ТЕХНОСФЕРА

Алгоритмы на графах

Лекция 8

План лекции

- Графы. Представление графов.
- Пути в графе. Понятие релаксации.
- Обход графа. Поиск путей в графах. `BFS`. `DFS`.
- Топологическая сортировка.
- Поиск остовных деревьев в графе. Алгоритмы Прима и Краскала.
- Алгоритм Дейкстры и его связь с жадными алгоритмами.
- Алгоритм Флойда-Уоршалла и его связь с динамическим программированием.
- Поиск максимального потока и алгоритм Форда-Фалкерсона.

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*).
- **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).
- **Множество смежных вершин**: обозначаем $Adj[v]$
- **Степень вершины**: величина $|Adj[v]|$
- **Путь из v_0 в v_n**: последовательность рёбер, таких, что $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2) \ldots e_n = (v_{n-1}, v_n)$.
- **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** $\delta(v_i, v_j)$: длина кратчайшего пути из v_i в v_j.

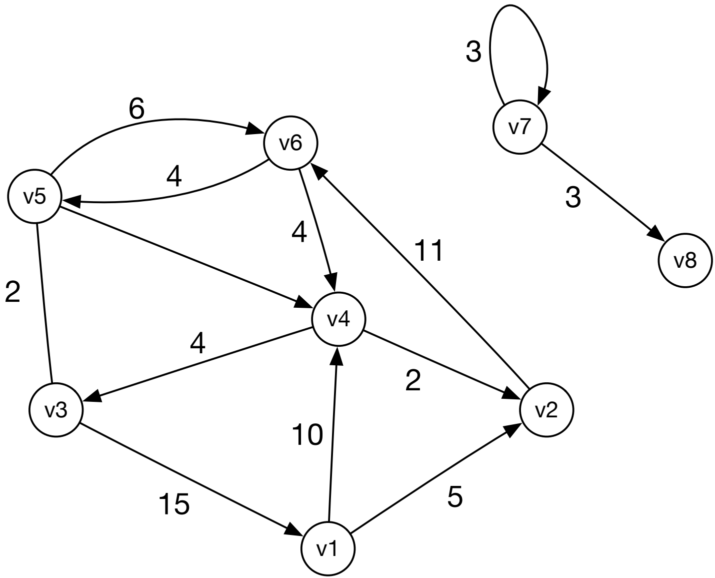
$$\delta(u, v) = 0 \Leftrightarrow u = v$$

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

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

Графы: основные термины

Пример ориентированного графа



Типичные: задачи на графах

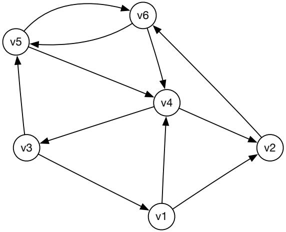
- Проверка графа на связность.
- Является ли граф деревом.
- Найти кратчайший путь из u в v.
- Найти цикл, проходящий по всем рёбрам ровно один раз (цикл Эйлера).
- Найти цикл, проходящий по всем вершинам ровно один раз (цикл Гамильтона).
- Проверка на планарность — определить, можно ли нарисовать граф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

Представление графа в памяти.

- Каждой вершине сопоставляется множество смежных с ней.
- Всё представляется в виде матрицы смежности.
- Всё представляется в виде списка рёбер.

Представление графов в памяти

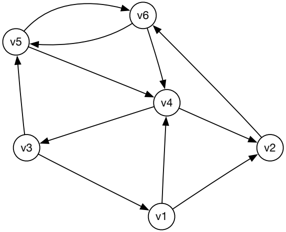
Представление графа в памяти в виде матрицы смежности



	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

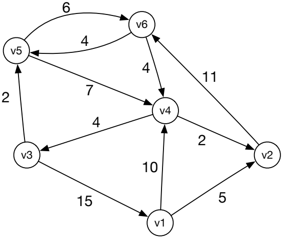
Представление графа в памяти в виде множеств смежности



$v_1 : \{2, 4\}$
$v_2 : \{6\}$
$v_3 : \{1, 5\}$
$v_4 : \{2, 3\}$
$v_5 : \{3, 4, 6\}$
$v_6 : \{4, 5\}$

Представление графов в памяти

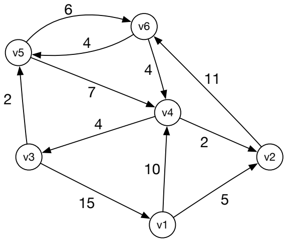
Представление взвешенного графа в памяти в виде матрицы смежности



	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

Представление графов в памяти

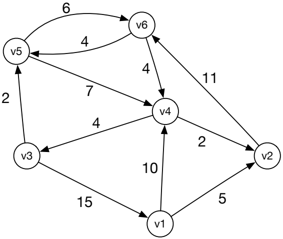
Представление взвешенного графа в памяти в виде множеств смежности



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

Представление графов в памяти

Представление взвешенного графа в памяти в виде списка рёбер



$$\{from, to, cost\} \ldots$$

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

$$\{4,2,2\},\{5,4,7\},\{5,6,6\},\{6,4,4\},\{6,5,4\}\}$$

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 графа

Абстракция: очередь

- **enqueue**: добавить элемент в конец очереди
- **dequeue**: извлечь с удалением элемент из начала очереди.

Ещё один термин:

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

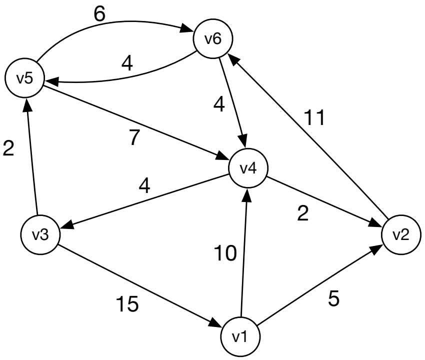
Обход графа: поиск в ширину, BFS

Поиск в ширину от вершины s — просмотр вершин графа в порядке возрастания расстояния от s.

procedure $\mathrm{BFS}(G : Graph, s : Vertex)$
 for all $u \in V[G] \setminus \{s\}$ **do**
  $d[u] \leftarrow \infty$; $c[u] \leftarrow$ white; $\pi[u] \leftarrow$ nil
 end for
 $d[s] \leftarrow 0$
 $c[s] \leftarrow$ grey
 Q.enqueue(s)
 while $Q \neq \emptyset$ **do**
  $u \leftarrow$ Q.dequeue()
  for all $v \in Adj[u]$ **do**
   if $c[v] =$ white **then**
    $d[v] \leftarrow d[u] + 1$
    $\pi[v] = u$
    $Q.enqueue(v)$
    $c[v] =$ grey
   end if
  end for
  $c[u] \leftarrow$ black
 end while
end procedure

Прогон алгоритма BFS

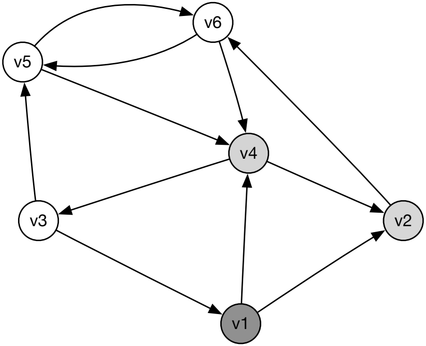
Начало алгоритма.



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

Прогон алгоритма BFS

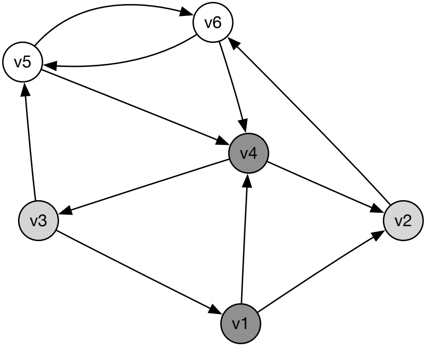
После первого прохождения цикла `While`



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

Прогон алгоритма BFS

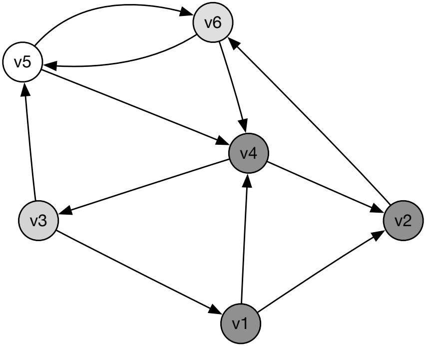
После второго прохождения цикла `While`



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

Прогон алгоритма BFS

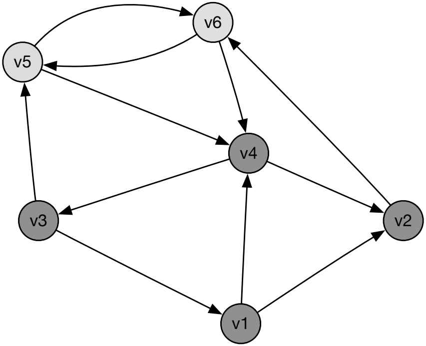
После третьего прохождения цикла `While`



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

Прогон алгоритма BFS

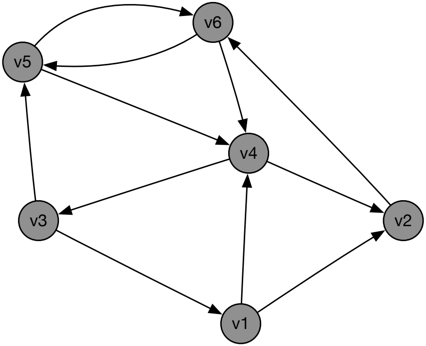
После четвёртого прохождения цикла `While`



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

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

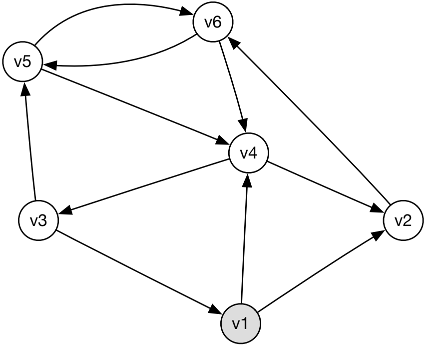
```
procedure DFS(G : Graph)
    for all u ∈ V[G] do
        c[u] ← white;        π[u] ← nil
    end for
    time ← 0
    for all u ∈ V[G] do
        if c[v] = white then
            DFS-vizit(v)
        end if
    end for
end procedure
```

Алгоритм DFS

procedure DFS-vizit(u : *Vertex*)
    $c[u] \leftarrow \text{grey}$
    $time \leftarrow time + 1$
    $d[u] \leftarrow time$
    for all $v \in Adj[u]$ **do**
        if $c[v] = \text{white}$ **then**
            $\pi[v] \leftarrow u$
            DFS-vizit(v)
        end if
    end for
    $c[u] \leftarrow \text{black}$
    $time \leftarrow time + 1$
    $f[u] \leftarrow time$
end procedure

