Введение в алгоритмы.

Лекция 13
Графы.

Сергей Леонидович Бабичев

Графы. Представление графов.

Графы: применение

- Географические карты. Какой маршрут из Москвы в Лондон требует наименьших расходов? Какой маршрут из Москвы в Лондон требует наименьшего времени? Требуется информация о связях между городами и о стоимости этих связей.
- Микросхемы. Транзисторы, резисторы и конденсаторы связаны между собой проводниками. Есть ли короткие замыкания в системе? Можно ли так переставить компоненты, чтобы не было пересечения проводников?
- Расписания задач. Одна задача не может быть начата без решения других, следовательно имеются связи между задачами. Как составить график решения задач так, чтобы весь процесс завершился за наименьшее время?

Графы: применение

- Компьютерные сети. Узлы — конечные устройства, компьютеры, планшеты, телефоны, коммутаторы, маршрутизаторы... Каждая связь обладает свойствами латентности и пропускной способности. По какому маршруту послать сообщение, чтобы она было доставлено до адресата за наименьшее время? Есть ли в сети «критические узлы», отказ которых приведёт к разделению сети на несвязные компоненты?
- Структура программы. Узлы — функции в программе. Связи — может ли одна функция вызвать другую (статический анализ) или что она вызовет в процессе исполнения программы (динамический анализ). Чтобы узнать, какие ресурсы потребуется выделять системе, требуется граф,

Графы: основные термины

- **Ориентированный граф**: $G = (V, E)$ есть пара из V — конечного множества и E — подмножества множества $V \times V$.
- **Вершины графа**: элементы множества V (*vertex, vertices*).
- **Рёбра графа**: элементы множества E (*edges*, *связи*).
- **Неориентированный граф**: рёбра есть неупорядоченные пары.
- **Петля:** ребро из вершины v_1 в вершину v_2, где $v_1 = v_2$.

Графы: основные термины

- **Смежные вершины**: v_i и v_j смежны, если имеется связь (v_i, v_j), *соседние*
- **Множество смежных вершин**: обозначаем $Adj[v]$, *соседи*.
- **Степень вершины**: величина $|Adj[v]|$
- **Путь из** v_0 **в** v_n: последовательность рёбер, таких, что $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2) \dots e_n = (v_{n-1}, v_n)$.
- **Простой путь**: путь, в котором все вершины попарно различны.
- **Длина пути**: количество n рёбер в пути.
- **Цикл**: путь, в котором $v_0 = v_n$.

Графы: основные термины

- **Неориентированный связный граф**: для любой пары вершин существует путь.
- **Связная компонента вершины** v: множество вершин неориентированного графа, до которых существует путь из v.
- **Расстояние между** v_i **и** v_j $\delta(v_i, v_j)$: длина кратчайшего пути из v_i в v_j.

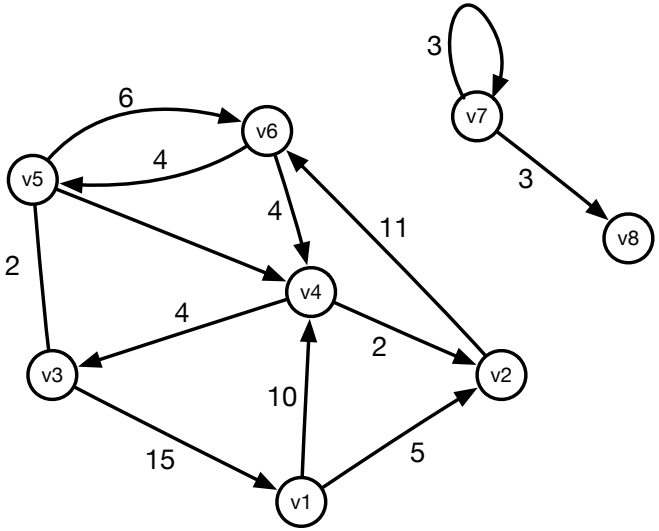
$$\delta(u, v) = 0 \Leftrightarrow u = v$$

$$\delta(u, v) \leqslant \delta(u, v') + \delta(v', v)$$

- **Дерево**: связный граф без циклов.
- **Граф со взвешенными рёбрами**: каждому ребру приписан вес $c(u, v)$.

Графы: основные термины

Ориентированный граф.



