

Алгоритмы и структуры данных

Лекция 13

Графы. Представление. Обход.

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 оно 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 .

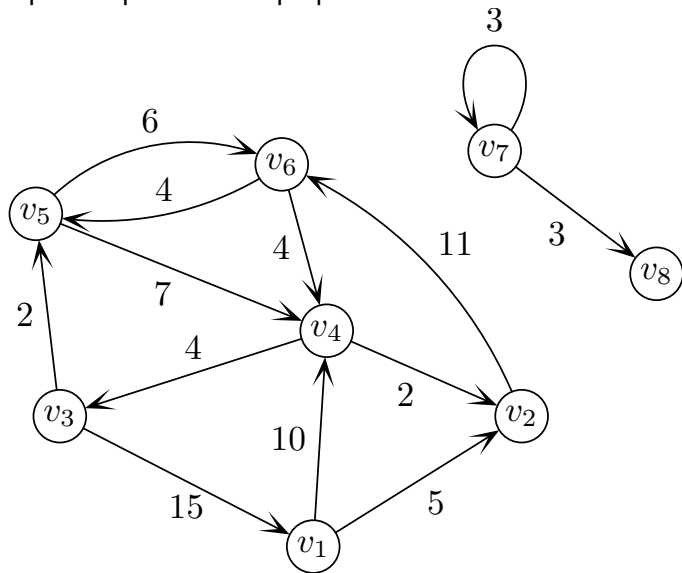
$$\delta(u, v) = 0 \Leftrightarrow u = v$$

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

- **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.

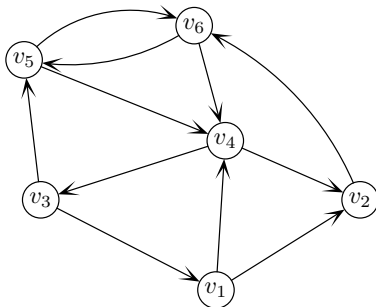
Представление графа в памяти

- 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

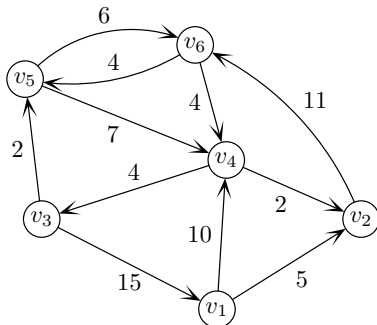
- Всё представляется в виде матрицы смежности.
- Каждой вершине сопоставляется множество смежных в каком-то виде.
- Всё представляется в виде списка рёбер.

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 (MATRIX)

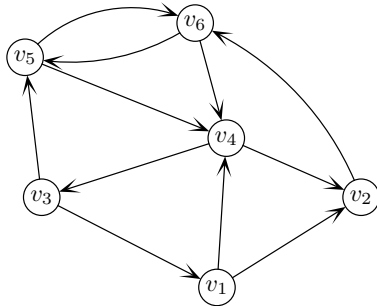
- Недостатки:

- ▶ 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



$v_1 : \{2, 4\}$

$v_2 : \{6\}$

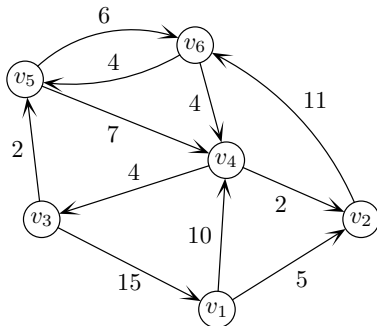
$v_3 : \{1, 5\}$

$v_4 : \{2, 3\}$

$v_5 : \{3, 4, 6\}$

$v_6 : \{4, 5\}$

Представление графов в памяти: взвешенный граф



$v_1 : \{(2, 5), (4, 10)\}$

$v_2 : \{(6, 11)\}$

$v_3 : \{(1, 15), (5, 2)\}$

$v_4 : \{(2, 2), (3, 4)\}$

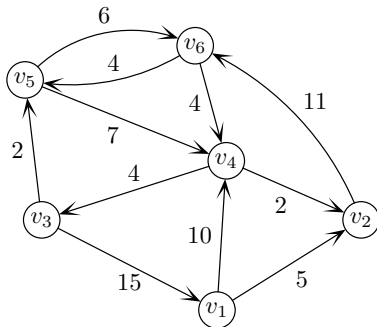
$v_5 : \{(4, 7), (6, 6)\}$

$v_6 : \{(4, 4), (5, 4)\}$

Множества смежности: варианты представления

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

Представление графов в памяти: список рёбер (LIST)



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

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: K — количество соседей у вершины.