Прогон алгоритма DFS

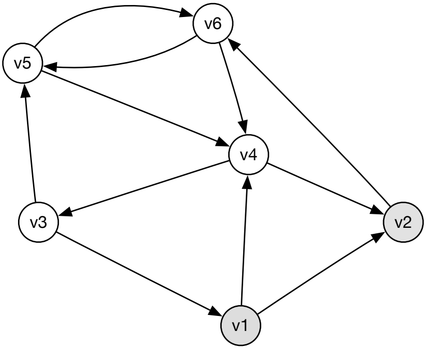
Начинается обход с вершины v_1



$d = \{1, -1, -1, -1, -1, -1\}$
$f = \{-1, -1, -1, -1, -1, -1\}$
$\pi = \{nil, nil, nil, nil, nil, nil\}$

Прогон алгоритма DFS

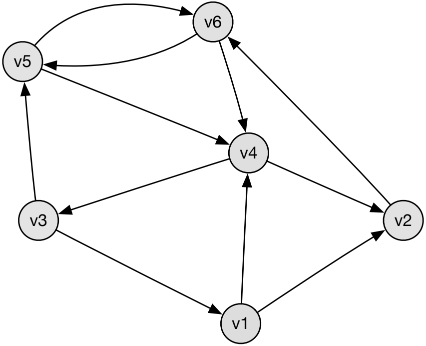
Первый рекурсивный вызов DFS-vizit(v_2)



$d = \{1, 2, -1, -1, -1, -1\}$
$f = \{-1, -1, -1, -1, -1, -1\}$
$\pi = \{nil, v_1, nil, nil, nil, nil\}$

Прогон алгоритма DFS

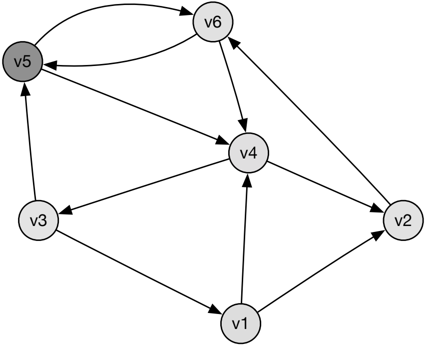
Второй рекурсивный вызов DFS-vizit(v_6)



$$d = \{1, 2, -1, -1, -1, 3\}$$
$$f = \{-1, -1, -1, -1, -1, -1\}$$
$$\pi = \{nil, v_1, nil, nil, nil, v_2\}$$

Прогон алгоритма DFS

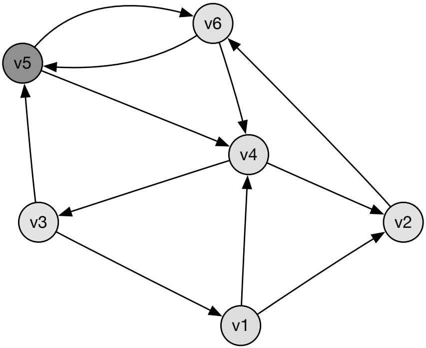
Пятый рекурсивный вызов DFS-vizit(v_5)



$d = \{1, 2, 5, 4, 6, 3\}$
$f = \{-1, -1, -1, -1, -1, -1\}$
$\pi = \{nil, v_1, v_4, v_6, v_3, v_2\}$

Прогон алгоритма DFS

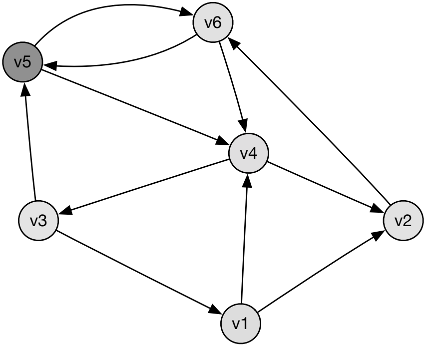
Выход из пятой рекурсии вызов `DFS-vizit`(v_5)



$$d = \{1, 2, 5, 4, 6, 3\}$$
$$f = \{-1, -1, -1, -1, 7, -1\}$$
$$\pi = \{nil, v_1, v_4, v_6, v_3, v_2\}$$

Прогон алгоритма DFS

Завершение алгоритма



$d = \{1, 2, 5, 4, 6, 3\}$
$f = \{12, 11, 8, 9, 7, 10\}$
$\pi = \{nil, v_1, v_4, v_6, v_3, v_2\}$

Алгоритм DFS

- Сложность алгоритма для представления в виде множества смежности равна $O(|V| + |E|)$
- Этот алгоритм — не для нахождения кратчайших маршрутов

Топологическая сортировка

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 `insertToFront`.

Топологическая сортировка

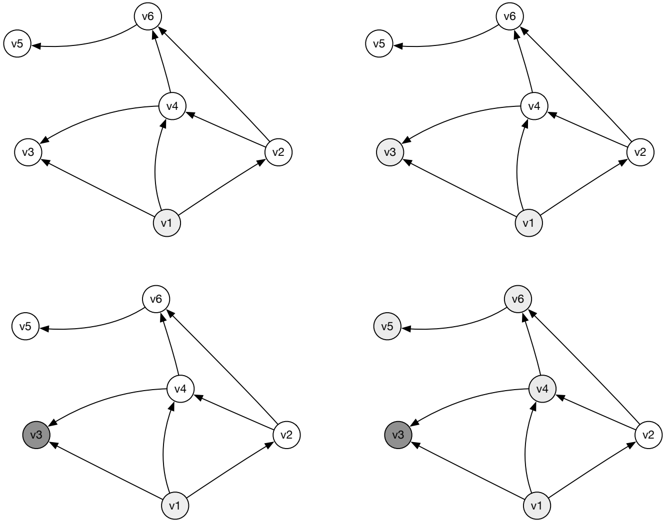
```
procedure TopoSort(G : Graph)
    L ← 0
    for all u ∈ V[G] do
        c[u] ← white;
    end for
    time ← 0
    for all u ∈ V[G] do
        if c[v] = white then
            DFS-vizit(u)
        end if
    end for
end procedure
```

Топологическая сортировка

```
procedure DFS-VIZIT(u : Vertex)
    c[u] ← grey
    for all v ∈ Adj[u] do
        if c[v] = white then
            π[v] ← u
            DFS-vizit(u)
        end if
    end for
    c[u] ← black
    L.insertToFront(u)
end procedure
```

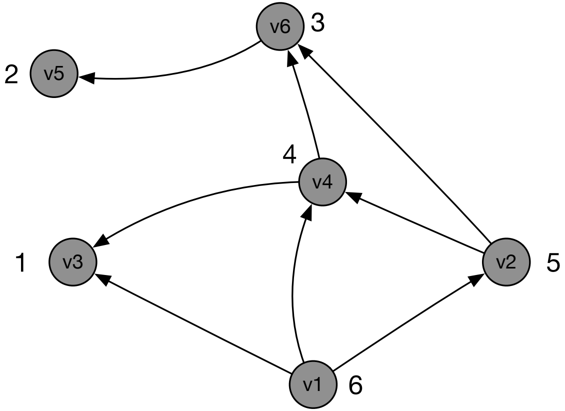
Прогон алгоритма топологической сортировки

Пусть обход начнётся с вершины v_1



Прогон алгоритма топологической сортировки

Результат обхода.



Порядок вершин (добавляем **в начало** по номерам):
$V_1, V_2, V_4, V_6, V_5, V_3$

Остовные деревья

Остовное дерево: ещё немного терминов

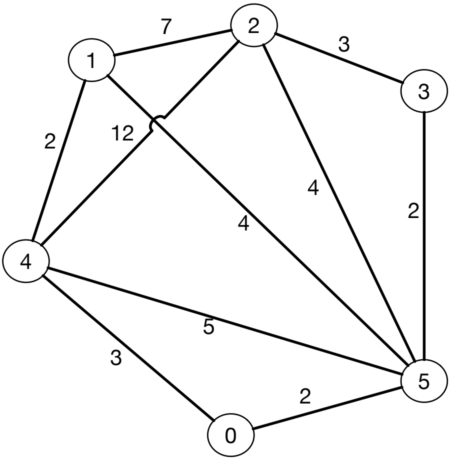
- С точки зрения теории графов **дерево** есть ациклический связный граф.
- Множество деревьев называется **лесом (forest)** или **бором**.
- **Остовное дерево** связного графа — подграф, который содержит все вершины графа и представляет собой полное дерево.
- **Остовный лес** графа — лес, содержащий все вершины графа.

Минимальное остовное дерево

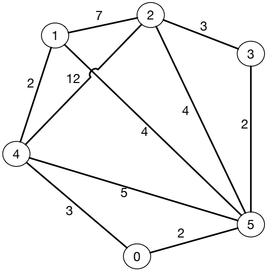
- Построение остовных деревьев — одна из основных задач в компьютерных сетях.
- Решение задачи — как спланировать маршрут от одного узла сети до других.
- Для некоторого типа узлов в передаче сообщений недопустимо иметь несколько возможных маршрутов. Например, если компьютер соединён с маршрутизатором по Wi-Fi и Ethernet одновременно, то в некоторых операционных системах сообщения от компьютера до маршрутизатора не будут доходить из-за наличия цикла.
- Построение остовного дерева — избавление от циклов в графе.

Остовные деревья

Каждый из узлов имеет информацию о связях с соседями.
Каждая связь имеет вес.



Остовные деревья



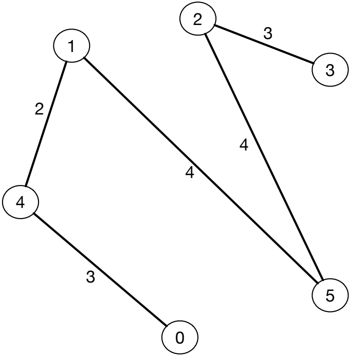
Матрица смежности.

n/m	0	1	2	3	4	5
0	0	0	0	0	3	2
1	0	0	7	0	2	4
2	0	7	0	0	12	4
3	0	0	0	0	0	2
4	3	2	12	0	0	5
5	2	4	4	2	5	0

Остовное дерево

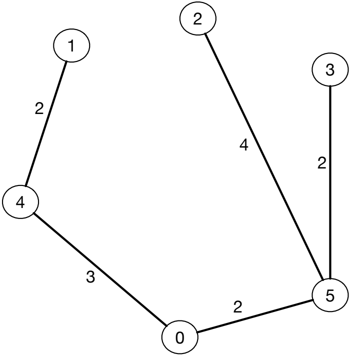
- множество достижимых узлов из некоторого *корневого* узла P_r должно совпадать с полным множеством.
- для каждого узла должен быть ровно один маршрут до любого из достижимых узлов.