Типичные: задачи на графах

- Проверка графа на связность.
- Является ли граф деревом.
- Найти кратчайший путь из u в v.
- Найти цикл, проходящий по всем рёбрам ровно один раз (цикл Эйлера).
- Найти цикл, проходящий по всем вершинам ровно один раз (цикл Гамильтона).
- Проверка на планарность — определить, можно ли нарисовать граф на плоскости без самопересечений.

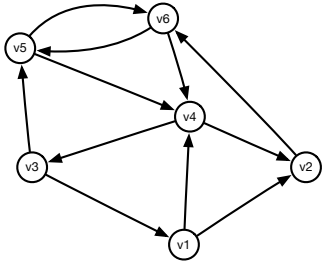
Представление графа в памяти

- Общий принцип: для каждого узла нужно как-нибудь определить, с какими узлами он имеет связь (ребро) и для каждого ребра определить, какие узлы она связывает.
- Рёбра могут иметь вес. Часто это просто число, иногда — функция.
- Идеального представления нет!.
- Иногда нас интересуют в первую очередь вершины и нам важно как-то добираться от одной к другой.
- Иногда нас интересуют в первую очередь рёбра и вершины нужны только чтобы переходить от одного ребра к другому.
- Иногда нам важно абсолютно всё. Именно по этой причине и существуют различные представления.

Представление графа в памяти

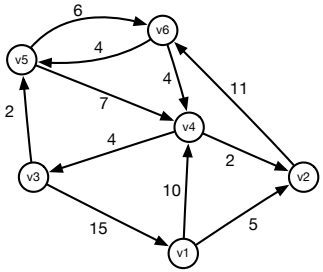
- Всё представляется в виде матрицы смежности.
- Каждой вершине сопоставляется множество смежных в каком-то виде.
- Всё представляется в виде списка рёбер.

Представление графов в памяти: матрица смежности



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	0	1
3	1	0	0	0	1	0
4	0	1	1	0	0	0
5	0	0	1	1	0	1
6	0	0	0	1	1	0

Представление графов в памяти: взвешенный граф

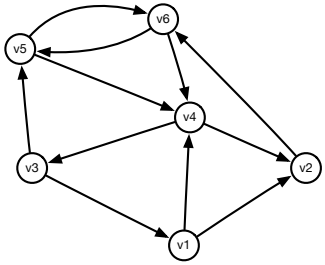


	1	2	3	4	5	6
1	0	5	0	10	0	0
2	0	0	0	0	0	11
3	15	0	0	0	2	0
4	0	2	4	0	0	0
5	0	0	0	7	0	6
6	0	0	0	4	4	0

Матрица смежности

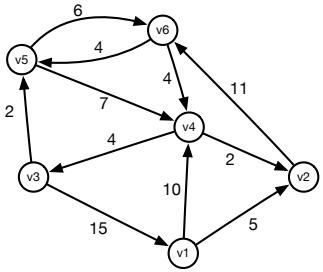
- Недостатки:
  - Во многих реальных графов количество рёбер относительно невелико: типичный граф — карта дорог — степени каждой вершины редко превосходят 5. В матрице хранятся не соседи, а все рёбра. Это требует много памяти — $(O(|V|^2)$.
  - При обходе графа для вершины часто требуется определить список всех соседних — для каждой вершины нужно проверить все элементы строки. Это медленно — $O(|V|)$ для определения всех соседних вершин данной вершины.
- Достоинства:
  - Для конкретного соседа i определение веса ребра за $O(1)$.
  - Для невзвешенного графа можно использовать побитовое представление и параллельную обработку бит.

Представление графов в памяти: множества смежности



$$
\begin{aligned}
v_1 &: \quad \{2, 4\} \\
v_2 &: \quad \{6\} \\
v_3 &: \quad \{1, 5\} \\
v_4 &: \quad \{2, 3\} \\
v_5 &: \quad \{3, 4, 6\} \\
v_6 &: \quad \{4, 5\}
\end{aligned}
$$

Представление графов в памяти: взвешенный граф

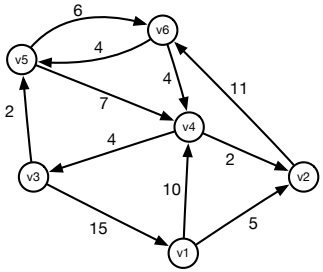


$$
\begin{aligned}
v_1 &: \{(2,5),(4,10)\} \\
v_2 &: \{(6,11)\} \\
v_3 &: \{(1,15),(5,2)\} \\
v_4 &: \{(2,2),(3,4)\} \\
v_5 &: \{(4,7),(6,6)\} \\
v_6 &: \{(4,4),(5,4)\}
\end{aligned}
$$

