**Алгоритмы на графах**

Полезные сайты и книги:

[Дасгупта Пападимитриу Вазирани](https://docviewer.yandex.ru/view/841867124/?page=1&*=%3D%3D&lang=ru)

[Кормен Лейзерсон Чарльз](https://docviewer.yandex.ru/view/841867124/?*=qz%%3D)

[E-Maxx](http://e-maxx.ru/algo)

[Конспекты ИТМО](https://neerc.ifmo.ru/wiki/index.php?title=%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)

[Олимпиады по информатике](http://codeforces.com/)

[Ерусалимский](http://en.bookfi.net/book/468036)

[Зыков](https://docviewer.yandex.ru/view/841867124/?*=%3D&lang=ru)

[Харари](https://drive.google.com/file/d/0B65Vi6UDbVBTZVBiMFhGNTVTWVk/view)

[Карпов](https://docviewer.yandex.ru/view/841867124/?page=1&*=&lang=ru)

Оглавление

[Лекция 1. Вводная лекция. 3](#_Toc31565113)

[Лекция 2. Представление графов. Поиск в ширину. 3](#_Toc31565114)

[Лекция 3. Поиск в глубину 5](#_Toc31565115)

[Схема поиска в глубину. 5](#_Toc31565116)

[Классификация дуг. Дерево поиска в глубину. 5](#_Toc31565117)

[Проверка графа на двудольность. 5](#_Toc31565118)

[Ацикличность графа. 6](#_Toc31565119)

[Поиск эйлерова цикла. 7](#_Toc31565120)

[Лекция 4. Поиск в глубину (продолжение) 7](#_Toc31565121)

[Поиск наименьшего общего предка (lca). 7](#_Toc31565122)

[Поиск мостов в неориентированном графе 8](#_Toc31565123)

[Правильная нумерация и топологическая сортировка 9](#_Toc31565124)

[Поиск компонент сильной связности. 10](#_Toc31565125)

[Второй алгоритм поиска мостов. 11](#_Toc31565126)

[Лекция 5. Ориентированные ациклические графы 11](#_Toc31565127)

[Алгоритм поиска кратчайших путей от заданной вершины до всех остальных на ациклическом графе. 11](#_Toc31565128)

[Типовые алгоритмы на ациклических графах. 11](#_Toc31565129)

[Лекция 6. Остовные деревья 12](#_Toc31565130)

[Теорема Кирхгофа 12](#_Toc31565131)

[Структуры данных куча и система непересекающихся множеств 12](#_Toc31565132)

[Алгоритм Прима 12](#_Toc31565133)

[Алгоритм Краскала 13](#_Toc31565134)

[Лекция 7. Кратчайшие расстояния 13](#_Toc31565135)

[Алгоритм Беллмана-Форда. 13](#_Toc31565136)

[Алгоритм Дейкстры 13](#_Toc31565137)

[Алгоритм А\* 14](#_Toc31565138)

[Лекция 8. Матрицы кратчайших расстояний 14](#_Toc31565139)

[Алгоритм «перемножения матриц» 14](#_Toc31565140)

[Алгоритм Флойда-Уоршала 14](#_Toc31565141)

[Алгоритм Джонсона 14](#_Toc31565142)

[Лекция 9. Потоки в сетях 14](#_Toc31565143)

[Лекция 10. Алгоритмы Эдмондса-Карпа и Диница 14](#_Toc31565144)

[Лекция 11. Паросочетания. 14](#_Toc31565145)

# Лекция 1. Вводная лекция.

Понятия из теории графов.

**Граф** — это абстрактное представление множества объектов и связей между ними.

G(V,E). **Вершины** (*vertex*), **ребра** (*edge*), **дуги** (*directed edge*). G(V,E,f), **мультиграф**, **квазиграф**, **полный граф**

**Степень вершины**. Теорема: сумма степеней всех вершин равна удвоенному количеством ребер (доказывается по индукции по количеству ребер). Терема о рукопожатиях: количество вершин нечетной степени – четно (доказательство: сумма степеней всех вершин = сумма степеней вершин четной степени + сумма степеней вершин нечетной степени = удвоенное количество ребер).

(сумма степеней всех вершин четна)

**Висячая вершина**, **изолированная вершина**

**Разреженный** (*sparse*) |E|<<|V|2, **плотный** (*dense*)

**Представления графов**:

* **Матрица смежности** (*adjacency-matrix representation*)
* **Список смежности** (*…*)
* **Матрица инцидентности** (…)
* **Список дуг** (*…*)

**Геометрический граф**

**Пути, цепи, циклы, контуры**

**Эйлеров путь, Эйлеров цикл, Эйлеров граф**

**Дерево, Теорема о соотношении между количеством вершин и количеством ребер дерева (|V|=|E|+1).**

# Лекция 2. Обход в ширину.

Обход в ширину (*Breadth-First Search*) относится к, так называемым, волновым алгоритмам. Их суть можно понять, представляя круги на воде появляющиеся от брошенного камня.