Представление	Матрица смежности MATRIX	Множества смежности ARRAY	Множества смежности OMAP	Множества смежности HMAP	Список рёбер LIST
Занимаемая память	$O(V ^2)$	$O(V + E)$	$O(V + E)$	$O(V + E)$	$O(E)$
Особенности	$O(V)$ обход всех соседей $O(1)$ поиск соседа	$O(K)$ обход всех соседей $O(K)$ поиск соседа	$O(K \log K)$ обход всех соседей $O(\log K)$ поиск соседа	$O(K)$ обхода всех соседей $O(1)$ поиск соседа	обхода нет поиска нет 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.

Одно из представлений графа (ARRAY)

```
// unweighted graph
class graph {
private:
    using color = enum color_e { WHITE, GREY, BLACK };
    std::vector<std::vector<int>> neibs;
    ...
};
```

BFS

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 *быстрая очередь*

- Графы для хорошей реализации требуют хороших структур данных — массива с возможностью быстрого добавления элементов, очереди, поисковых структур.
- Иногда удобно применять абстракцию *быстрая очередь*
- В ней, наряду с телом, имеются два указателя — на начало и на конец очереди, которые поочерёдно передвигаются.
- Такая очередь имеет ограничение на количество операций `enqueue`.

Быстрая очередь

```
template<typename T>
class fast_queue {
private:
    T *body, *begin, *end, *body_end;
public:
    explicit fast_queue(size_t size) {
        body = new T[size];
        begin = end = body;
        body_end = body + size;
    }
    ~fast_queue() { delete[] body; }
    void enqueue(T const &el) { *end++ = el; }
    bool empty() const { return begin == end; }
    T dequeue() { return *begin++; }
};
```

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 соседей 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 .

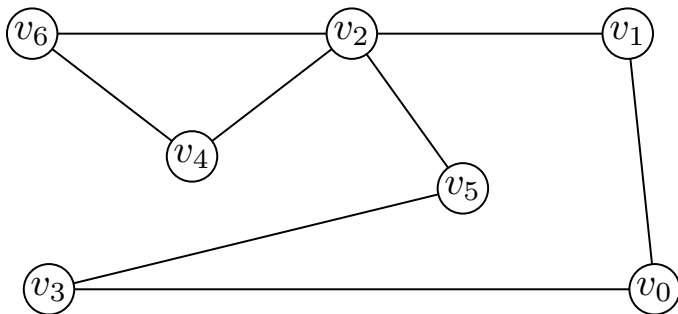
```
1: procedure BFS( $G : Graph, s : Vertex$ )
2:   for all  $u \in V[G] \setminus \{s\}$  do
3:      $d[u] \leftarrow \infty; \quad c[u] \leftarrow \text{white}; \quad \pi[u] \leftarrow \text{nil}$ 
4:   end for
5:    $d[s] \leftarrow 0; \quad c[s] \leftarrow \text{grey}$ 
6:    $Q.enqueue(s)$ 
7:   while  $Q \neq \emptyset$  do
8:      $u \leftarrow Q.dequeue()$ 
9:     for all  $v \in Adj[u]$  do
10:      if  $c[v] = \text{white}$  then
11:         $Q.enqueue(v)$ 
12:         $d[v] \leftarrow d[u] + 1$ 
13:         $\pi[v] = u; \quad c[v] = \text{grey}$ 
14:      end if
15:    end for
16:     $c[u] \leftarrow \text{black}$ 
17:  end while
18: end procedure
```

Код алгоритма BFS

```
auto graph::bfs(int s) {
    vector<int> d(N, 100000);
    vector<color> c(N, WHITE);
    fast_queue<int> q(N*N);
    q.enqueue(s);
    d[s] = 0; c[s] = GREY;
    while (!q.empty()) {
        int u = q.dequeue();
        for (auto v: neibs[u]) {
            if (c[v] != WHITE) continue;
            d[v] = d[u] + 1;
            c[v] = GREY;
            q.enqueue(v);
        }
        c[u] = BLACK;
    }
    return d;
}
```

Прогон алгоритма BFS

Начало алгоритма.