Множества смежности: варианты представления

- Для каждого узла должны храниться все его действительные соседи. Пусть для конкретного узла имеется K соседей.
- Их можно хранить:
  - В массиве пар: (номер, вес). Недостаток: медленный поиск по номеру — $O(K)$ или $O(\log K)$, в зависимости от упорядоченности. Достоинство: перечисление всех соседей занимает $O(K)$.
  - В дереве поиска. Недостатки: большой расход памяти, перечисление всех соседей занимает $O(K \log K)$. Достоинство: поиск по номеру $O(\log K)$.
  - В хеш-таблице. Недостатки: большой расход памяти, соседи перечисляются не по порядку. Достоинство: поиск по номеру $O(1)$.

Представление графов в памяти: список рёбер



$$\{\{1,2,5\},\{1,4,10\},\{2,6,11\},\{3,1,15\},\{3,5,2\},$$
$$\{4,2,2\},\{5,4,7\},\{5,6,6\},\{6,4,4\},\{6,5,4\}\}$$

Список рёбер: варианты представления

Пусть имеется L рёбер. Общий недостаток: сложный поиск соседей конкретного ребра. Общее достоинство: удобно представляются мультирёбра.

- Массив троек: (откуда, куда, вес). Недостатки: поиск конкретного ребра за $O(L)$ в лексикографически неупорядоченном массиве, $O(\log L)$ в лексикографически упорядоченном. Достоинство: для некоторых алгоритмов удобно именно такое представление.
- Поисковая структура, где ключ — пара (откуда, куда), значение — вес. Сложность поиска конкретного ребра: $O(\log L)$.

Представление графов в памяти

Преимущества и недостатки методов представления:

Представление	Матрица смежности	Множества смежности	Список рёбер
Занимаемая память	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$	$O(\|E\|)$
Особенности	Простой доступ	Требует мало памяти для ряда графов	Можно иметь мультирёбра

Обход графа. Поиск путей в графах. BFS. DFS.

Обход графов

- Иногда единственное, что нам нужно от графов — их обойти.

  Обход графа: исполнение каких-либо действий при последовательном посещении узлов, перемещаясь при этом строго по рёбрам.

  Корень обхода: узел s, в котором начинается исполнение алгоритма обхода.

  Предшественник $\pi(u)$ на пути от s: предпоследний узел в кратчайшем пути из s в u.
- Обходить графы можно для многих целей. Вот немногие:
  - для определения кратчайших маршрутов от корня обхода до остальных узлов;
  - для определения числа компонент связности графа;
  - для разделения графа на компоненты связности;
  - для определения особых узлов (точек сочленения) и особых вершин (мостов);
  - для выстраивания узлов порядке, подходящем для целей планирования решений подзадач.

Поиск в ширину: алгоритм BFS

- Этот алгоритм сначала пытается обработать всех соседей текущей вершины.
- Используется абстракция *очередь* с методами **enqueue** и **dequeue**.

- Термин: **предшественник** $\pi(u)$ **на пути от** s: предпоследняя вершина в кратчайшем пути из s в u.

- Используются цвета:
  - Белый для непросмотренных вершин.
  - Серый для обрабатываемых вершин.
  - Чёрный для обработанных вершин.

Структура данных *очередь*

- Графы для хорошей реализации требуют хороших структур данных — массива с возможностью быстрого добавления элементов, очереди, деревьев поиска.
- Массив с добавлением легко реализовать на Си.
- Деревья поиска мы изучили.
- Давайте дадим пример структуры данных *очередь*.
- В ней, наряду с телом, имеются два указателя — на начало и на конец очереди, которые поочерёдно передвигаются.
- Наша простейшая очередь имеет ограничение на количество операций enqueue.

```
typedef int T;
typedef struct queue_s {
    T *body, *begin, *end, *body_end;
} queue;
```

Структура данных *очередь*

- Методы создания и удаления просты:

```
queue* queue_new(size_t size) {
    queue *t = (queue *)malloc(sizeof(queue));
    assert(t != NULL);
    t->body = (int *)malloc(sizeof(T)*size);
    t->begin = t->end = t->body;
    t->body_end = t->body + size;
    return t;
}

void queue_delete(queue *t) {
    free(t->body);
    free(t);
}
```