BFS (G, s)

1. **for** (для) каждой u ∈ V[G]\{s} **do**
2. color[u] ⟵ WHITE
3. d[u] ⟵ ∞
4. π[u] ⟵ NIL
5. color[s] ⟵ GRAY
6. d[s] ⟵ 0
7. π[s] ⟵ NIL
8. Q ⟵ ∅
9. В очередь(Q, s)
10. **while** Q ≠ ∅ **do**
11. u ⟵ из очереди(Q)
12. **for** (для) каждой v ∈ Adj[u] **do**
13. **If** color[v] = WHITE **than**
14. color[v] ⟵ GRAY
15. d[v] ⟵ d[u]+1
16. π[v] ⟵ u
17. В очередь(Q, v)
18. color[u] ⟵ BLACK

К задачам, при решении которых можно использовать обход в ширину можно отнести:

* Задача нахождения **кратчайшего пути** в невзвешенном графе от вершины, из которой делался обход до произвольной вершины графа.

Любой искомый путь можно получить, двигаясь от вершины по предкам, полученным обходом в ширину.

PATH (v, π)

1. **if** color[v] = WHITE **then**
2. Нет пути
3. **else**
4. u ⟵ v
5. **while** u != NIL **do**
6. В стек (path, u)
7. u ⟵ π[u]

* Задача проверки **связности** графа

Если первый обход не оставил не посещенных вершин

* Задача определения **компонент связности**

Запускается обход в ширину от каждой не посещенной вершины. Каждый отдельный запуск находит отдельную компоненту.

* Задача нахождения **кратчайшего цикла** в ориентированном невзвешенном графе.

Обход в ширину запускается для каждой вершины графа. Для каждого обхода сохраняется первый найденный цикл. Наименьший из этих циклов и будет кротчайшим.

* Задача нахождения решения какой-либо задачи (игры) **с наименьшим числом ходов**, если каждое состояние системы можно представить вершиной графа, а переходы из одного состояния в другое — рёбрами графа.
* Задача нахождения кратчайшего пути в **0-1-графе**
* Задача нахождения всех рёбер графа, лежащих **на каком-либо кратчайшем пути** между заданной парой вершин A и B.
* Задача нахождения всех вершин графа, лежащих **на каком-либо кратчайшем пути** между заданной парой вершин A и B.
* Задача нахождения **кратчайшего чётного пути** в графе (т.е. пути чётной длины).

# Лекция 3. Обход в глубину

## Схема обхода в глубину.

Сложность O(N+M).

DFS (G)

1. **for** (для) каждой u ∈ V[G] **do**
2. color[u] ⟵ WHITE
3. π[u] ⟵ NIL
4. time ⟵ 0
5. **for** (для) каждой u ∈ V[G] **do**
6. **if** color[u] = WHITE **then**
7. DFS\_Visit(u)

DFS\_Visit (u)

1. color[u] ⟵ GRAY
2. time ⟵ time +1
3. d[u] ⟵ time
4. **for** (для) каждой u ∈ Adj[u] **do**
5. **if** color[v] = WHITE **then**
6. π[v] ⟵ u
7. DFS\_Visit(v)
8. color[u] ⟵ BLACK
9. time ⟵time+1
10. f[u] ⟵ time

## Классификация дуг. Дерево обхода в глубину.

Введем понятие времени входа и выхода.

Теорема о скобках

Возможен только один из трех вариантов:

u\_start < v\_start < v\_finish < u\_finish

v\_start < u\_start <u\_finish < v\_finish

u\_start < u\_finish < v\_start < v\_finish

т.е. невозможны случаи:

u\_start < v\_start < u\_finish < v\_finish

v\_start < u\_start < v\_finish < u\_finish

Доказательство

Пусть u\_start < v\_start, значит, если u\_finish < v\_start – чтд

Поэтому рассмотрим оставшийся случай v\_start < u\_finish. Получается, что вершина v была открыта до закрытия вершины u. Значит исходя из алгоритма, вершина u является предком вершины v.

* Задача проверки является ли вершина u **предком** вершины v.

u\_start < v\_start < v\_finish < u\_finish

**Классификация дуг**:

**Ребра дерева** – это ребра графа Gπ. Ребро (u, v) является ребром дерева, если при исследовании этого ребра была открыта v (т.е. u является непосредственным предком дерева обхода).

