

# Введение в алгоритмы.

## Лекция 14 Графы-II.

Сергей Леонидович Бабичев

## План лекции

- Компоненты связности. Алгоритм Косарайю.
- Поиск остовных деревьев в графе.
- Алгоритм Прима.
- Алгоритм Краскала. Система непересекающихся множеств.
- Алгоритм Дейкстры.

# Поиск компонент связности

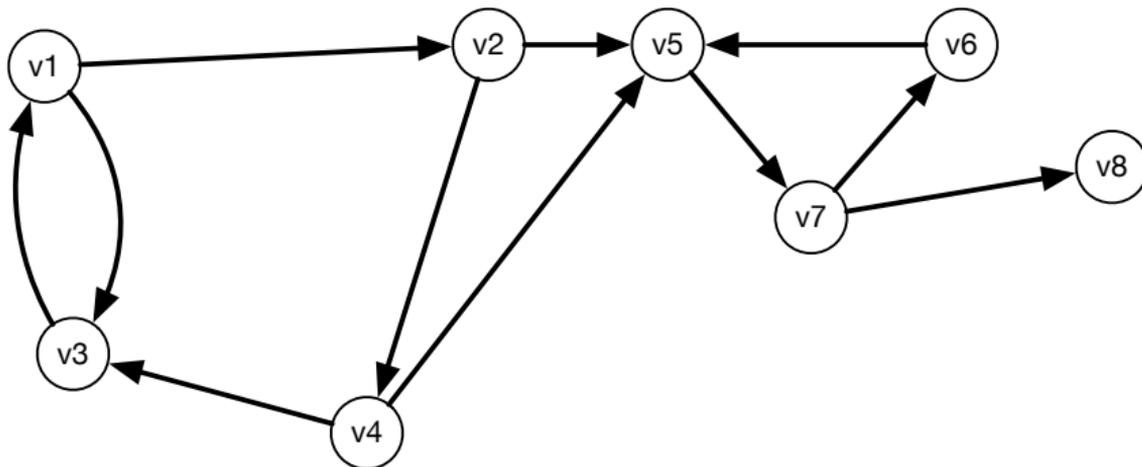
# Поиск компонент связности

- Для неориентированных графов: запусив поиск BFS или DFS.
- Все выкрашенные по завершении поиска вершины образуют компоненту связности.
- Выбирается произвольным образом необработанная вершина и алгоритм повторяется, формируя другую компоненту связности.
- Алгоритм заканчивается, когда не остаётся необработанных вершин.

# Поиск компонент связности

- Для ориентированных графов: результаты зависят от порядка обхода вершин.
- **Компонента сильной связности ориентированного графа:** максимальное по размеру множество вершин, взаимно достижимых друг из друга.

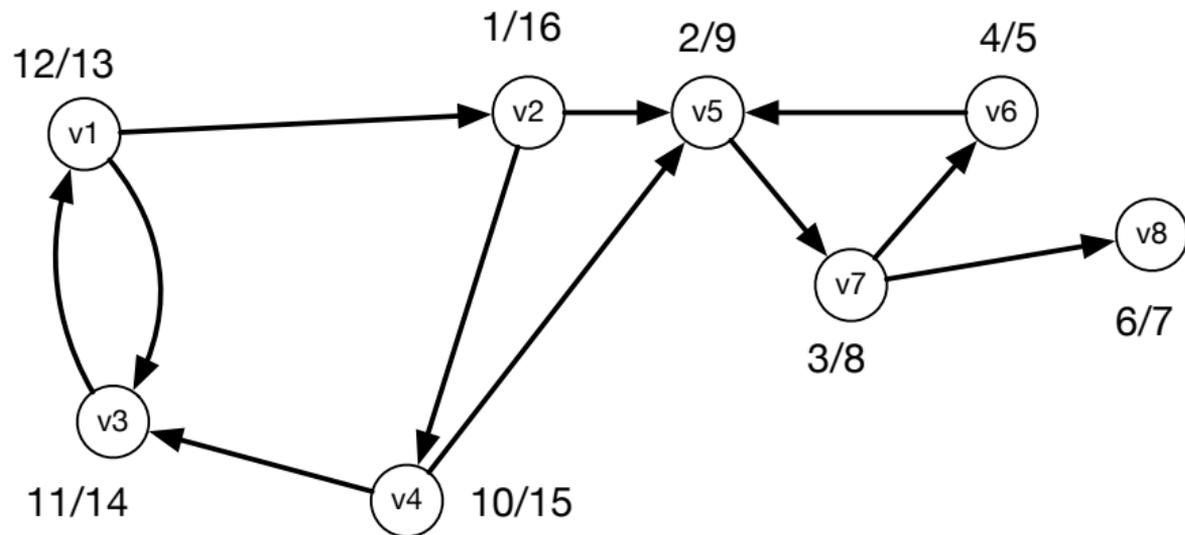
# Поиск компонент связности: алгоритм Косарайю