Структура данных *очередь*

- Остальные методы — тоже:

```
void queue_enqueue(queue *t, T el) {
    assert(t->end < t->body_end);
    *t->end++ = el;
}

int queue_empty(queue const *t) {
    return t->begin == t->end;
}

T queue_dequeue(queue *t) {
    return *t->begin++;
}
```

Представление графа

- Для простоты будем представлять граф в виде линеаризированной матрицы.

```
typedef enum color_t {
    WHITE, GREY, BLACK
} color;

// unweighted graph
typedef struct graph_t {
    char *neibs; // [N*N]
    size_t N; // ^^d0^^a0^^d0^^b0^^d0^^b7^^d0^^bc^^d0^^b5^^d1^^80
    int unordered; // ^^d0^^9d^^d0^^b0^^d0^^bf^^d1^^80^^d0^^b0^^d0^^b2^^d0^^bb^^d0^^b
} graph;

graph *graph_new(size_t nodes, int is_ordered) {
    graph *t = (graph *) malloc(sizeof(graph));
    t->N = nodes;
    t->unordered = !is_ordered;
    t->neibs = (char *) calloc(sizeof(char), nodes * nodes);
    return t;
}
```

Представление графа

- Операции удаления и добавления ребра.

```
void graph_delete(graph *g) {
    free(g->neibs);
    free(g);
}

void graph_add_edge(graph *t, int from, int to) {
    t->neibs[from * t->N + to] = 1;
    if (t->unordered) {
        t->neibs[to * t->N + from] = 1;
    }
}
```

Алгоритм BFS

- Одна из целей алгоритма BFS — найти кратчайшие расстояния от заданной вершины (корня) до остальных.
- Идея: если расстояние от рассматриваемой вершины до корня равно d, то расстояние от этой вершины до всех ещё не просмотренных будет d+1.
- Для этого вершины выкрашиваются в разные цвета.
- Алгоритм BFS ведёт очередь тех элементов, которые уже начали обрабатываться, но для них не просмотрены их соседи.
- Эти элементы извлекаются из очереди и обрабатываются.
- При обработке одного элемента в очередь попадают все необработанные соседи.

Обход графа: поиск в ширину от s, BFS

Просмотр вершин графа в порядке возрастания расстояния от s.

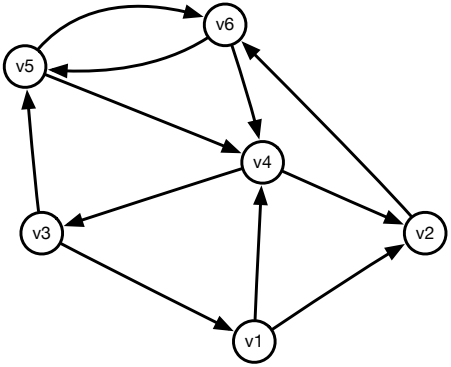
```
 1: procedure BFS(G : Graph, s : Vertex)
 2:     for all u ∈ V[G] \ {s} do
 3:         d[u] ← ∞;      c[u] ← white;      π[u] ← nil
 4:     end for
 5:     d[s] ← 0;    c[s] ← grey
 6:     Q.enqueue(s)
 7:     while Q ≠ ∅ do
 8:         u ← Q.dequeue()
 9:         for all v ∈ Adj[u] do
10:             if c[v] = white then
11:                 Q.enqueue(v)
12:                 d[v] ← d[u] + 1
13:                 π[v] = u;    c[v] = grey
14:             end if
15:         end for
16:         c[u] ← black
17:     end while
18: end procedure
```

Код алгоритма BFS

```
void graph_bfs(graph* t, int s, int *d)  {
    color *c = (color *)calloc(sizeof(color),t->N);
    for (int i = 0; i < t->N; i++) d[i] = -1;
    queue *q = queue_new(t->N*t->N);
    queue_enqueue(q, s);
    d[s] = 0; c[s] = GREY;
    while (!queue_empty(q)) {
        int u = queue_dequeue(q);
        for (int z = 0; z < t->N; z++) {
            if (t->neibs[u*t->N+z] == 0 || c[z] != WHITE) continue;
            d[z] = d[u] + 1;
            c[z] = GREY;
            queue_enqueue(q, z);
        }
        c[u] = BLACK;
    }
    queue_delete(q);
    free(c);
}
```

Прогон алгоритма BFS

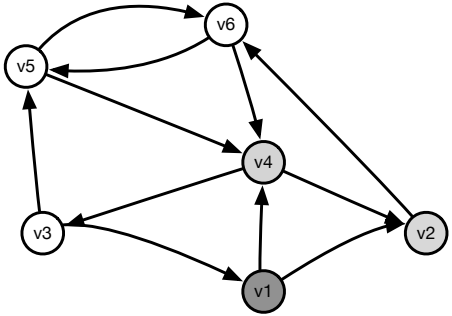
Начало алгоритма.