Это — остовное дерево для корневого узла P_r.



Минимальное остовное дерево

Задача: определение кратчайшего пути из корневого узла.
Минимальное остовное дерево



Минимальное остовное дерево

- `MST` — Minimal Spanning Tree.
- Минимальное остовное дерево взвешенного графа есть остовное дерево, вес которого (сумма его всех рёбер) не превосходит вес любого другого остовного дерева.
- Именно минимальные остовные деревья больше всего интересуют проектировщиков сетей.
- **Сечение графа** — разбиение множества вершин графа на два непересекающихся подмножества.
- **Перекрёстное ребро** — ребро, соединяющее вершину одного множества с вершиной другого множества.

Свойства MST

- **Лемма.** Если T — произвольное остовное дерево, то добавление любого ребра e между двумя вершинами u и v создаёт цикл, содержащий вершины u, v и ребро e.

Свойства MST

- **Лемма.** При любом сечении графа каждое минимальное перекрёстное ребро принадлежит некоторому MST-дереву и каждое MST-дерево содержит перекрёстное ребро.
- **Доказательство** от противного. Пусть e — минимальное перекрёстное ребро, не принадлежащее ни одному MST и пусть T — MST дерево, не содержащее e. Добавим e в T. В этом графе есть цикл, содержащий e и он содержит ребро e', с весом, не меньшим e. Если удалить e', то получится остовное дерево не большего веса, что противоречит условию минимальности T или предположению, что e не содержится в T.

Свойства MST

- **Следствие.** Каждое ребро дерева MST есть минимальное перекрёстное ребро, определяемое вершинами поддеревьев, соединённых этим ребром.

Свойства MST

- **Лемма (без доказательства).** Пусть имеется граф G и ребро e. Пусть граф G' есть граф, полученный добавлением ребра e к графу G. Результатом добавления ребра e в `MST` графа G и последующего удаления максимального ребра из полученного цикла будет `MST` графа G'.
- Эта лемма выявляет рёбра, которые не должны входить в `MST`.

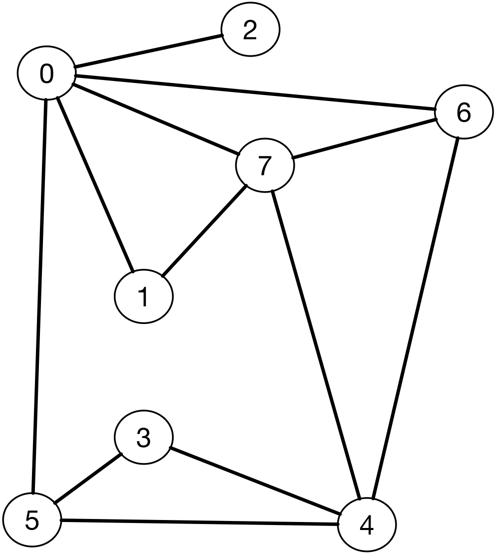
Алгоритмы поиска MST

Алгоритм Прима

1. Один из наиболее простых алгоритмов для нахождения MST.
2. Используется сечение графа на два подграфа — древесных вершин и недревесных вершин.
3. Выбираем произвольную вершину. Это — MST дерево, состоящее из одной древесной вершины.
4. Выбираем минимальное перекрёстное ребро между MST вершиной и недревесным множеством.
5. Повторяем операцию до тех пор, пока все вершины не окажутся в дереве.

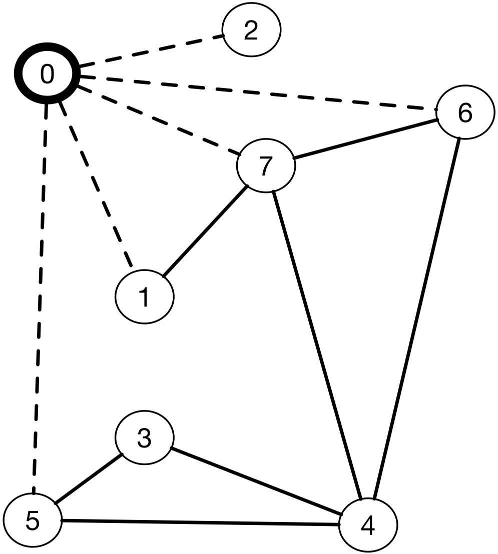
Алгоритм Прима

Исходный граф.



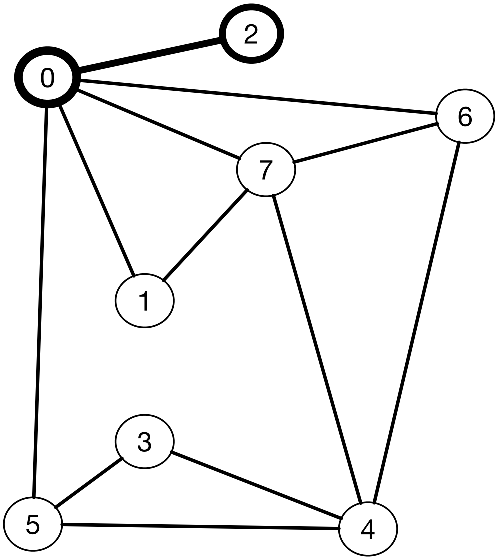
Алгоритм Прима

Вершина `0` — корневая. Переводим её в `MST`. Проверяем все веса из `MST` в не `MST`.



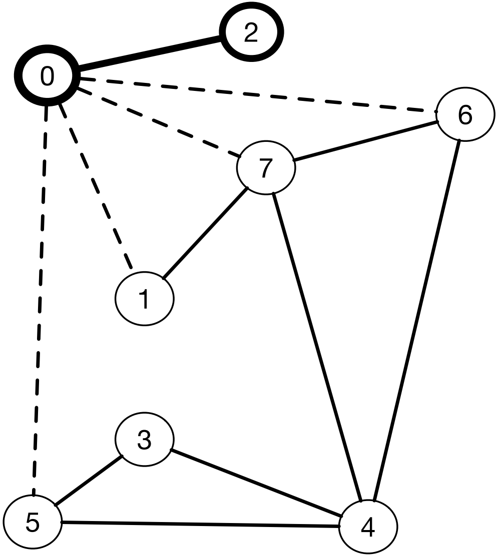
Алгоритм Прима

(0–2) самое лёгкое ребро. Переводим вершину 2 и ребро (0–2) в MST.



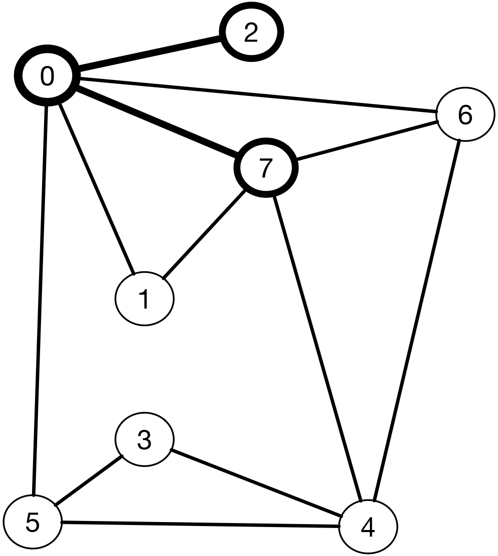
Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



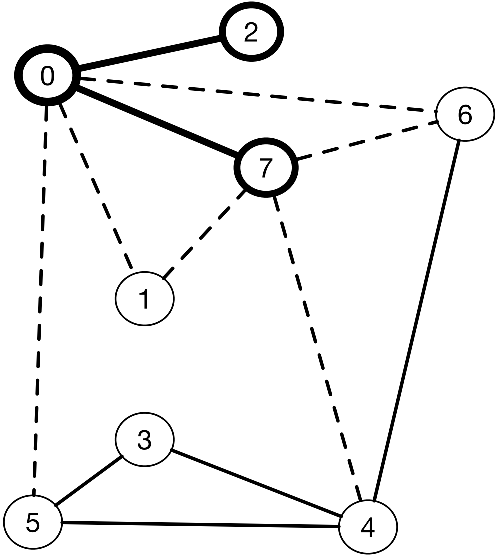
Алгоритм Прима

Переносим вершину 7 и ребро (0-7) в MST.



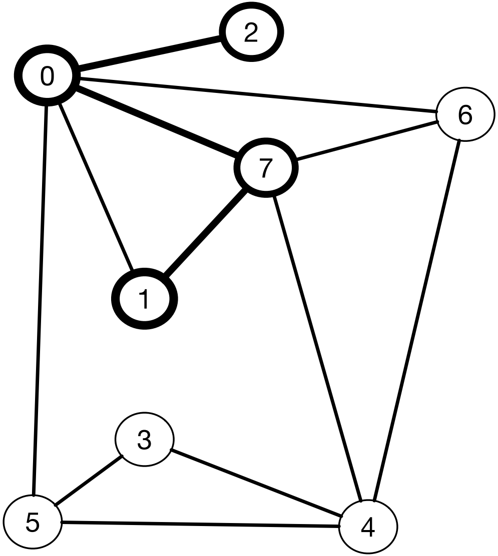
Алгоритм Прима

Отмечаем все рёбра из `MST` в не `MST`.



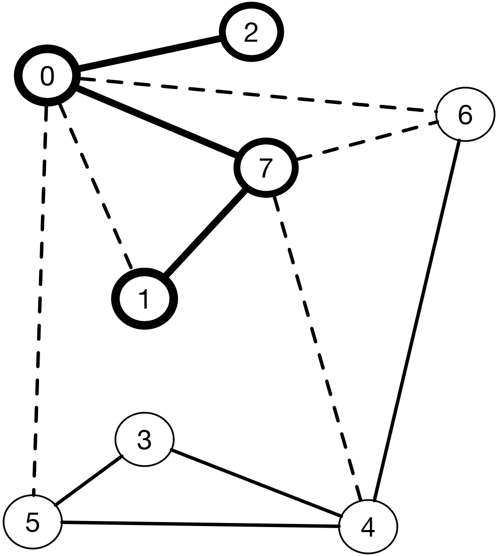
Алгоритм Прима

Переносим вершину 1 и ребро (1-7) в MST.