**Прямые ребра** – это ребра (u, v), не являющиеся ребрами дерева обхода и соединяющие вершину u с ее потомком v в дереве обхода.

**Обратные ребра** – это ребра (u, v), соединяющие вершину u с ее предком v в дереве обхода. Ребра-циклы, которые могут встречаться в орграфах, рассматриваются как обратные ребра.

**Перекрестные ребра** – это ребра графа, не относящиеся к трем описанным выше типам. Они могут соединять вершины одного и того же дерева обхода, когда ни одна из вершин не является предком другой, или соединять вершины в различных деревьях.

Дуга (u,v) прямая, если u\_start < v\_start, v\_finish < u\_finish

Дуга (u,v) обратная, если v\_start < u\_start, u\_finish < v\_finish

Дуга (u,v) поперечная, если u\_finish < v\_start

## Проверка графа на двудольность.

**Признак двудольности:**

**Теорема**. Граф является двудольным тогда и только тогда, когда все его циклы имеют чётную длину.

**Алгоритм1**

В соответствии с теоремой ищутся все циклы.

Проверить граф на двудольность можно еще следующим алгоритмом:

**Алгоритм2**

Производится серия обходов в ширину (т.е. запускается обход в ширину из каждой непосещённой вершины). Вершина, из которой начинается обход, помещается в первую долю. Новая вершина в процессе обхода помещается в долю, отличную от доли текущей вершины. Если же обход направляется по ребру в вершину, которая уже посещена, то проверяется, находятся ли начало ребра и конец в разных долях. Если нет, то граф двудольным не является.

DFS (G)

1. **for** (для) каждой u ∈ V[G] **do**
2. part[u] ⟵ 0
3. **for** (для) каждой u ∈ V[G] **do**
4. **if** part[u] = 0 **then**
5. part[u] ⟵ 1
6. DFS\_Visit(u)

DFS\_Visit (u)

1. **for** (для) каждой u ∈ Adj[u] **do**
2. **if** part[v] = 0 **then**
3. part[v] ⟵ (part[u] +1) mod 2
4. **If** !(DFS\_Visit(v)) **then**
5. **return** false
6. **else**
7. **return** false
8. **return** true

## Ацикличность графа.

Если во время обхода в глубину очередной вершиной выбирается уже просмотренная или помеченная, то это означает наличие цикла.

DFS (G)

1. **for** (для) каждой u ∈ V[G] **do**
2. color[u] ⟵ WHITE
3. π[u] ⟵ NIL
4. Q ⟵ ∅
5. cycle\_start ⟵ NIL
6. cycle\_end ⟵ NIL
7. **for** (для) каждой u ∈ V[G] **do**
8. **if** color[u] = WHITE **then**
9. DFS\_Visit(u)
10. **If** cycle\_start = NIL **then**
11. **write** “acyclic”
12. **else**
13. **write** “cyclic”
14. в очередь (Q, cycle\_start)
15. **while** cycle\_start != cycle\_end **do**
16. cycle\_start ⟵ π[cycle\_start]
17. в очередь (Q, cycle\_start)

DFS\_Visit (u)

1. color[u] ⟵ GRAY
2. **for** (для) каждой u ∈ Adj[u] **do**
3. **if** color[v] = WHITE **then**
4. π[v] ⟵ u
5. **If** DFS\_Visit(v) **then**
6. **return** true
7. **else**
8. cycle\_start ⟵ v
9. cycle\_end ⟵ u
10. **return** true
11. color[u] ⟵ BLACK
12. **return** false

## Поиск эйлерова цикла.

**Теорема** Эйлеров цикл существует тогда и только тогда, когда степени всех вершин чётны.

**Доказательство (конструктивное):** Находятся множество не пересекающихся простых циклов охватывающих весь граф. После этого они «склеиваются».

**Теорема** Эйлеров путь существует тогда и только тогда, когда количество вершин с нечётными степенями равно двум или нулю (в случае существования эйлерова цикла).

**Доказательство:**

1. Если есть (две) вершины нечетной степени, то они соединяются временным ребром.
2. Находятся множество не пересекающихся простых циклов охватывающих весь граф. После этого они «склеиваются».
3. В полученном эйлеровом цикле удаляется временное ребро, после чего остается эйлеров путь начинающийся в одной из удаленных вершин, а заканчивающийся в другой.

# Лекция 4. Поиск в глубину (продолжение)

## Поиск наименьшего общего предка в дереве (lca).

Для нахождения наименьшего общего предка вершин **u** и **v** нужно пометить все вершины на пути от **u** к корню дерева обхода, а затем найти первую помеченную вершину на пути от вершины **v** к корню.