$d = \{0, \infty, \infty, \infty, \infty, \infty\}$
$\pi = \{nil, nil, nil, nil, nil, nil\}$
$Q = \{v_1\}$

Прогон алгоритма BFS

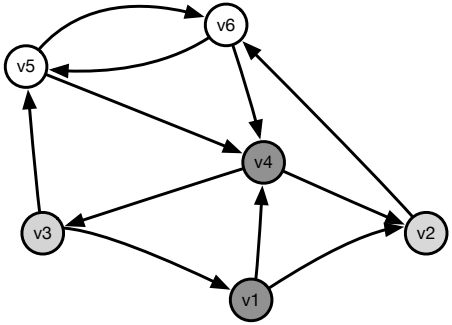
После первого прохождения цикла `While`



$d = \{0, 1, \infty, 1, \infty, \infty\}$
$\pi = \{nil, v_1, nil, v_1, nil, nil\}$
$Q = \{v_4, v_2\}$

Прогон алгоритма BFS

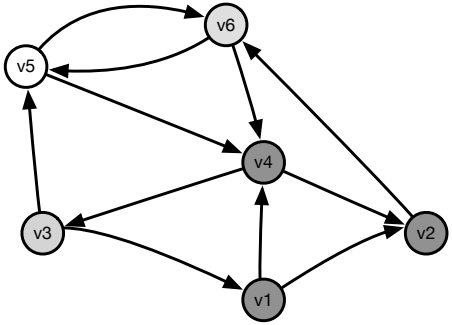
После второго прохождения цикла `While`



$d = \{0, 1, 2, 1, \infty, \infty\}$
$\pi = \{nil, v_1, v_4, v_1, nil, nil\}$
$Q = \{v_2, v_3\}$

Прогон алгоритма BFS

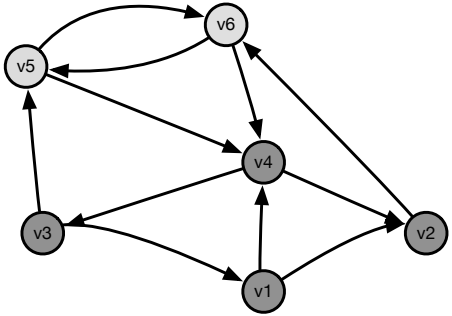
После третьего прохождения цикла `While`



$d = \{0, 1, 2, 1, \infty, 2\}$
$\pi = \{nil, v_1, v_4, v_1, nil, v_2\}$
$Q = \{v_3, v_6\}$

Прогон алгоритма BFS

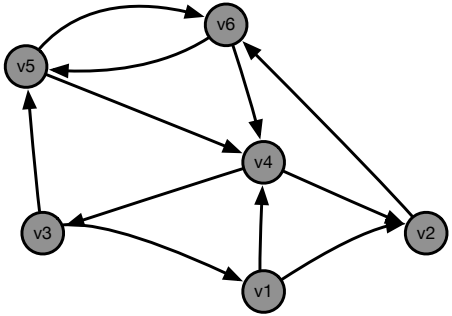
После четвёртого прохождения цикла `While`



$d = \{0, 1, 2, 1, 3, 2\}$
$\pi = \{nil, v_1, v_4, v_1, v_3, v_2\}$
$Q = \{v_6, v_5\}$

Прогон алгоритма BFS

Завершение алгоритма



$d = \{0, 1, 2, 1, 3, 2\}$
$\pi = \{nil, v_1, v_4, v_1, v_3, v_2\}$
$Q = \{\}$

Алгоритм BFS: свойства

Сложность алгоритма:

- представление в виде множества смежности:
  - Инициализация: $O(|V|)$
  - Каждая вершина обрабатывается не более одного раза. Проверяются все смежные вершины.

$$\sum_{v \in V} |Adj(v)| = O(|E|)$$

  - $T = O(|V| + |E|)$

Поиск в глубину: алгоритм DFS

- Этот алгоритм пытается идти вглубь, пока это возможно.
- Обнаружив вершину, алгоритм не возвращается, пока не обработает её полностью.
- Используются переменные
  - $time$ — глобальные часы.
  - $d[u]$ — время начала обработки вершины. u
  - $f[u]$ — время окончания обработки вершины. u
  - $\pi[u]$ — предшественник вершины u.