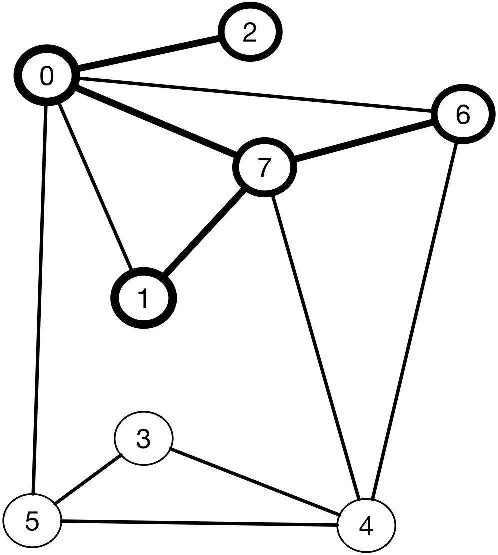
Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



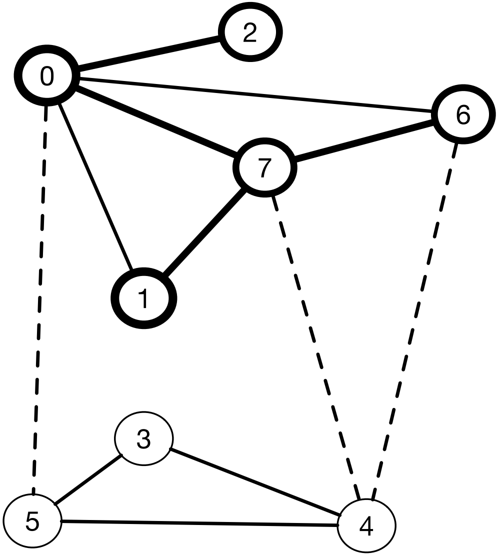
Алгоритм Прима

Переносим вершину 6 и ребро (7-6) в MST.



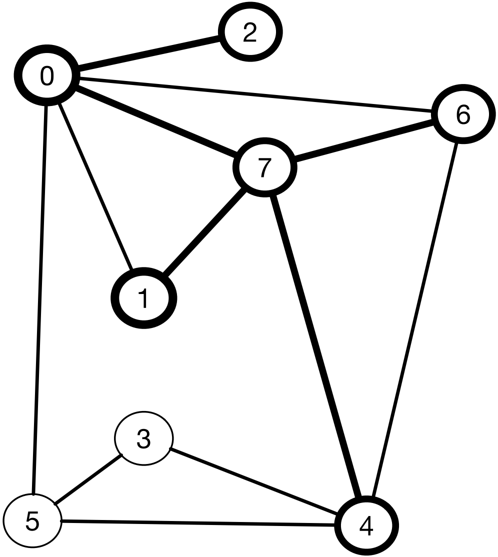
Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



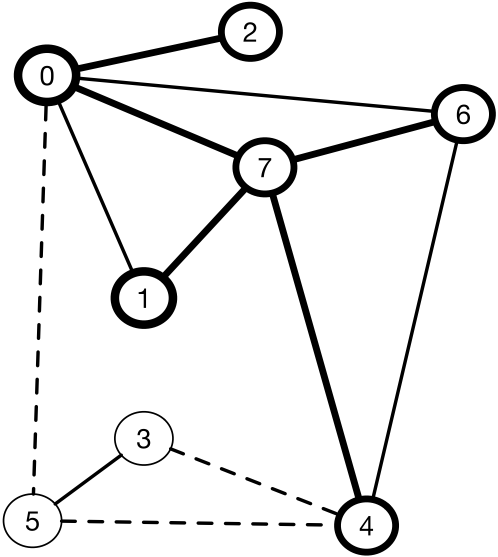
Алгоритм Прима

Переносим вершину 4 и ребро (7-4) в `MST`.



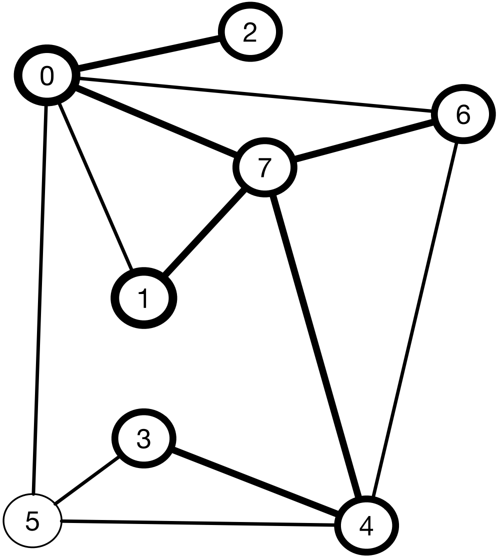
Алгоритм Прима

Отмечаем все рёбра из `MST` в не `MST`.



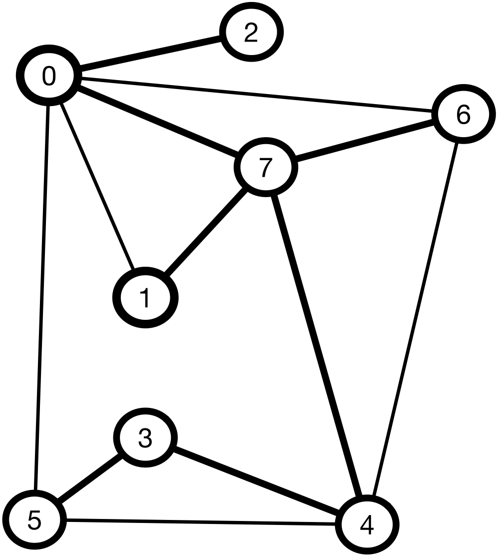
Алгоритм Прима

Переносим вершину 3 и ребро (3-4) в MST.



Алгоритм Прима

Все вершины в `MST`.



Алгоритм Прима

- В данном виде алгоритм не очень эффективен.
- На каждом шаге мы забываем про те рёбра, который уже проверяли.
- Введём понятие **накопителя**.
- Накопитель содержит множество рёбер-кандидатов.
- Каждый раз в `MST` включается самое лёгкое ребро.
- Сложность алгоритма $O(|V|^2)$

Алгоритм Прима

Более эффективная реализация алгоритма Прима

1. Выбираем произвольную вершину. Это — `MST` дерево, состоящее из одной вершины. Делаем вершину текущей.
2. Помещаем в накопитель все рёбра, которые ведут из этой вершины в не `MST` узлы. Если в какой-либо из узлов уже ведёт ребро с большей длиной, заменяем его ребром с меньшей длиной.
3. Выбираем ребро с минимальным весом из накопителя.
4. Повторяем операцию до тех пор, пока все вершины не окажутся в дереве.

Алгоритм Прима

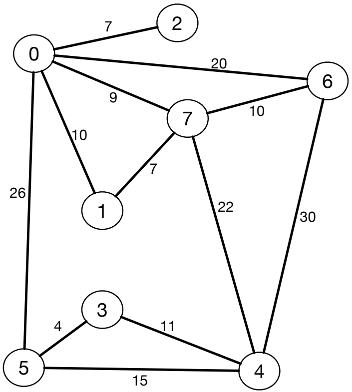
- Алгоритм Прима — обобщение поиска на графе.
- Накопитель представляется очередью с приоритетами.
- Используется операция «извлечь минимальное».
- Используется операция «увеличить приоритет».
- Такой поиск на графе называется `PFS` — поиск по приоритету.
- Сложность алгоритма $O(|E| \log |V|)$

Алгоритм Краскала

Алгоритм Краскала (Kruscal).

- Один из самых старых алгоритмов на графах (1956).
- Предварительное условие: связность графа.
  1. Создаётся число непересекающихся множеств по количеству вершин и каждая вершина составляет своё множество.
  2. Множество `MST` вначале пусто.
  3. Из всех рёбер, не принадлежащих `MST` выбирается самое короткое из всех рёбер, не образующих цикл. Вершины ребра должны принадлежать различным множествам.
  4. Выбранное ребро добавляется к множеству `MST`
  5. Множества, которым принадлежат вершины выбранного ребра, сливаются в единое.
  6. Если размер множества `MST` стал равен $|V| - 1$, то алгоритм завершён, иначе отправляемся к пункту 3.

Алгоритм Краскала



Список рёбер упорядочен по возрастанию:

i	3	0	1	0	0	6	3	4	0	0	4
j	5	2	7	7	1	7	4	5	6	5	6
W_{ij}	4	7	7	9	10	10	11	15	22	26	30

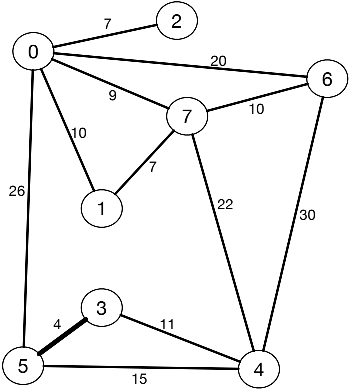
Алгоритм Краскала

Таблица принадлежности вершин множествам:

V_i	0	1	2	3	4	5	6	7
p	0	1	2	3	4	5	6	7

Алгоритм Краскала:первая итерация

Вершины 3 и 5 самого короткого ребра в разных множествах $\rightarrow$ отправляем ребро в множество `MST` и объединяем множества.



V_i	0	1	2	3	4	5	6	7
p	0	1	2	3	4	**3**	6	7

Алгоритм Краскала:вторая итерация

Два подходящих ребра с одинаковым весом:

i	0	1	0	0	6	3	4	0	0	4
j	2	7	7	1	7	4	5	6	5	6
W_{ij}	7	7	9	10	10	11	15	22	26	30

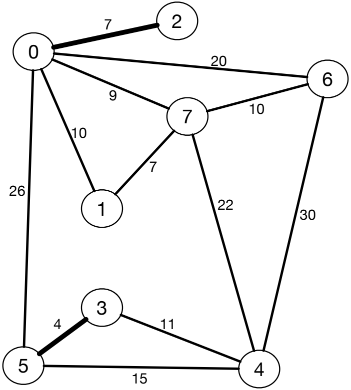
Лемма. При равных подходящих рёбрах можно выбирать произвольное.

Доказательство. Если добавление первого ребра не помешает добавлению второго, то, всё ОК.

Если помешает, (добавление второго создаст цикл), то можно удалить любое из них, общий вес дерева останется неизменным.

Алгоритм Краскала:вторая итерация

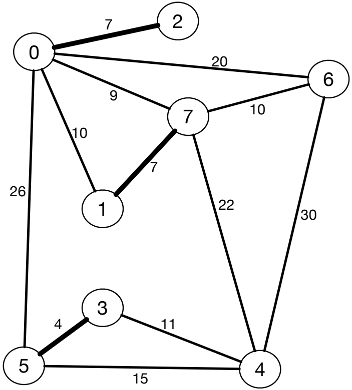
Выберем ребро (0,2) и поместим вершину 2 в множество номер 0.



