v, to );
        }
    }
}
void find_bridges() {
    time = 0;
    for( int i = 0; i < n; ++i )
        used[i] = false;
    for( int i = 0; i < n; ++I )
        if( !used[i] )
            dfs( i );
}
```

Поиск точек сочленения

Утверждение. Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины v. v – не первая вершина обхода dfs. Тогда, если из вершины v и из любого её потомка в дереве обхода в глубину нет обратного ребра в предка v, то эта вершина является точкой сочленения. В противном случае она не является точкой сочленения.

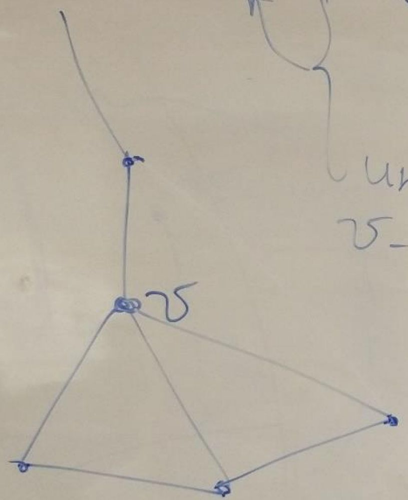
Если v – первая вершина обхода dfs, то она является точкой сочленения тогда и только тогда, когда после обхода dfs из первой связанной с v существуют непосещенные вершины среди связанных с v.

v – не первая в dfs

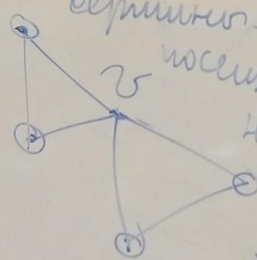
если в момент leave[v]

lowest[v] < entry[v], то v – не явл. т.с.

иначе v – т.с.



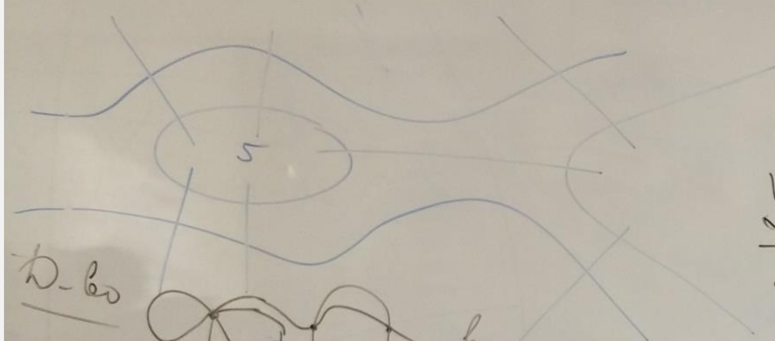
v – первая в dfs: После обработки одной вершины – проверить, посещены ли остальные u: $(v,u) \in E$



В этом случае v – не явл. т.с.

иначе v – т.с.

Эйлеровы графы.



G – неор. граф

Эйлеров путь – реберно-простой путь, сод. все рёбра G

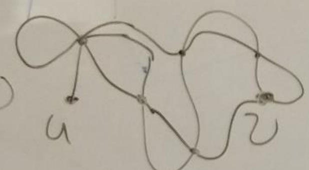
Эйлеров цикл – аналогичен

Утв. G – связный неор. граф.

- Эйлеров путь сущест. $\Leftrightarrow$ в G не более 2х нечётных вершин.
- Эйлеров цикл существует $\Leftrightarrow$ в G все вершины – чётные.

Д-во

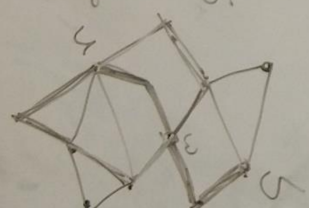
$\Rightarrow$



все промеж. вершины – чётные

$u == v \Leftrightarrow u, v$ – чётные

$\Leftarrow$ $u \neq v$ – нечётные. Построим простой путь из u в v.



$u \rightsquigarrow w \overset{\text{цикл}}{\rightsquigarrow} w \rightsquigarrow v$

МГТУ
им. Н. Э. Баумана
ТЕХНОПАРК
Mail.Ru Group

???

Спасибо за внимание!