Алгоритм DFS

```
1:  procedure DFS(G : Graph)
2:      for all u ∈ V[G] do
3:          c[u] ← white;        π[u] ← nil
4:      end for
5:      time ← 0
6:      for all u ∈ V[G] do
7:          if c[u] = white then
8:              DFS-vizit(u)
9:          end if
10:     end for
11: end procedure
```

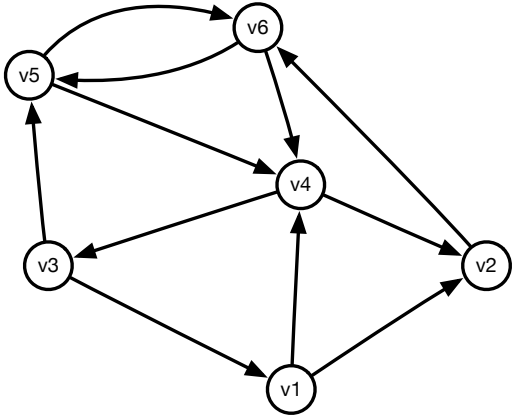
Алгоритм DFS

```
1: procedure DFS-VIZIT(u : Vertex)
2:     c[u] ← grey
3:     time ← time + 1
4:     d[u] ← time
5:     for all v ∈ Adj[u] do
6:         if c[v] = white then
7:             π[v] ← u
8:             DFS-vizit(v)
9:         end if
10:    end for
11:    c[u] ← black
12:    time ← time + 1
13:    f[u] ← time
14: end procedure
```

Прогон алгоритма DFS

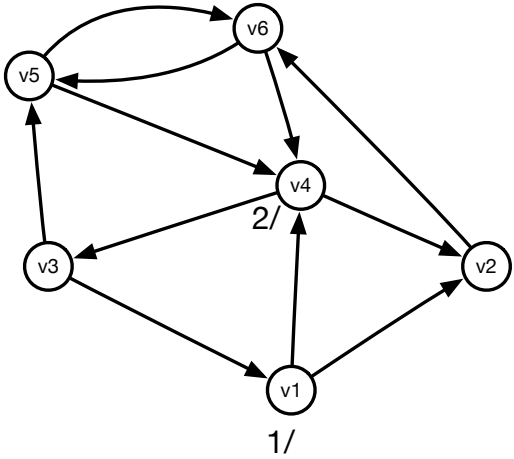
Начинается обход с вершины v_1

Около каждой вершины пишем два числа: время входа в вершину и через знак / — время выхода из вершины.



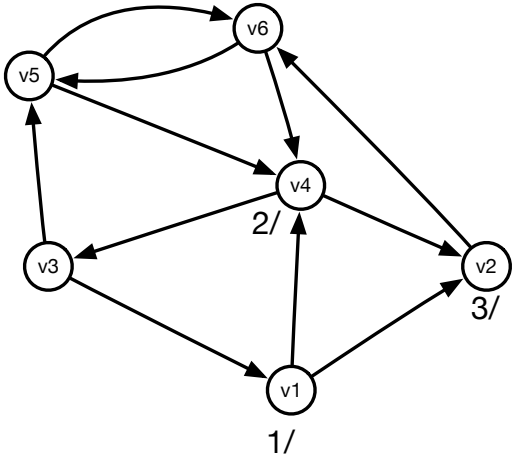
Прогон алгоритма DFS

Первый рекурсивный вызов DFS-vizit(v_4)



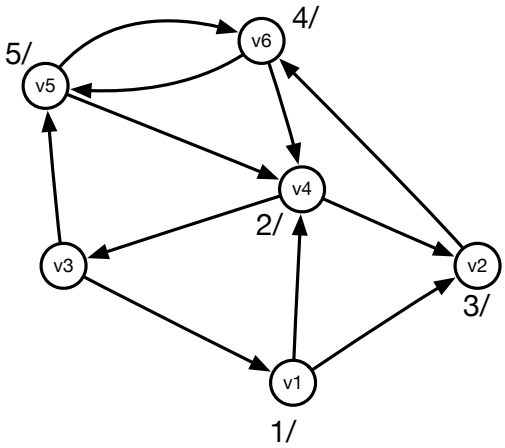
Прогон алгоритма DFS

Второй рекурсивный вызов DFS-vizit(v_2)