V_i	0	1	2	3	4	5	6	7
p	0	1	**0**	3	4	3	6	7

Алгоритм Краскала:третья итерация

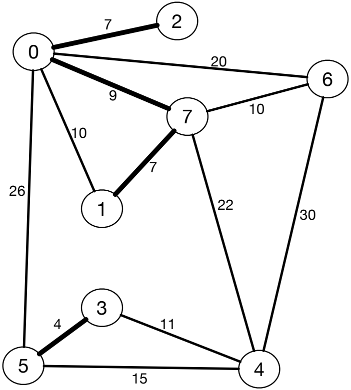
Ребро (1,7) привело к слиянию множеств 1 и 7.



V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	**1**

Алгоритм Краскала:четвёртая итерация

Самым коротким ребром из оставшихся оказалось ребро (0,7).



Нам нужно слить два множества — одно, содержащее $\{0, 2\}$ и другое — содержащее $\{1, 7\}$.

Алгоритм Краскала: четвёртая итерация

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	**1**

- Пусть новое множество получит номер 0.
- Нужно ли найти в массиве p все единицы (номер второго множества) и заменить их на нули (номер того множества, куда переходят элементы первого)?
- Можно быстрее, используя *систему непересекающихся множеств* `Union-Find` или `Disjoint Set Union`, DSU.

Система непересекающихся множеств, DSU

Абстракция `DSU` реализует три операции:

- `create(n)` — создать набор множеств из n элементов.
- `find_root(x)` — найти представителя множества.
- `merge(l,r)` — сливает два множества `l` и `r`.

Система непересекающихся множеств

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	1

- `create(n)` — Массив p есть массив представителей.
- `find_root(x)`.

Система непересекающихся множеств

- `p[7]==1` — номер множества, он же и *представитель* .
- Если для слияния множеств `{0,2}` и `{1,7}` поместим в `p[7]` число 0, то для вершины 1 представителем останется 1, что неверно (после слияния вершина 1 должна принадлежать множеству 0).
- Так как `p[7]==1`, то и у седьмой, и у первой вершины представители одинаковые.
- Если номер вершины совпадает с номером представителя, то, в массив `p` при исполнении ничего не было записано $\rightarrow$ эта вершина есть корень дерева.
- После слияния нужно заменить всех родителей вершины на нового представителя. Это делается изящным рекурсивным алгоритмом:

```
int find_root(int r) {
    if (p[r] == r) return r; // Тривиальный случай
    return p[r] = find_root(p[r]); Рекурсивный случай
}
```

Система непересекающихся множеств

- `merge(l,r)`. Для сохранения корректности алгоритма вполне достаточно любого из присвоений: `p[l] = r` или `p[r] = l`. Всю дальнейшую корректировку родителей в дальнейшем сделает метод `find_root`.
- Приёмы балансировки деревьев:
  - Использование ещё одного массива, хранящего длины деревьев: слияние производится к более короткому дереву.
  - Случайный выбор дерева-приёмника.

```
void merge(int l, int r) {
   l = find_root(l); r = find_root(r);
   if (rand() % 1) p[l] = r;
   else p[r] = l;
}
```

- Важно: что внутри операции слияния имеется операция поиска, которая заменяет аргументы значениями корней их деревьев!

Алгоритм Краскала

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	1

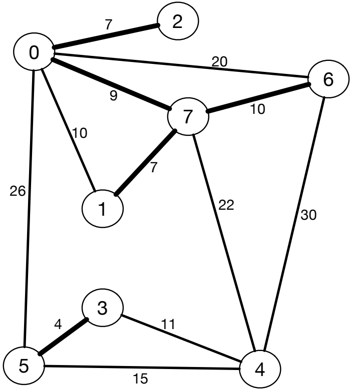
- Определяем, каким деревьям принадлежат концы ребра `(0,7)`.
- `find_root(0)` вернёт 0 как номер множества.
- `find_root(7)` сначала убедится, что в `p[7]` лежит 1 и вызовет `find_root(1)`, после чего, возможно, заменит `p[7]` на 1.

Алгоритм Краскала

- Концы ребра `0` принадлежат разным множествам $\to$ сливаем множества, вызвав `merge(0,7)`.
- Операция `merge` — заменит свои аргументы, 0 и 7, корнями деревьев, которым принадлежат 0 и 7, то есть, 0 и 1 соответственно.
- В `p[1]` помещается 0 и деревья слиты.
- Обратите внимание на то, что в `p[7]` всё ещё находится 1!.

V_i	0	1	2	3	4	5	6	7
p	0	**0**	0	3	4	3	6	1

Алгоритм Краскала

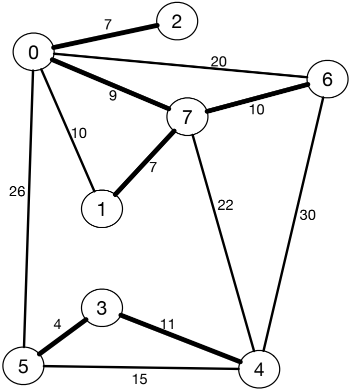


Следующее ребро — `(6,7)`. `find_root(7)` установит `p[7]=0`.
Это же значение будет присвоено и `p[6]`.

V_i	0	1	2	3	4	5	6	7
p	0	**0**	0	3	4	3	**0**	0

Алгоритм Краскала

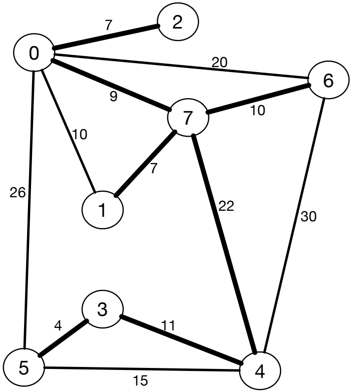
На следующем этапе ребро `(3,4)` окажется самым коротким.



V_i	0	1	2	3	4	5	6	7
p	0	0	0	3	3	3	0	0

Алгоритм Краскала

Концы рёбер `(4,5)` и `(0,6)` принадлежат одним множествам. Ребро `(4,7)` подходит. Количество рёбер в множестве `MST` достигло $7 = N - 1$. Конец



V_i	0	1	2	3	4	5	6	7
p	0	0	0	3	0	3	0	0

Алгоритм Краскала: сложность

- Первая часть алгоритма — сортировка рёбер. Сложность этой операции $O(|E| \log |E|)$.
- В 1984 году Tarjan доказал, используя функцию Аккермана, что операция поиска в `DSU` имеет сложность `O(1)`.
- Сложность всего алгоритма Краскала и есть $O(|E| \log |E|)$.
- Для достаточно разреженных графов он обычно быстрее алгоритма Прима, для заполненных — наоборот.

Алгоритм Дейкстры.

Дерево кратчайших путей — SPT

- Пусть задан граф G и вершина s. **Дерево кратчайших путей** для s — подграф, содержащий s и все вершины, достижимые из s, образующий направленное поддерево с корнем в s, где каждый путь от вершины s до вершины u является кратчайшим из всех возможных путей.

Алгоритм Дейкстры

- Строит `SPT` (Shortest Path Tree).
- Определяет длины кратчайших путей от заданной вершины до остальных.
- Веса рёбер могут принимать произвольные значения (включая отрицательные).
- **Обязательное условие:** граф не должен содержать циклов с отрицательным весом.

Алгоритм Дейкстры

1. В SPT заносится корневой узел (исток).
2. На каждом шаге в SPT добавляется одно ребро, которое формирует кратчайший путь из истока в не-SPT.
3. Вершины заносятся в SPT в порядке их расстояния по SPT от начальной вершины.

Алгоритм Дейкстры

Жадная стратегия

- Пусть найдено оптимальное множество U.
- Изначально оно состоит из вершины s
- Длины кратчайших путей до вершин множества обозначим, как $d(s, v), v \in U$.
- Среди вершин, смежных с U находим вершину $u, u \notin U$ такую, что достигается минимум

$$\min_{v \in U, u \notin U} d(s, v) + w(v, u)$$

- Обновляем множество $U : U \leftarrow U \cup \{u\}$ и повторяем операцию.

Алгоритм Дейкстры

Используются переменные

- $d[u]$ — длина кратчайшего пути из вершины s до вершины u
- $\pi[u]$ — предшественник u в кратчайшем пути от s
- $w(u, v)$ — вес пути из u в v (длина ребра, вес ребра, метрика пути)
- Q — приоритетная по значению d очередь узлов на обработку
- U — множество вершин с уже известным финальным расстоянием

Алгоритм Дейкстры

procedure $\textsc{Dijkstra}(G : Graph; w : weights; s : Vertex)$
    for all $v \in V$ **do**
        $d[v] \leftarrow \infty$
        $\pi[v] \leftarrow nil$
    end for
    $d[s] \leftarrow 0$
    $U \leftarrow \emptyset$
    $Q \leftarrow V$
    while $Q \neq \emptyset$ **do**
        $u \leftarrow Q.extractMin()$
        $U \leftarrow U \cup \{u\}$
        for all $v \in Adj[u], v \notin U$ **do**
            Relax(u,v)
        end for
    end while
end procedure

Алгоритм Дейкстры

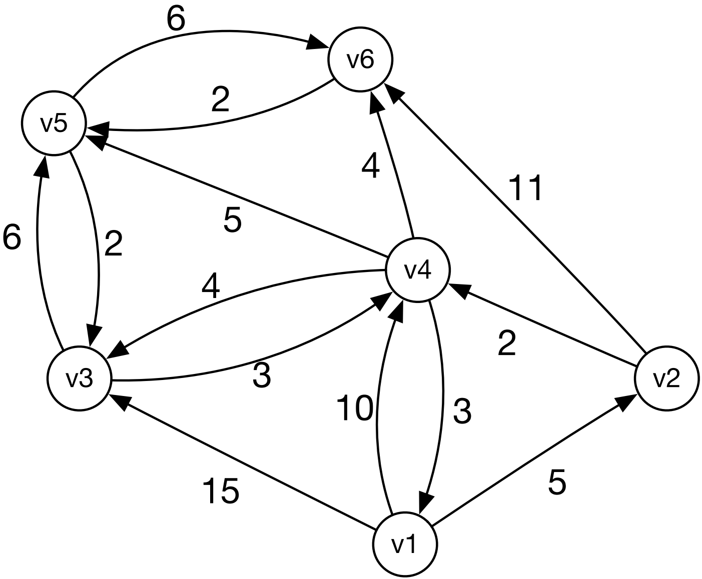
procedure $\textsc{Relax}(u, v : \mathit{Vertex})$
    if $d[v] > d[u] + w(u, v)$ **then**
        $d[v] = d[u] + w(u, v)$
        $\pi[v] \leftarrow u$
    end if
end procedure

Алгоритм Дейкстры

- Операция *Relax* — релаксация
- Два вида релаксации:
  - Релаксация ребра. Даёт ли продвижение по данному ребру новый кратчайший путь?
  - Релаксация пути. Даёт ли прохождение через данную вершину новый кратчайший путь, соединяющий две другие заданные вершины.