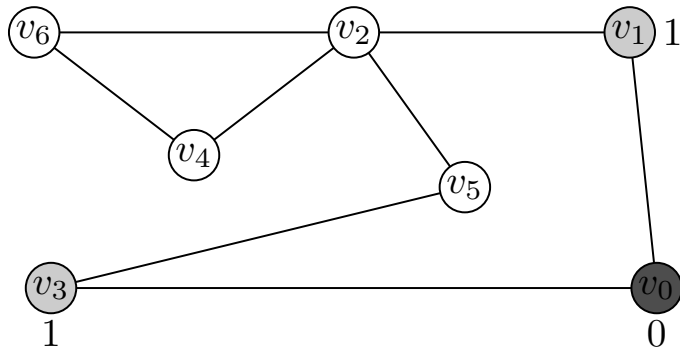


$$d = \{0, \infty, \infty, \infty, \infty, \infty, \infty\}$$

$$q = \{v_0\}.$$

Прогон алгоритма BFS

После первой итерации цикла `while` в очередь попали все смежные с v_0 белые вершины.

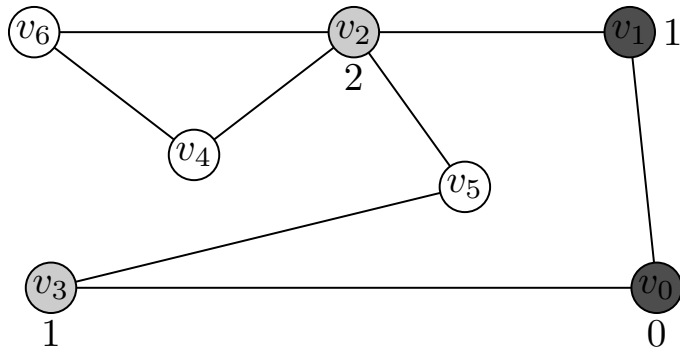


$$d = \{0, 1, \infty, 1, \infty, \infty, \infty\}$$

$$q = \{v_1, v_3\}.$$

Прогон алгоритма BFS

При втором проходе цикла `while` из очереди извлечена вершина v_1 — и туда добавлена вершина v_2 .

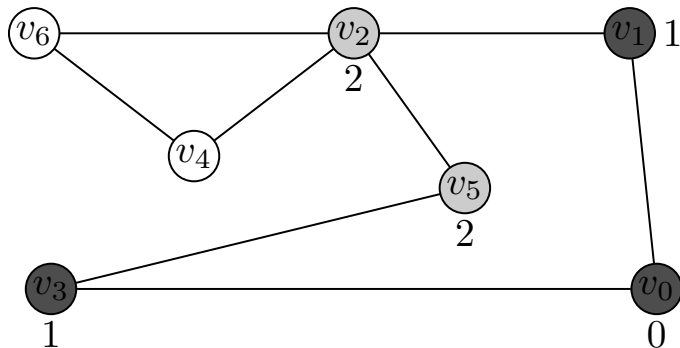


$$d = \{0, 1, 2, 1, \infty, \infty, \infty\}$$

$$q = \{v_3, v_2\}.$$

Прогон алгоритма BFS

Третье прохождение цикла `while` уберёт из очереди v_3 и добавит в конец очереди v_5 .