# Поиск компонент связности: алгоритм Косарайю

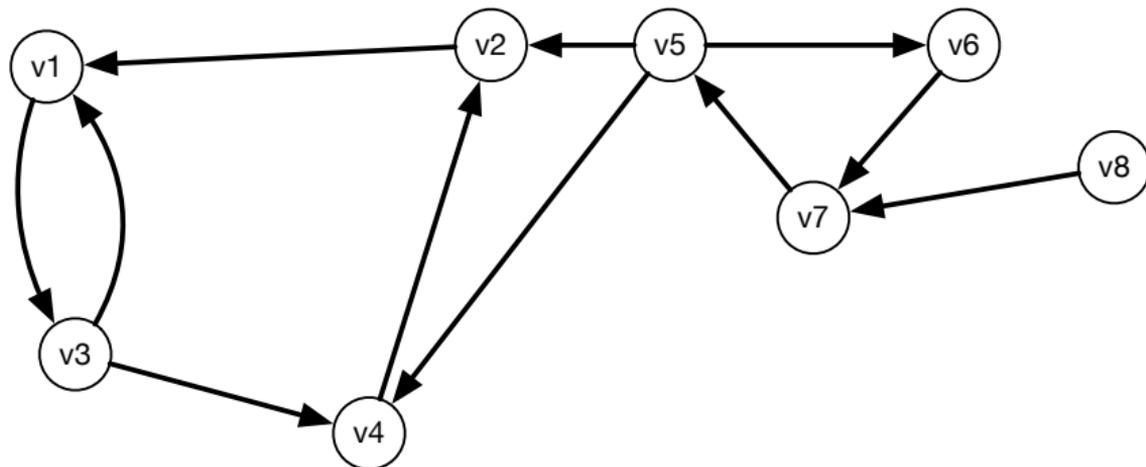
- Проведём полный DFS поиск.
- В алгоритме полного DFS не специфицировано, с какой вершины начинается поиск → можно выбрать произвольную.

Обход с вершины  $v_2$ :



# Поиск компонент связности: алгоритм Косарайю

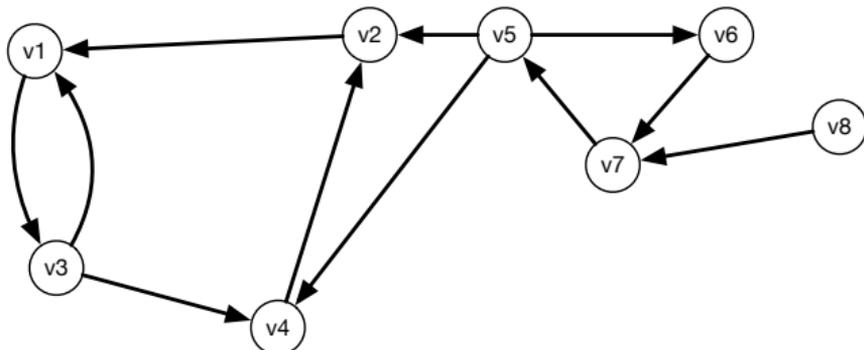
- Заменяем направления всех рёбер (перевернём все стрелки).
- Каждое ребро  $u \rightarrow v$  заменяется на  $v \rightarrow u$ .