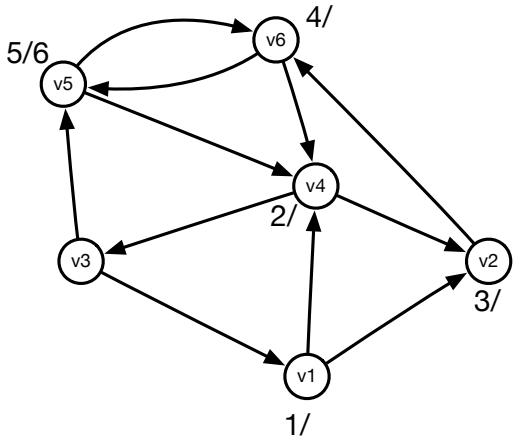
Прогон алгоритма DFS

Пятый рекурсивный вызов DFS-vizit(v_5)



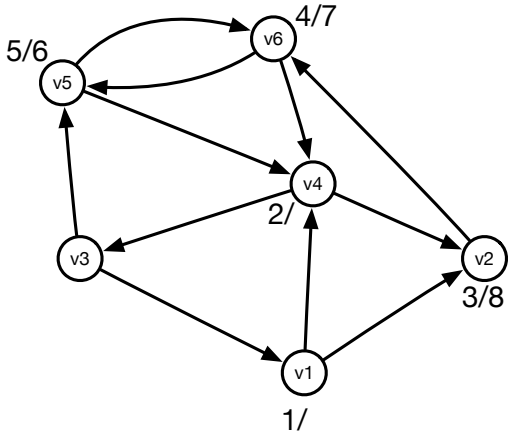
Прогон алгоритма DFS

Выход из пятой рекурсии вызова DFS-vizit(v_5)



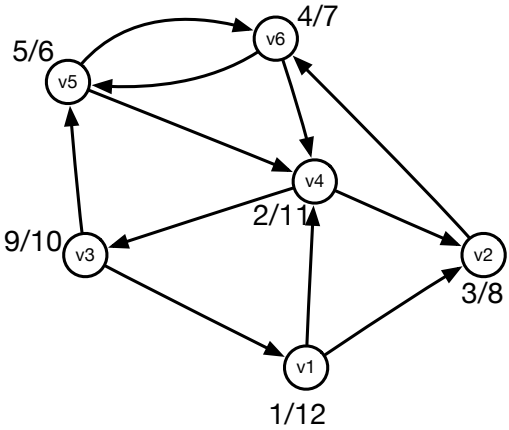
Прогон алгоритма DFS

Выход из рекурсии вызов DFS-vizit(v_2)



Прогон алгоритма DFS

Завершение алгоритма



Алгоритм DFS

- Сложность алгоритма для представления в виде множества смежности равна $O(|V| + |E|)$
- Этот алгоритм не находит кратчайшие маршруты!

Топологическая сортировка

Небольшая задача

Удалось получить несколько зашифрованных слов, которые в незашифрованном виде были лексикографически отсортированы. Эти слова состоят из прописных букв латинского алфавита.
Ваша задача состоит в том, чтобы вы восстановили шифрующую строку.
Известно, что входной словарь состоит из подряд идущих букв латинского алфавита и известна его мощность.
Например, если мощность алфавита равна 5 и шифрующая строка `BAEDC`, то все символы `A` преобразуются в `B`, `B` — в `A`, `C` — в `E`, `D` останется неизменным, `C` — в `E`.

Небольшая задача: вход

```
5
20
DDCDA
DDADE
DDECE
DCEAC
DCEAA
DBADA
DBACB
ADCCE
ACCDE
ACCDB
ABCDA
ABEDA
EDCDA
ECCDB
ECADE
ECAAE
ECEDE
EBCDE
```

Небольшая задача

- А разве это задача на графы? Да.
- Первые две шифрованные строки — `DDCDA` и `DDADE`.
- Так образы этих строк, были упорядочены, можно сделать вывод, что то, что было зашифровано буквой `C` меньше, чем то, что было зашифровано буквой `A`.
- Так это же ребро графа!
- Наш граф состоит из вершин, букв, и направленных рёбер — порядка на этих буквах.
- Этот граф ациклический и направленный (ориентированный).
- Наша задача — упорядочить вершины таким образом, чтобы на временно́й линии все вершины, значение которых меньше, располагались левее.
  Такая задача называется *топологической сортировкой*.