$$d = \{0, 1, 2, 1, \infty, 2, \infty\}$$

$$q = \{v_2, v_5\}.$$

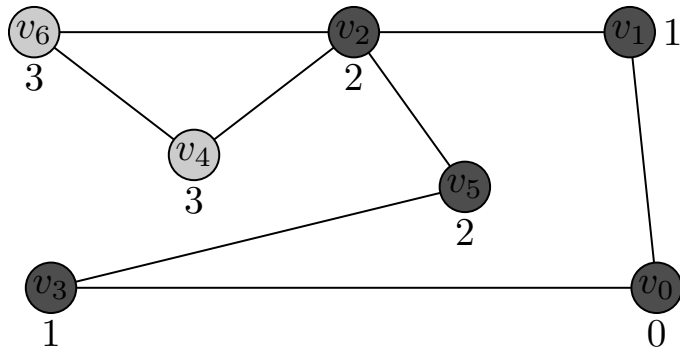
Прогон алгоритма BFS

Четвёртая итерация попытается найти белых соседей у вершины v_5 , ей это не удастся и вершина будет выкрашена в чёрный цвет.

После пятого прохождения цикла `while` очередь покинет вершина v_2 и в неё добавятся вершины v_4 и v_6 . После этого все вершины выкрашены в цвета, отличные от белого, и в очередь больше ничего добавляться не будет.

Прогон алгоритма BFS

Завершение алгоритма: заполнен массив расстояний от начальной вершины. Если требуется, массив предшественников.



$$d = \{0, 1, 2, 1, 3, 2, 3\}$$

$$q = \{v_5, v_5\}.$$

Алгоритм BFS: сложность

- Фаза инициализации имеет сложность $O(|V|)$. Каждая вершина попадает в очередь не более одного раза.
- Извлечение смежных вершин имеет различную сложность для различных представлений. $K = |Adj(v)|$
 - ▶ *MATRIX*: получение всех смежных вершин для данной $O(|V|)$;
 - ▶ *ARRAY*: $O(K)$;
 - ▶ *OMAP*: $O(K \log K)$;
 - ▶ *HMAP*: $O(K)$.
- Лучший результат у *ARRAY*.

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

- Итоговая сложность алгоритма: $T = O(|V| + |E|)$.

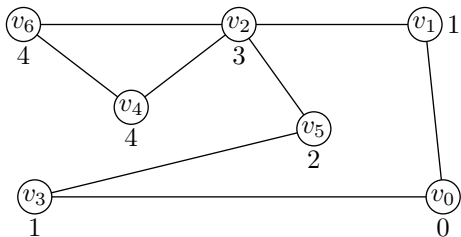
DFS

- В алгоритме BFS мы пользовались структурой данных queue или FIFO, сокращение от First In -- First Out.
- Поместив в FIFO все найденные необработанные вершины, мы автоматически обрабатывали их в порядке поступления: как только мы заходим в обработчик очередной вершины, первым делом он кладёт в очередь всех своих белых соседей и только после этого обработка вершины считается завершённой.
- Если мы поменяем порядок обработки на противоположный, то есть если первым будет обрабатываться только что пришедшая вершина, дисциплина обработки станет LIFO, Last In -- First Out, или stack.

Наивная попытка реализовать DFS

```
auto graph::dfs(int s) {
    vector<int> d(N, -1);
    vector<color> c(N, WHITE);
    std::stack<int> q;
    q.push(s);
    d[s] = 0; c[s] = GREY;
    while (!q.empty()) {
        int u = q.top(); q.pop();
        for (auto v: neibs[u]) {
            if (c[v] != WHITE) continue;
            d[v] = d[u] + 1;
            c[v] = GREY;
            q.push(v);
        }
        c[u] = BLACK;
    }
    return d;
}
```


Наивный прогон алгоритма



- Что значит «расстояние до вершины 6» равно 4?
- А что мы хотим получить от алгоритма для вершины?
- Пару: номер шага алгоритма, на котором эта вершина поступила на рассмотрение (время входа в вершину), и время выхода из вершины.
- Каким образом можно зафиксировать время выхода из вершины?
- В алгоритме BFS всё просто: как только для текущей вершины все смежные белые отправились в очередь, вершину можно считать закрытой.
- В DFS закрыть её можно только тогда, когда будут обработаны все смежные.
- Нужно помнить, кто из соседей обработан, а кто ещё нет. Используем рекурсию.