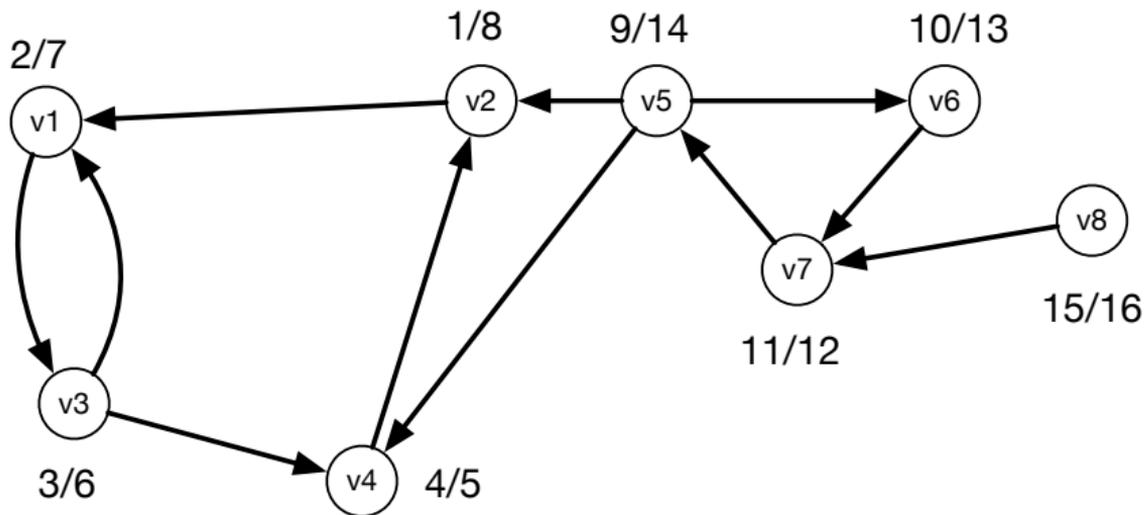
# Поиск компонент связности: алг. Косарайю

- Обходим ещё раз. Начальная вершина — из необработанных, у которой наибольшее значение времени выхода.
- Обход из вершины 2 покрасил вершины  $v_1, v_2, v_3$  и  $v_4$ .
- Остались непокрашенные вершины  $v_5, v_6, v_7$  и  $v_8$ .
- Повторяем, пока останутся непокрашенные вершины.

Номер	1	2	3	4	5	6	7	8
Вход/выход	12/13	1/16	11/14	10/15	2/9	4/5	3/8	6/7

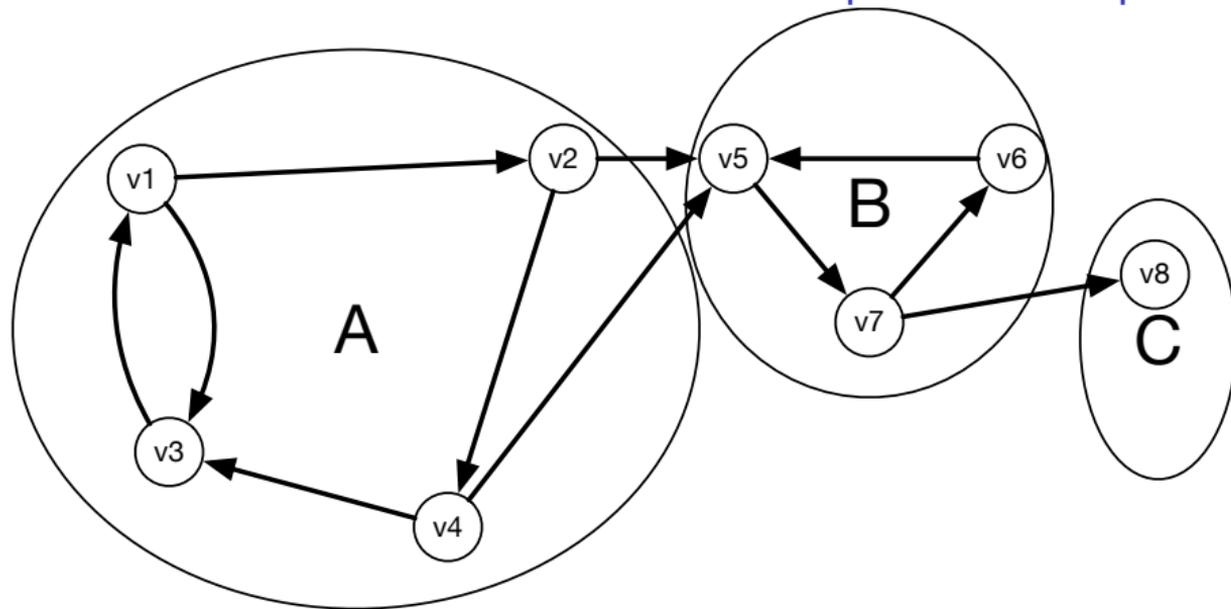


# Поиск компонент связности: алгоритм Косарайю



- Каждый «малый» проход алгоритма DFS даст нам вершины, которые принадлежат одной компоненте сильной связности.