Топологическая сортировка

Задача: имеется ориентированный граф $G = (V, E)$ без циклов.

Требуется указать такой порядок вершин на множестве V, что любое ребро ведёт из меньшей вершины к большей.
Требуемая структура данных: L — очередь с операцией `enqueue`.

Топологическая сортировка

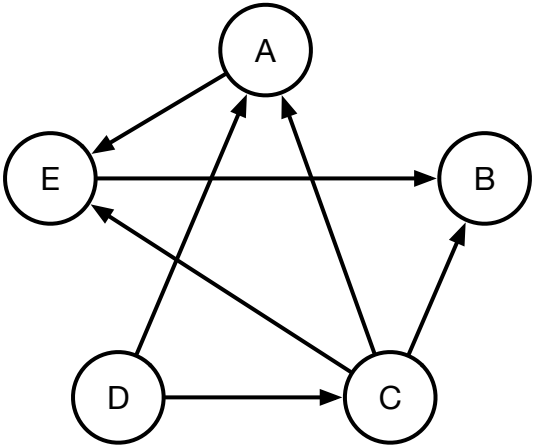
```
1: procedure TopoSort(G : Graph)
2:     L ← 0
3:     for all u ∈ V[G] do
4:         c[u] ← white;
5:     end for
6:     time ← 0
7:     for all u ∈ V[G] do
8:         if c[v] = white then
9:             DFS-vizit(u)
10:        end if
11:    end for
12: end procedure
```

Топологическая сортировка

```
1:  procedure DFS-VIZIT(u : Vertex)
2:      c[u] ← grey
3:      for all v ∈ Adj[u] do
4:          if c[v] = white then
5:              π[v] ← u
6:              DFS-vizit(u)
7:          end if
8:      end for
9:      c[u] ← black
10:     L.enqueue(u)
11: end procedure
```

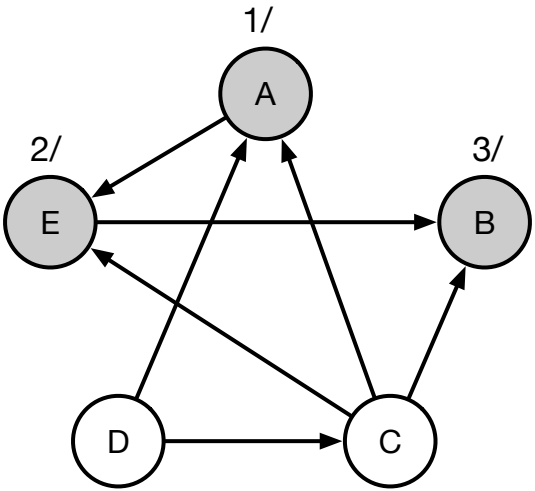
Прогон алгоритма топологической сортировки

Пусть обход начнётся с вершины A



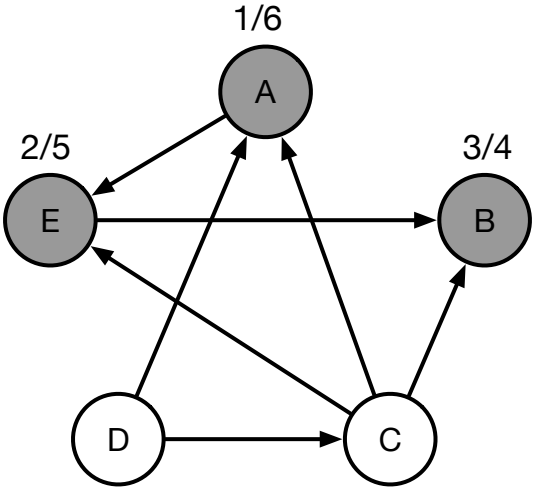
Прогон алгоритма топологической сортировки

Пусть обход начнётся с вершины A



Прогон алгоритма топологической сортировки

Пусть обход начнётся с вершины A



Прогон алгоритма топологической сортировки

Результат обхода.