LCA (root, u, v)

1. **for** (для) каждой u ∈ V[G] **do**
2. color[u] ⟵ WHITE
3. a ⟵ u
4. **while** a!= root **do**
5. color[a] = BLACK
6. a ⟵ π[a]
7. color[a] = BLACK
8. a ⟵ v
9. **while** a! **do**
10. a ⟵ π[a]
11. **return** a

## Поиск мостов в неориентированном графе

Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины *v*. Рассмотрим ребро (*v*, *to*). Обозначим *A* – множество, состоящее из вершины *v* и ее предков, в дереве обхода в глубину. Обозначим *D* – множество, состоящее из вершины *to* и ее потомков, в дереве обхода в глубину. Если не существует ни одного обратного ребра, соединяющего вершину из *D* с вершиной из *A*, то ребро (*v*, *to*) – мост, иначе оно мостом не является.

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

Во время обхода в глубину будем вычислять следующую функцию:

Критерий: ребро (v,to) является мостом, тогда и только тогда, когда f(to) > tin(v).

FIND\_Bridges (G)

1. **for** (для) каждой u ∈ V[G] **do**
2. color[u] ⟵ WHITE
3. time ⟵ 0
4. Q ⟵ ∅
5. **for** (для) каждой u ∈ V[G] **do**
6. **if** color[u] = WHITE **then**
7. DFS\_Visit(u)

DFS\_Visit (u)

1. color[u] ⟵ GRAY
2. time ⟵ time+1
3. time\_in[u] ⟵ time
4. f[u] ⟵ time
5. **for** (для) каждой v ∈ Adj[u] **do**
6. **if** color[v] = WHITE **then**
7. DFS\_Visit(v)
8. f[v] = min (f[v], f[to]);
9. if (f[to] > tin[v])
10. bridges.push\_back(make\_pair(v,to));
11. **else**
12. f[v] = min (f[u], tin[v]);
13. color[u] ⟵ BLACK

void dfs (int v, int p = -1) {

used[v] = true;

tin[v] = f[v] = timer++; //init

for (int to : g[v]) {

if (to == p) continue;

if (used[to]) // back edge

f[v] = min (f[v], tin[to]);

else {

dfs (to, v); // tree edge

f[v] = min (f[v], f[to]);

if (f[to] > tin[v])

bridges.push\_back(make\_pair(v,to));

}

}

}

void find\_bridges() {

timer = 0;

used.assign(g.size(), false);

for (size\_t i=0; i<g.size(); ++i)

if (!used[i])

dfs(i);

}

Что будет если не учитывать, что to может быть предком?

## Правильная нумерация и топологическая сортировка

Нумерация вершин графа называется правильной, если любая дуга ведет из вершины с меньшим номером в вершину с большим номером.

Topological\_sort (G(V,E))

1. **for** (для) каждой u ∈ V[G] **do**
2. color[u] ⟵ WHITE
3. order[v] ⟵ 0
4. time ⟵ |V|
5. **for** (для) каждой u ∈ V[G] **do**
6. **if** color[u] = WHITE **then**
7. DFS\_Visit(u)

DFS\_Visit (u)

1. color[u] ⟵ BLACK
2. **for** (для) каждой v ∈ Adj[u] **do**
3. **if** color[v] = WHITE **then**
4. DFS\_Visit(v)
5. time ⟵time-1
6. order[v] ⟵ time

## Поиск компонент сильной связности.

Сначала делается топологическая сортировка. Потом «транспонируется» граф и проводится серия поисков в глубину в порядке, определяемом топологической сортировкой. Каждое дерево поиска определяет набор вершин относящихся к одной компоненте сильной связности.

**Фактор-граф по компонентам сильной связности** (конденсация графа) – это граф, вершинами которого являются компоненты сильной связности исходного графа, а дугой соединены u и v, если существует u’∈u и v’∈v, такие что имеется дуга (u’, v’) исходного графа.

(а дугой соединены две вершины, если в КСС соответствующей первой вершине имеется вершина u, а в КСС соответствующей второй вершине имеется вершина v, такие что в исходном графе есть дуга (u, v).)

void dfs\_r (int v, int n) {

used[v] = true;

component[v] = n;

for (int to : gr[v]) {

if (!used[to])

dfs\_r (to, n);

}

}

void scan\_reverse\_graph() {

used.assign(gr.size(), false);

int count = 0;

for (int i : order)

if (!used[i])

dfs\_r(i, count++);

}

void search\_for\_components () {

topological\_sort();

scan\_reverse\_graph();

}