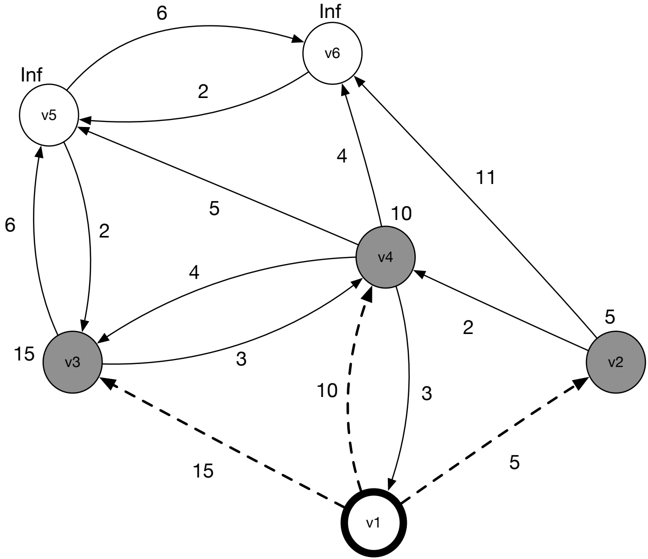
Алгоритм Дейкстры

Исходный граф



Алгоритм Дейкстры

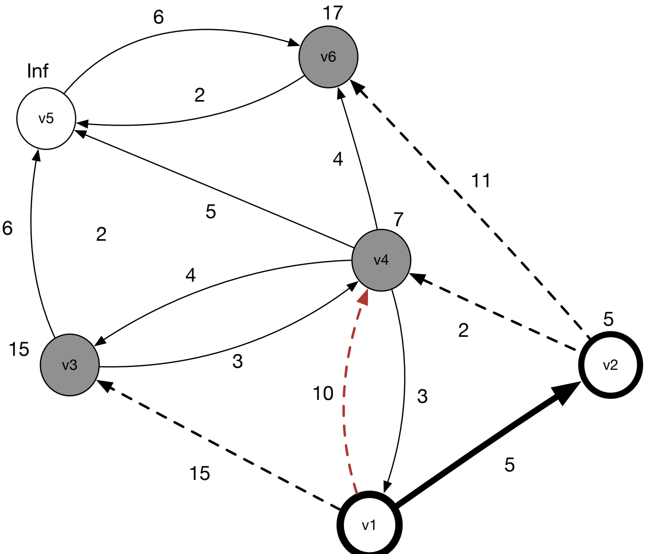
v1 в SPT, v2, v3 и v4 — в накопителе.



Алгоритм Дейкстры

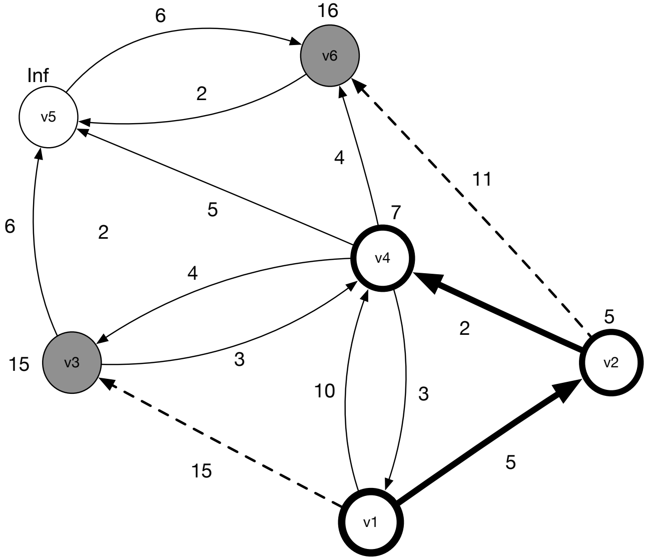
Выбран узел v2. Корректируются расстояния от него.

Релаксация: $(1 \to 4)$ заменён на $(1 \to 2 \to 4)$.



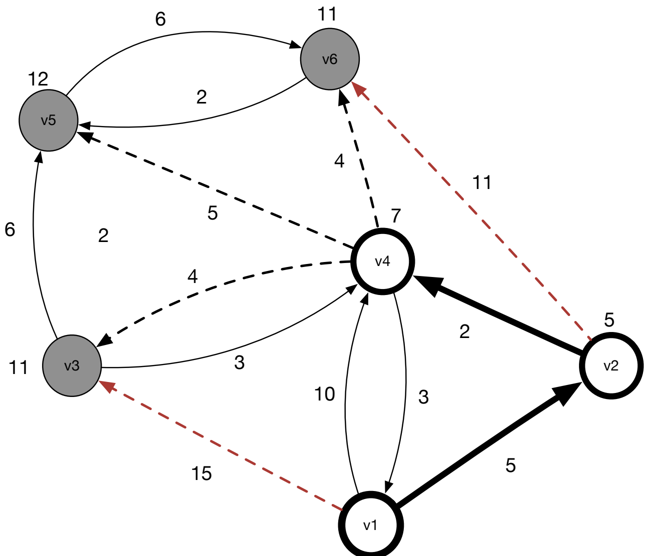
Алгоритм Дейкстры

Выбран узел v3.

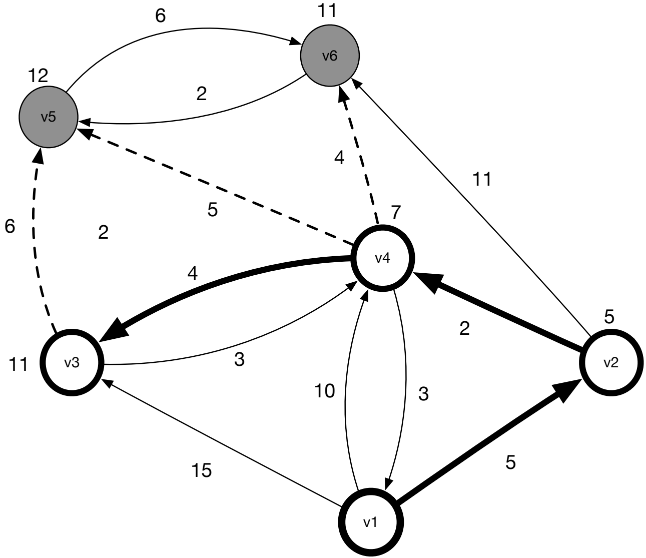


Алгоритм Дейкстры

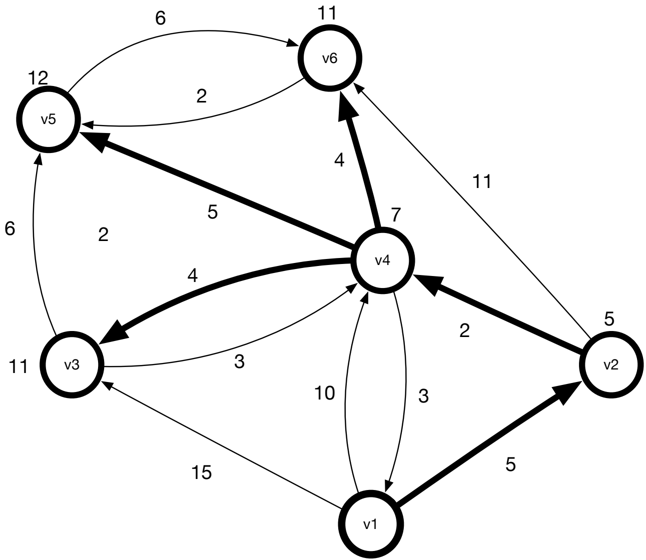
В накопитель отправляется $v5$. Релаксация: $(1 \to 2 \to 6)$ заменено на $(1 \to 2 \to 4 \to 6)$, $(1 \to 3)$ на $(1 \to 2 \to 4 \to 3)$



Алгоритм Дейкстры



Алгоритм Дейкстры



Алгоритм Дейкстры: сложность

- Имеется $|V| - 1$ шаг.
- На каждом шаге корректировка расстояние до соседей и выбор минимального из накопителя.
- Для насыщенных деревьев сложность алгоритма $O(V^2 \log V)$

Алгоритм Дейкстры: альтернативная реализация

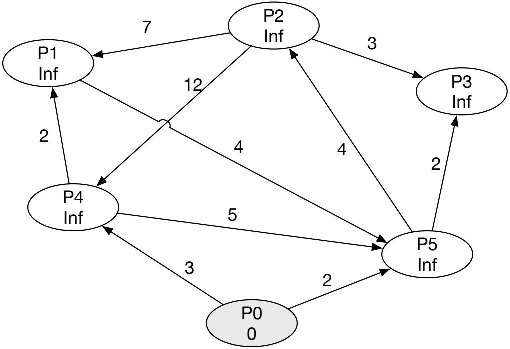
- Для разреженных графов можно воспользоваться вариантом PFS.
- Используется очередь необработанных рёбер.
- Используется операция «удаление минимального»

Алгоритм Дейкстры: альтернативная реализация

function DIJKSTRA($C : AdjacencyMatrix, root : int$): set of double
    $Q : Set\ of\ Pair(double, int)$
    $const\ BIGINT \leftarrow 10^{100}$
    $D : vector\ of\ double;$
    $D.fill(C.size(), BIGINT); D[root] \leftarrow 0$
    $Q.insert(0, root)$
    while $not\ Q.empty()$ **do**
        $top = Q.first()$ ▷ Fetch any element of set
        $Q.erase(top)$ ▷ Clear the fetched element
        $v \leftarrow top.second$
        $d \leftarrow top.first$
        for all $v2 \in C[v]$ **do**
            $cost \leftarrow v2.second.cost$
            if $(D[v2.first] > D[v] + cost)$ **then**
                if $(D[v2.first] < BIGINT)$ **then**
                    $Q.erase(Q.find(D[v2.first], v2.first)))$
                end if
                $D[v2.first] \leftarrow D[v] + cost$
                $Q.insert(D[v2.first], v2.first))$
            end if
        end for
    end while
    return D
end function

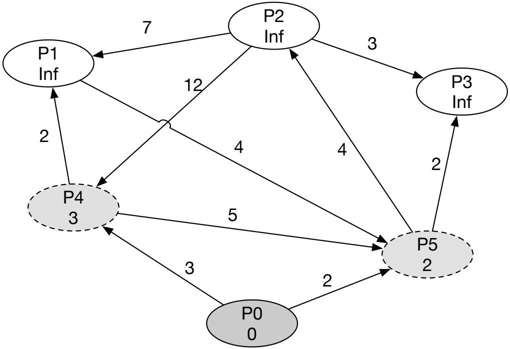
Альтернативная реализация алгоритм Дейкстры: начальное состояние