## Поиск компонент связности: алгоритм Косарайю



- Рассматривая компоненту сильной связности как единую мета-вершину, мы получаем новый граф, который называется *конденсацией* исходного графа или *конденсированным графом*.

# Остовные деревья

## Остовное дерево: ещё немного терминов

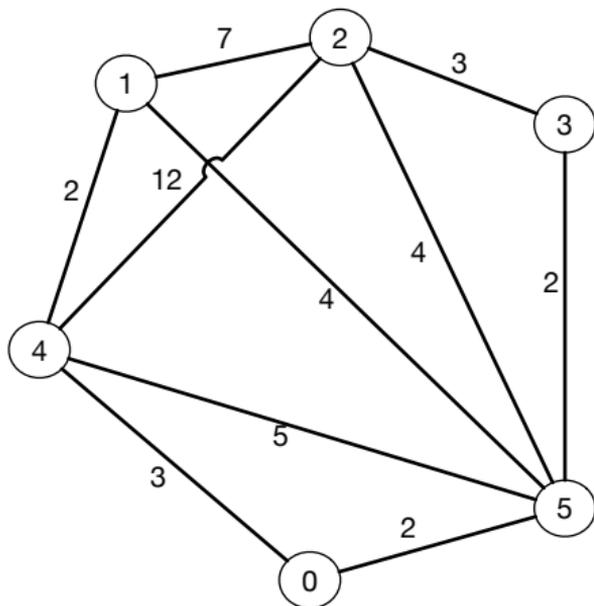
- С точки зрения теории графов **дерево** есть ациклический связный граф.
- Множество деревьев называется **лесом (forest)** или **бором**.
- **Остовное дерево** связного графа — подграф, который содержит все вершины графа и представляет собой полное дерево.
- **Остовный лес** графа — лес, содержащий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

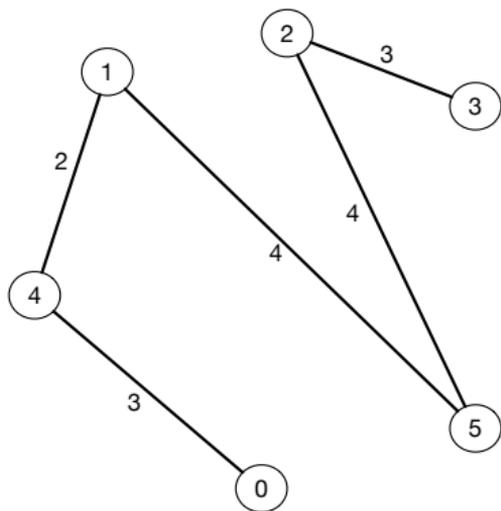
Каждый из узлов имеет информацию о связях с соседями (рёбрах). Каждое ребро имеет вес.



# 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.

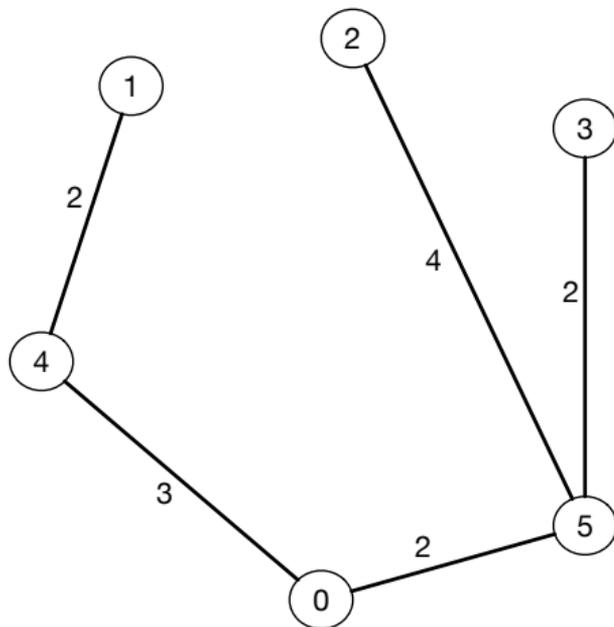
Это — остовное дерево для корневого узла  $P_r$ .



# 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.
- **Перекрытое ребро** — ребро, соединяющее вершину одного множества с вершиной другого множества.

- **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  $e$ .
- **Лемма.** При любом сечении графа каждое минимальное перекрёстное ребро принадлежит некоторому MST-дереву и каждое MST-дерево содержит перекрёстное ребро.