Алгоритм DFS

- Разобьём алгоритм на две части: нерекурсивную и рекурсивную.
- Нерекурсивная часть инициализирует переменные и запускает рекурсивный обход для каждой из ещё непосещённых вершин.
- В рекурсивной части используются общие переменные, созданные и инициализированные в нерекурсивной части.
 - ▶ $time$ — глобальные часы ал.
 - ▶ $d[u]$ — 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  $u \in V[G]$  do
3:      $c[u] \leftarrow \text{white}; \quad \pi[u] \leftarrow \text{nil}$ 
4:   end for
5:    $time \leftarrow 0$ 
6:   for all  $u \in V[G]$  do
7:     if  $c[u] = \text{white}$  then
8:       DFS-vizit( $u$ )
9:     end if
10:  end for
11: end procedure
```

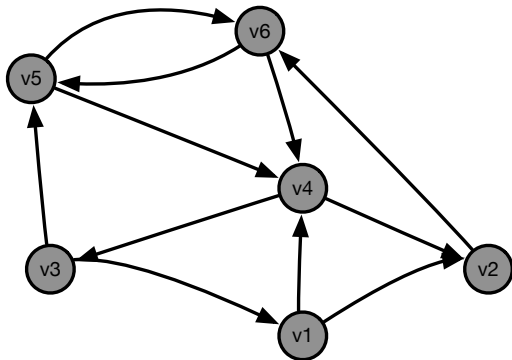
Алгоритм DFS

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

Прогон алгоритма DFS

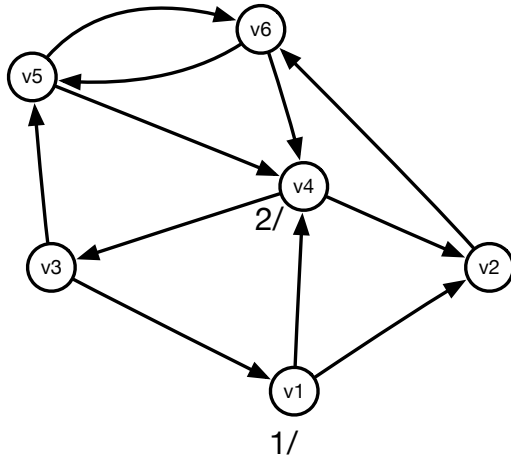
Начинается обход с вершины v_1

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



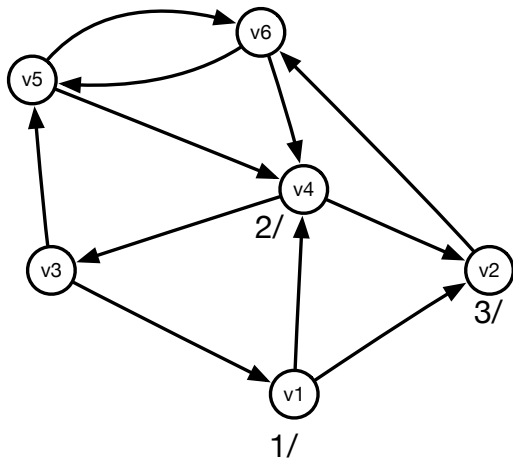
Прогон алгоритма DFS

Первый рекурсивный вызов $\text{DFS-visit}(v_4)$



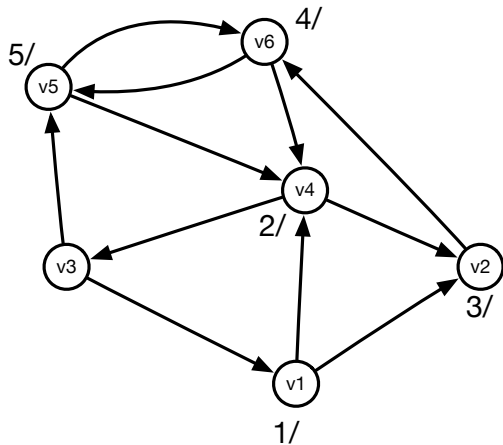
Прогон алгоритма DFS

Второй рекурсивный вызов $\text{DFS-vizit}(v_2)$



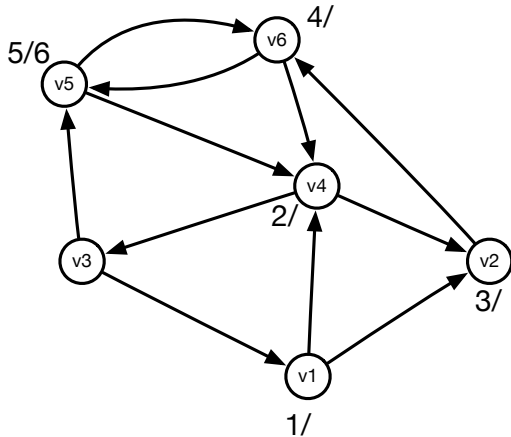
Прогон алгоритма DFS

Пятый рекурсивный вызов $\text{DFS-visit}(v_5)$



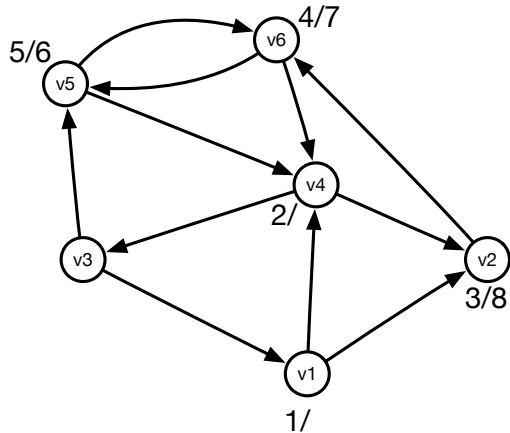
Прогон алгоритма DFS

Выход из пятой рекурсии вызова $\text{DFS-visit}(v_5)$



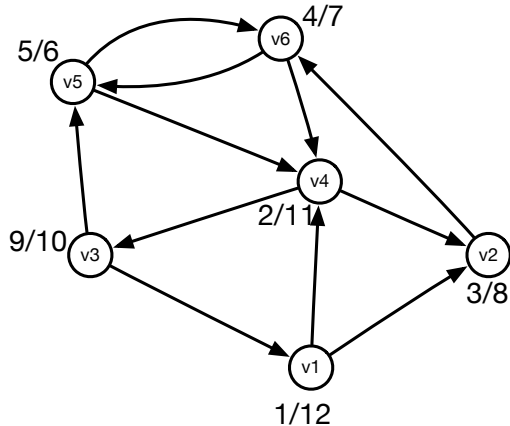
Прогон алгоритма DFS

Выход из рекурсии вызов $\text{DFS-visit}(v_2)$



Прогон алгоритма DFS