$root = 0$
$D = \{0, \infty, \infty, \infty, \infty, \infty\}$
$Q = \{H_0\}$



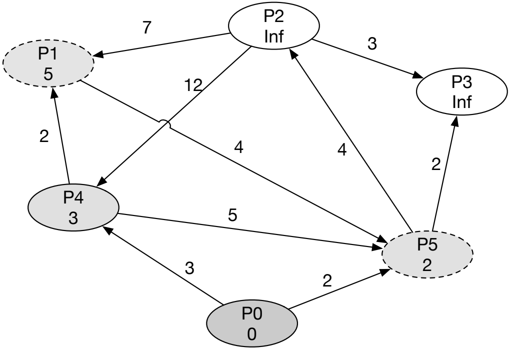
Альтернативный алгоритм Дейкстры: после первой итерации

$root = 0$
$D = \{0, \infty, \infty, \infty, 3, 2\}$
$Q = \{H_4, H_5\}$



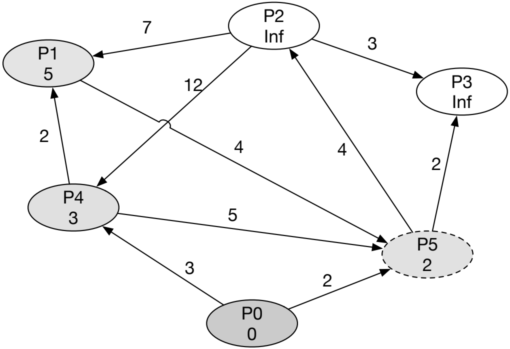
Альтернативный алгоритм Дейкстры: после второй итерации

$root = 0$
$D = \{0, 5, \infty, \infty, 3, 2\}$
$Q = \{H_1, H_5\}$



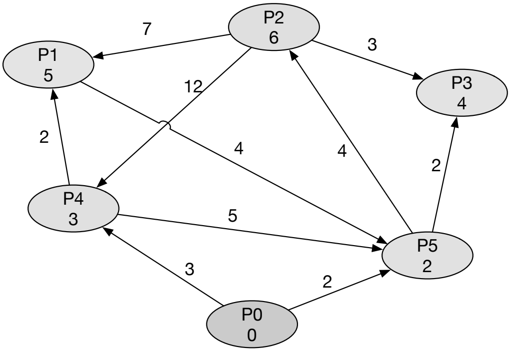
Альтернативный алгоритм Дейкстры: после третьей итерации

$root = 0$
$D = \{0, 5, \infty, \infty, 3, 2\}$
$Q = \{H_5\}$



Альтернативный алгоритм Дейкстры: результат

$root = 0$
$D = \{0, 5, 6, 4, 3, 2\}$
$Q = \{\}$



Альтернативный алгоритм Дейкстры: сложность

- Требуется $|E|$ операций.
- В каждой операции присутствует операция «удаление минимального» из приоритетной очереди максимального размера $|V|$.
- Сложность алгоритма $|E| \log |V|$

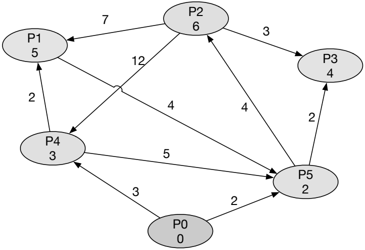
Алгоритм Дейкстры: применения

Задача: для заданного узла построить таблицу маршрутов ко всем достижимым узлам.

- **Маршрут** к узлу есть путь кратчайшего веса.
- Модификация алгоритма Дейкстры позволяет построить таблицу предшественников для каждого из узлов.
- Имеется таблица предшественников — строится таблица маршрутов.

Алгоритм Дейкстры

Построим такие таблицы маршрутов узлов дерева:



Пункт назначения	Сосед
P_0	-
P_2	P_5
P_3	P_5
P_4	P_5
P_5	P_5

Таблица маршрутов для узла P_1

Алгоритм Дейкстры: применения

Остальные таблицы строятся аналогично.
Совокупная таблица маршрутов: недостижимые узлы отмечены знаком минус, узлы «откуда» перечислены в верхней строке, узлы «куда» — в соответствующем столбце.

Откуда	P_0	P_1	P_2	P_3	P_4	P_5
P_0	P_0	-	-	-	-	-
P_1	P_5	P_1	P_1	-	-	P_2
P_2	P_5	P_5	P_2	-	-	P_2
P_3	P_5	P_5	P_3	P_3	-	P_3
P_4	P_4	P_5	P_4	-	P_4	P_4
P_5	P_5	P_5	P_1	-	-	P_5

Таблица маршрутов для всех узлов

Множественный алгоритм Дейкстры

- Вычисление таблиц для каждого узла в отдельности имеет сложность $O(N^2 \log N)$.
- Вычисление таблиц для всех узлов требует времени $N \cdot O(N^2 \log N) = O(N^3 \log N)$
- Существует более быстрый алгоритм, имеющий сложность $O(N^3)$.

Алгоритм Флойда-Уоршалла.

Алгоритм Флойда-Уоршалла

Построение таблиц маршрутизации.

- Известен с 1962 года.
- Определяет кратчайшие пути во взвешенном графе, описанном матрицей смежности.
- В матрице смежности число, находящееся в i-й строке и j-м столбце есть вес связи между ними.
- Изменим представление и будем полагать, что в матрице смежности $C_{ij} = \infty$, если узлы i и j не являются соседями.
- На входе алгоритм принимает модифицированную матрицу смежности, а на выходе эта матрица будет содержать в элементе C_{ij} вес кратчайшего пути из P_i в P_j.
- Допускается наличие путей с отрицательным весом.
- Не должно быть циклов с отрицательной длиной.

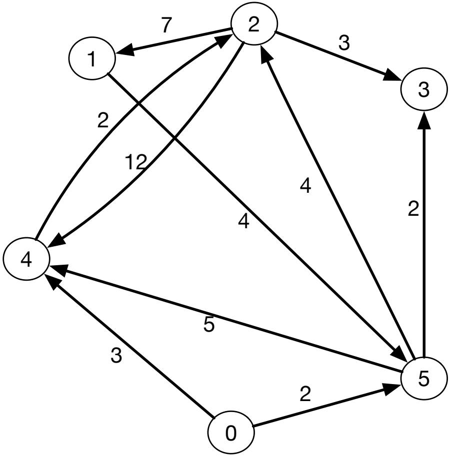
Алгоритм Флойда-Уоршалла

Сам алгоритм может быть описан в рекурсивной форме как

$$D_{ij}^{(k)} = \begin{cases} C_{ij}, & \text{если } k = 0, \\ \min\left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right), & \text{если } k \geqslant 1 \end{cases}$$

Это — задача динамического программирования.

Этапы прохождения алгоритма для графа



Алгоритм Флойда-Уоршалла

Исходная матрица связей:

$$D^{(0)} = \begin{array}{|c|cccccc|} \hline & P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\ \hline P_0 & 0 & \infty & \infty & \infty & 3 & 2 \\ P_1 & \infty & 0 & \infty & \infty & \infty & 4 \\ P_2 & \infty & 7 & 0 & 3 & 12 & \infty \\ P_3 & \infty & \infty & \infty & 0 & \infty & \infty \\ P_4 & \infty & \infty & 2 & \infty & 0 & \infty \\ P_5 & \infty & \infty & 4 & 2 & 5 & 0 \\ \hline \end{array}$$

Алгоритм Флойда-Уоршалла

Перед началом алгоритма матрица наилучших соседей инициализируется следующим образом:

$$B_{ij}^{(0)} = \begin{cases} j, & \text{если } C_{ij} \neq 0, \\ NIL, & \text{если } C_{ij} = 0. \end{cases}$$

Для удобства записи будем обозначать в таблице отсутствие пути символом `'-'`, а не символом `NIL`.

Алгоритм Флойда-Уоршалла

$B^{(0)} =$

	P_0	P_1	P_2	P_3	P_4	P_5
P_0	0	-	-	-	4	5
P_1	-	1	-	-	-	5
P_2	-	1	2	3	4	-
P_3	-	-	-	3	-	-
P_4	-	-	-	-	4	-
P_5	-	-	2	3	4	5

Начальная матрица $D^{(0)}$ содержит метрики всех наилучших маршрутов единичной длины, в матрица $B^{(0)}$ — наилучших соседей для этих маршрутов. Каждая следующая итерация алгоритма добавляет в матрицу $B^{(i+1)}$ элементы, связанные с маршрутами длины i, на единицу большей а в матрицу $D^{(i+1)}$ — метрики этих маршрутов.

Алгоритм Флойда-Уоршалла

После первой итерации матрицы не изменяются.
После второй итерации получается следующее (жирным шрифтом помечены изменившиеся элементы таблиц):

$$D^{(2)} = \begin{array}{|c|cccccc|}
\hline
 & P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\
\hline
P_0 & 0 & \infty & \infty & \infty & 3 & 2 \\
P_1 & \infty & 0 & \infty & \infty & \infty & 4 \\
P_2 & \infty & 7 & 0 & 3 & 12 & \mathbf{11} \\
P_3 & \infty & \infty & \infty & 0 & \infty & \infty \\
P_4 & \infty & \infty & \infty & \infty & 0 & \infty \\
P_5 & \infty & \infty & 4 & 2 & 5 & 0 \\
\hline
\end{array}$$

Алгоритм Флойда-Уоршалла

$B^{(2)} =$

	P_0	P_1	P_2	P_3	P_4	P_5
P_0	0	-	-	-	4	5
P_1	-	1	-	-	-	5
P_2	-	1	2	3	4	**1**
P_3	-	-	-	3	-	-
P_4	-	-	-	-	4	-
P_5	-	-	2	3	4	5

Алгоритм Флойда-Уоршалла

$D^{(3)} =$

	P_0	P_1	P_2	P_3	P_4	P_5
P_0	0	∞	∞	∞	3	2
P_1	∞	0	∞	∞	∞	4
P_2	∞	7	0	3	12	11
P_3	∞	∞	∞	0	∞	∞
P_4	∞	∞	∞	∞	0	∞
P_5	∞	**11**	4	2	5	0

Алгоритм Флойда-Уоршалла

$$B^{(3)} = \begin{array}{|c|cccccc|}
\hline
 & P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\
\hline
P_0 & 0 & - & - & - & 4 & 5 \\
P_1 & - & 1 & - & - & - & 5 \\
P_2 & - & 1 & 2 & 3 & 4 & 1 \\
P_3 & - & - & - & 3 & - & - \\
P_4 & - & - & - & - & 4 & - \\
P_5 & - & \mathbf{2} & 2 & 3 & 4 & 5 \\
\hline
\end{array}$$

Результат четвёртой и пятой итерации совпадает с результатом третьей.