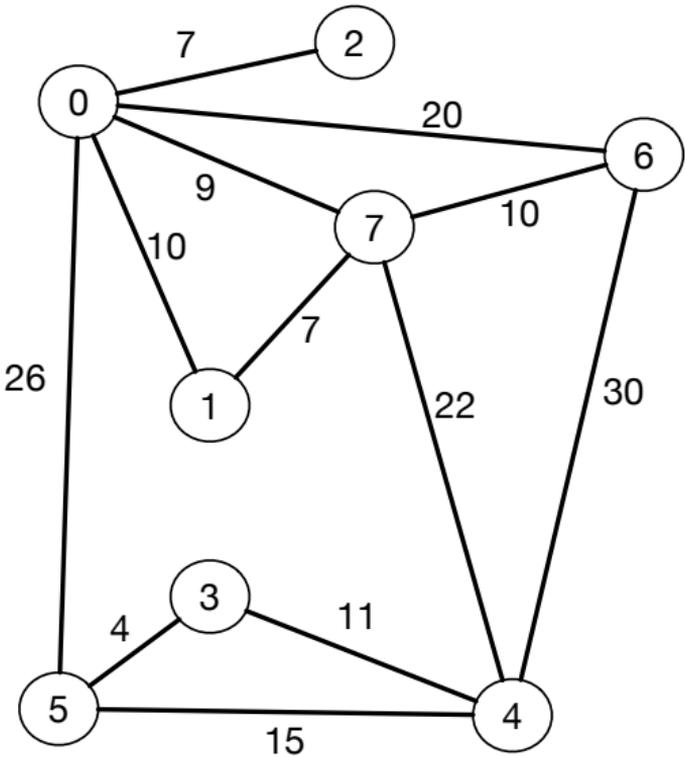
# Алгоритмы поиска MST

## Алгоритм Прима

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

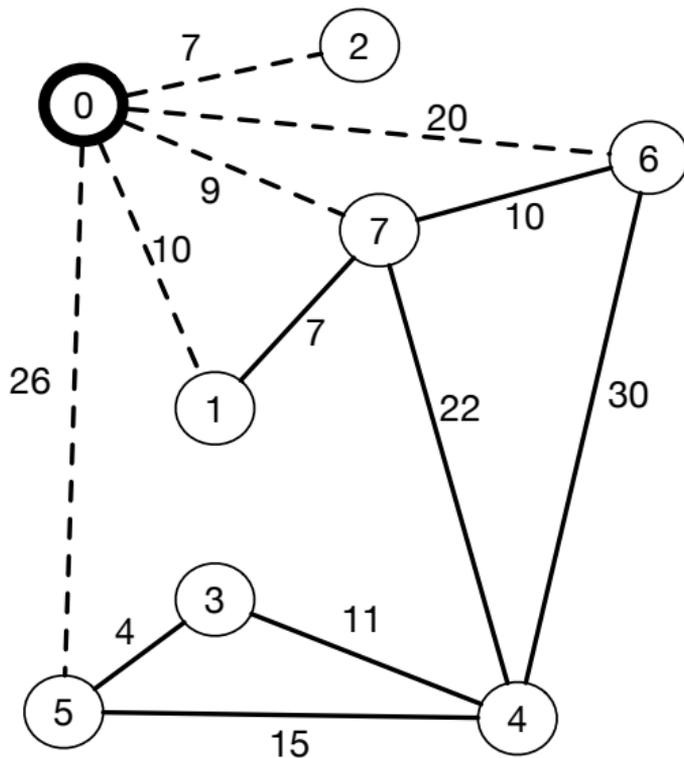
# Алгоритм Прима

Исходный граф.