Завершение алгоритма



Алгоритм DFS: без рекурсии

- Мы же уже пытались! Не получилось. Почему?
- Если мы кладём в стек сразу все соседние вершины, то каким образом мы обнаружим, что все соседи уже обработаны и пора закрывать текущую вершину?
- Трюк: как только вершина первый раз попадает в стек, мы помечаем её флажком MARK.
- Если при извлечении из стека вершина помечена, то в стек отправляется она без метки и всех её помеченные белые соседи.
- Если она не помечена, то мы вернулись в ту вершину, которую уже обрабатывали и её можно закрыть.

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, E — в C.

Небольшая задача: вход

5

20

DDCDA

DDADE

DDECE

DCEAC

DCEAA

DBADA

DBACB

ADCCE

ACCDE

ACCDB

ABCD A

ABEDA

EDCDA

ECCDB

ECADE

ECAAE

ECEDE

EBCDE

Небольшая задача

- А разве это задача на графы? Да.
- Первые две зашифрованные строки — DDCDA и DDADE.
- Так образы этих строк, были упорядочены, можно сделать вывод, что то, что было зашифровано буквой С меньше, чем то, что было зашифровано буквой А.
- 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 временной 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

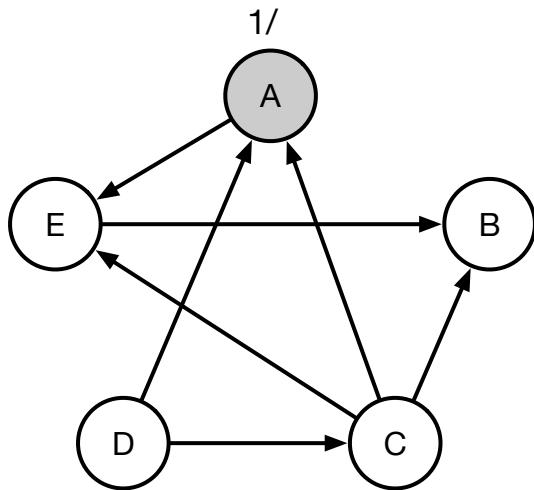
```
1: procedure TOPOSORT( $G : Graph$ )
2:    $L \leftarrow 0$ 
3:   for all  $u \in V[G]$  do
4:      $c[u] \leftarrow \text{white};$ 
5:   end for
6:    $time \leftarrow 0$ 
7:   for all  $u \in V[G]$  do
8:     if  $c[v] = \text{white}$  then
9:       DFS-visit( $u$ )
10:    end if
11:  end for
12: end procedure
```

Топологическая сортировка