Алгоритм Флойда-Уоршалла

Шестая, последняя итерация:

$$D^{(6)} = \begin{array}{|c|cccccc|}
\hline
 & P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\
\hline
P_0 & 0 & \mathbf{13} & \mathbf{6} & \mathbf{4} & 3 & 2 \\
P_1 & \infty & 0 & \mathbf{8} & \mathbf{6} & \mathbf{9} & 4 \\
P_2 & \infty & 7 & 0 & 3 & 12 & 11 \\
P_3 & \infty & \infty & \infty & 0 & \infty & \infty \\
P_4 & \infty & \infty & \infty & \infty & 0 & \infty \\
P_5 & \infty & 11 & 4 & 2 & 5 & 0 \\
\hline
\end{array}$$

Алгоритм Флойда-Уоршалла

$B^{(6)} =$

	P_0	P_1	P_2	P_3	P_4	P_5
P_0	0	**5**	**5**	**5**	4	5
P_1	-	1	**5**	**5**	**5**	5
P_2	-	1	2	3	4	1
P_3	-	-	-	3	-	-
P_4	-	-	-	-	4	-
P_5	-	2	2	3	4	5

Поиск максимального потока.

Поиск максимального потока

- Предположим, что мы сидим за рулём автомобиля.
- Дорожная сеть — граф.
- Алгоритм Дейкстры определит, за какое время мы доберёмся до любого пункта назначения.
- Как определить все ли автомобили могут проехать по данному маршруту, или пропускная способность транспортной сети ограничена?
- Москва 9 мая: все хотят попасть в центр на парад, но часть дорог вообще перекрыта, а часть — имеет ограниченную ширину.
- Как узнать максимальное число автомобилей, которые могут проехать в центр за, скажем, один час.
- Требуется найти **максимальный поток** между стартом и финишем, источником и стоком.

Поиск максимального потока

- Толчок: вторая мировая война, Д. Б. Данциг, отдел статистического управления ВВС США.
- Нужна математическая модель, каким образом можно быстро сконцентрировать войска и войсковую инфраструктуру вблизи критических точек на театре военных действий.
- Более общая задача: определения пропускной способности рёбер транспортного графа поставлена им в 1951 году.
- 1955 год: Лестер Форд и Делберт Фалкерсон разработали алгоритм, решающий именно эту задачу.
- Хорошее решение данной задачи критически важно для современных транспортных графов (Москва: более 100000 узлов).

Поиск максимального потока: термины

- **Ёмкость ребра** — максимальное интенсивность потока, проходящего через ребро.
- **Насыщенное ребро** — ребро, по которому проходит максимальный поток.

Поиск максимального потока: алгоритм

Алгоритм ищет максимальный поток в сети из источника (*source*) в сток (*destination*).

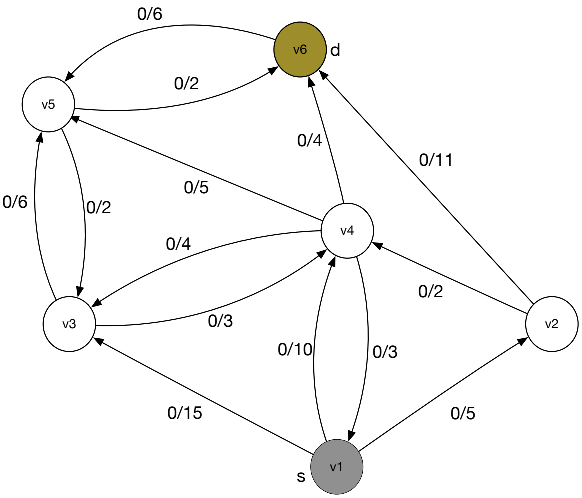
- Каждому ребру ставится в соответствие пара чисел (c, l).
- c — достигнутый до сих пор поток по ребру, вначале он равен нулю, затем это число будет только увеличиваться, пока не достигнет l, ёмкости ребра.
- Алгоритм продолжается, когда на хотя бы одном маршруте из s все рёбра — ненасыщенные, то есть на всех рёбрах $c < l$.

Поиск максимального потока

1. Ищем любой маршрут, содержащий только ненасыщенные рёбра из s в d. Если такого нет, то алгоритм закончен, искомый поток есть сумма потоков всех рёбер, приходящих в d.
2. Мы нашли **дополняющий маршрут**. Определяем значение максимального потока m, который мы можем пропустить по данному маршруту. Он определяется как минимальная из всех возможных разностей ёмкости l и существующего потока c по всем рёбрам маршрута.
3. К каждому из c на маршруте прибавляем m. Хотя бы одно ребро станет насыщенным.
4. Возвращаемся к (1).

Поиск максимального потока

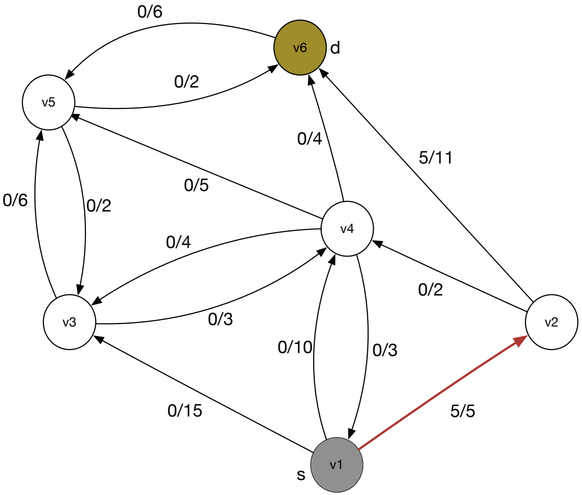
$v1 \to v2 \to v6.$



Поиск максимального потока

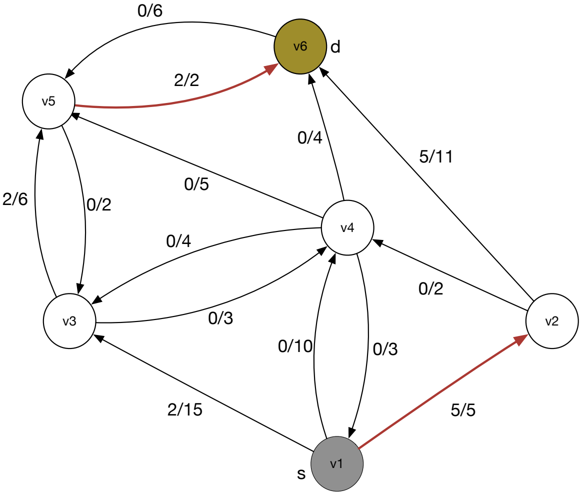
- Найдём произвольный маршрут из s в d. Пусть это будет маршрут $v1 \to v2 \to v6$.
- Наименьшая разница между l и c равна 5, поэтому мы пропускаем поток с интенсивностью 5 по этому маршруту.
- Заметим, что ребро $v1 \to v2$ стало насыщенным и не способно пропускать через себя другие потоки.

Поиск максимального потока



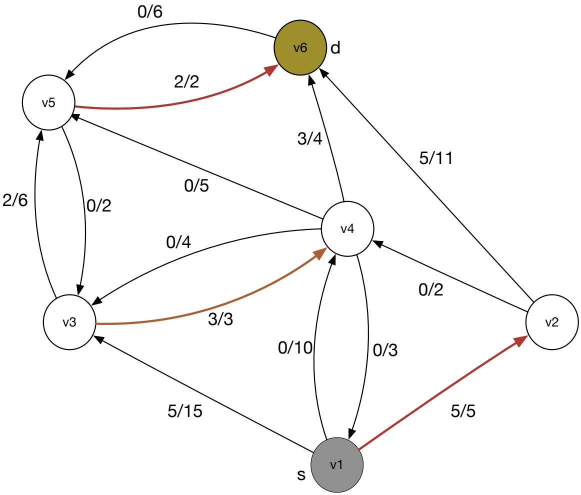
Поиск максимального потока

Найдём ещё один маршрут из $v1$ в $v6$. Через $v2$ все дороги для нас теперь закрыты. Попробуем $v1 \to v3 \to v5 \to v6$. m для этого маршрута равен двум.



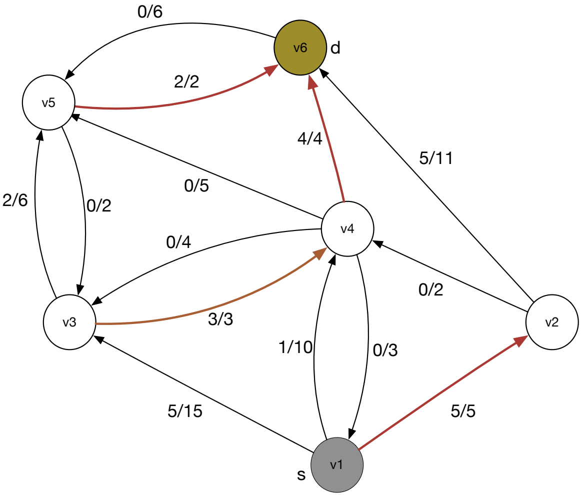
Поиск максимального потока

Следующий этап — маршрут $v1 \to v3 \to v4 \to v6$.



Поиск максимального потока

Следующий этап — маршрут $v1 \to v4 \to v6$. На ребре $v4 \to v6$ имеется резерв для потока, равный 1. Он и даёт нам значение m.

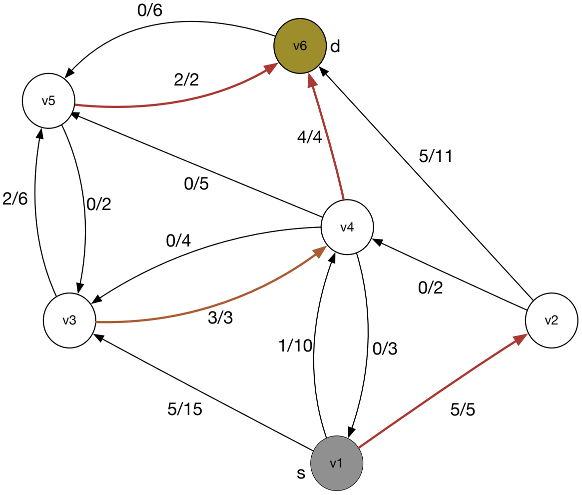


Поиск максимального потока

- Теперь любой маршрут из $v1$ в $v6$ задевает насыщенные рёбра. Р
- Результат равен сумме всех потоков на рёбрах, приходящих в $v6$, то есть, $2 + 4 + 5 = 11$.
- Подсчитаем, сумму потоков всех рёбер, исходящих из $v1$. $5 + 5 + 1 = 11$. Совпадение ли это?

Поиск максимального потока

Где бы вы построили дорогу и какой ёмкости, чтобы увеличить поток от $v1$ к $v6$?



Спасибо за внимание.