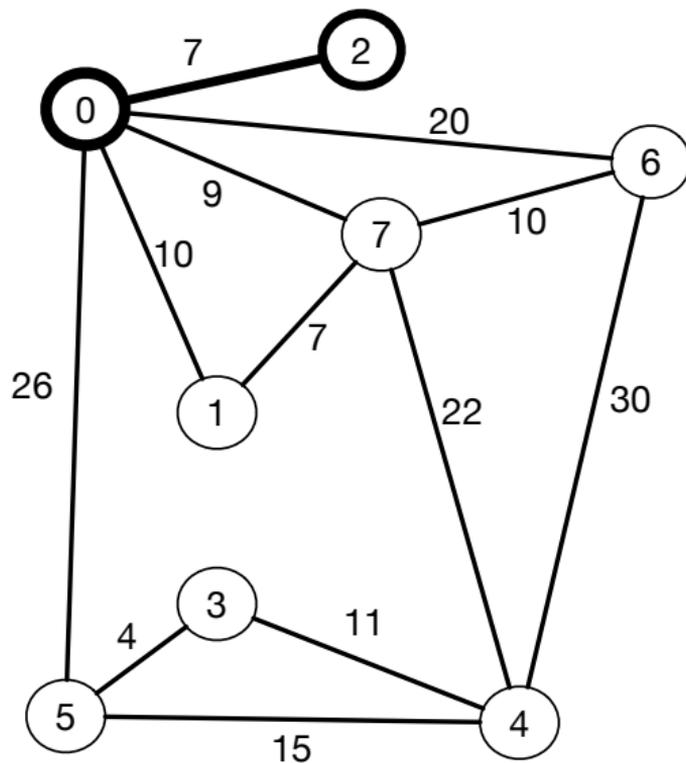
# Алгоритм Прима

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



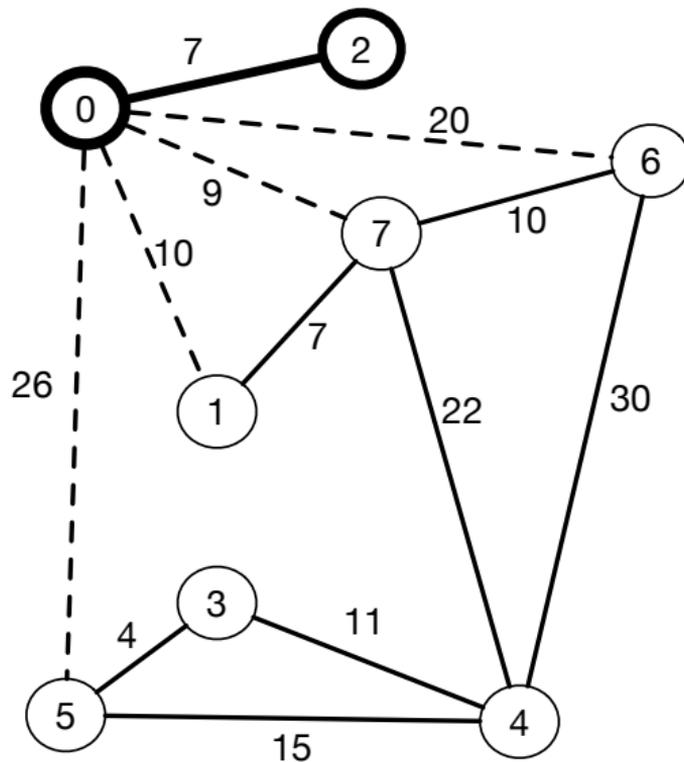
# Алгоритм Прима

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



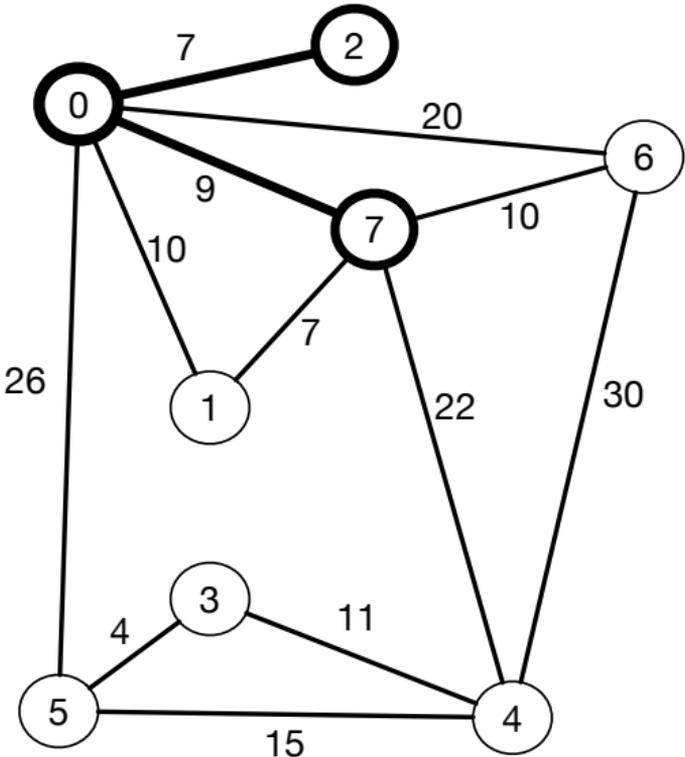
# Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