```
1: procedure DFS-VIZIT( $u : Vertex$ )
2:    $c[u] \leftarrow \text{grey}$ 
3:   for all  $v \in Adj[u]$  do
4:     if  $c[v] = \text{white}$  then
5:        $\pi[v] \leftarrow u$ 
6:       DFS-vizit( $u$ )
7:     end if
8:   end for
9:    $c[u] \leftarrow \text{black}$ 
10:  L.enqueue( $u$ )
11: end procedure
```

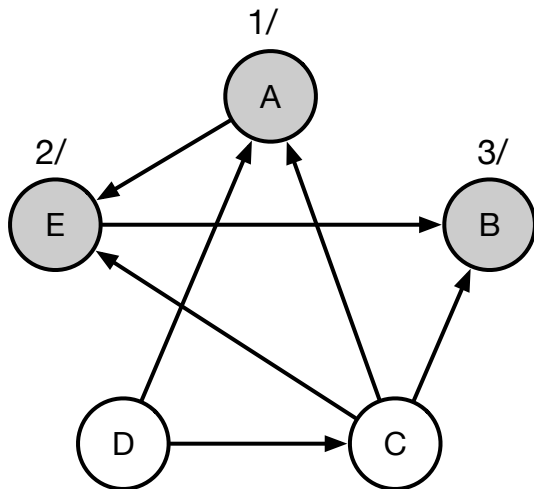
Прогон алгоритма топологической сортировки

Пусть обход начнётся с вершины A



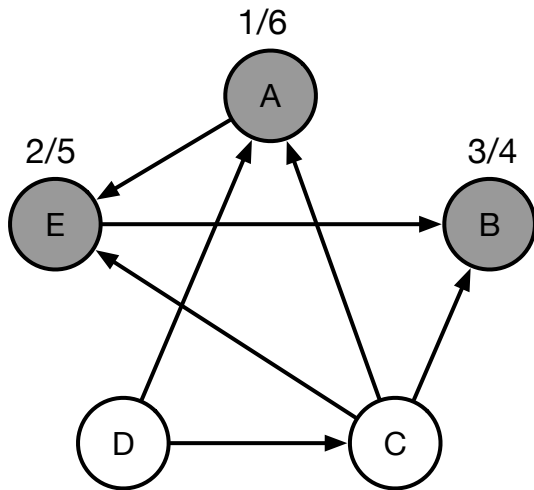
Прогон алгоритма топологической сортировки

Обход продолжился по вершинам E и B . Из B дороги нет — вершина выкрашивается в чёрный цвет и отправляется в очередь.



Прогон алгоритма топологической сортировки

За вершиной B чернеют E и A и они отправляются *перед* B .



Прогон алгоритма топологической сортировки

Результат обхода.