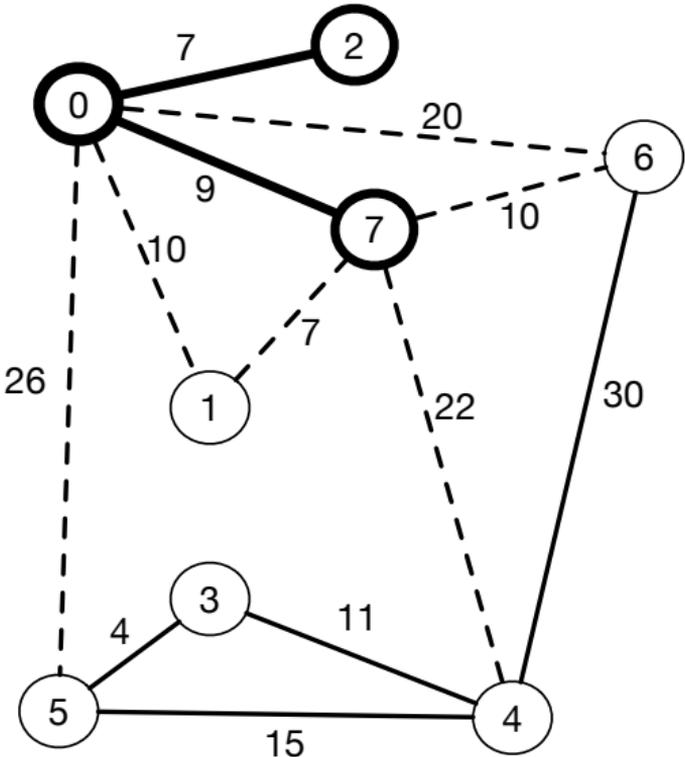
# Алгоритм Прима

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



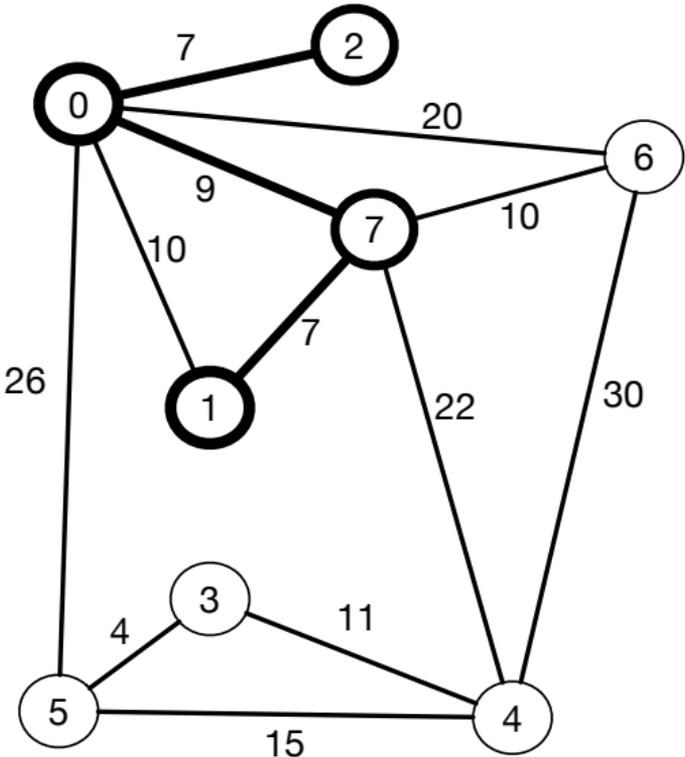
# Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



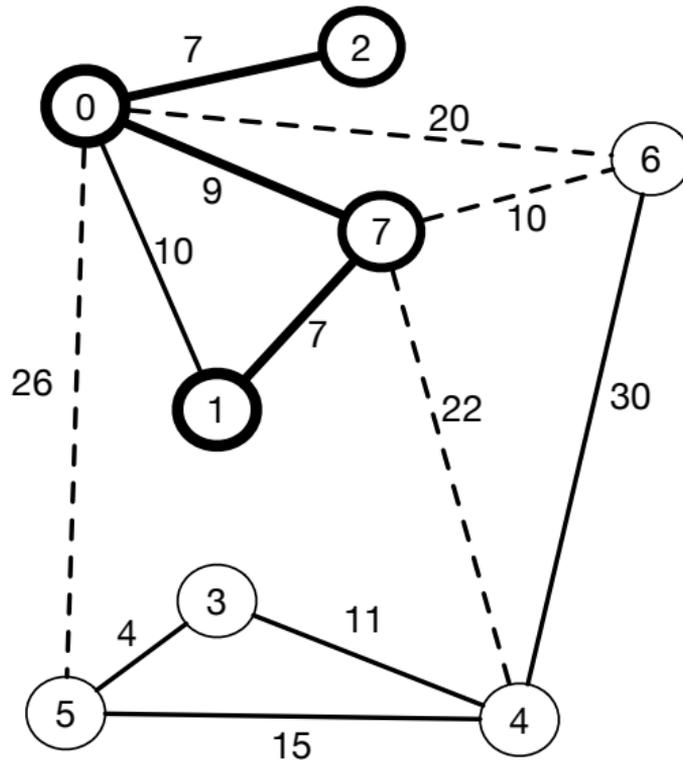
# Алгоритм Прима

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



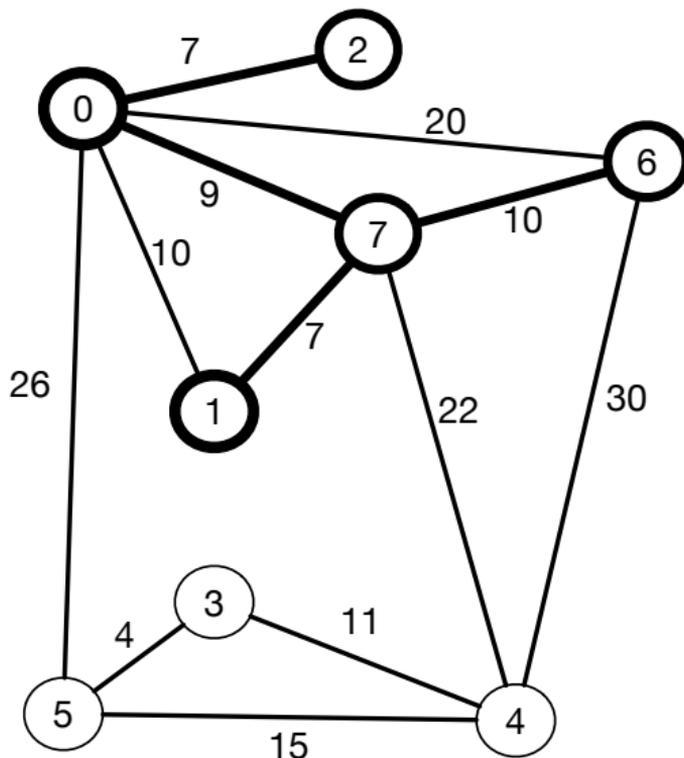
# Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



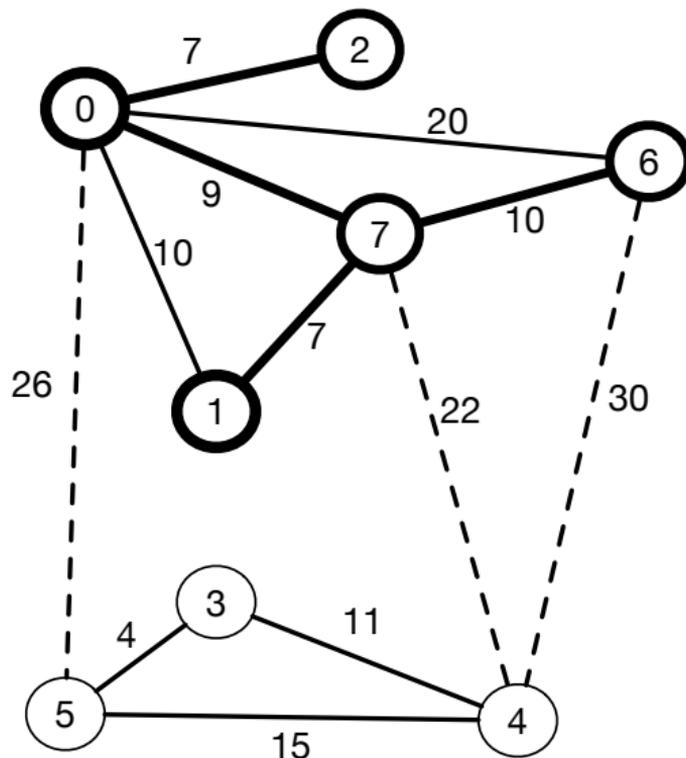
# Алгоритм Прима

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



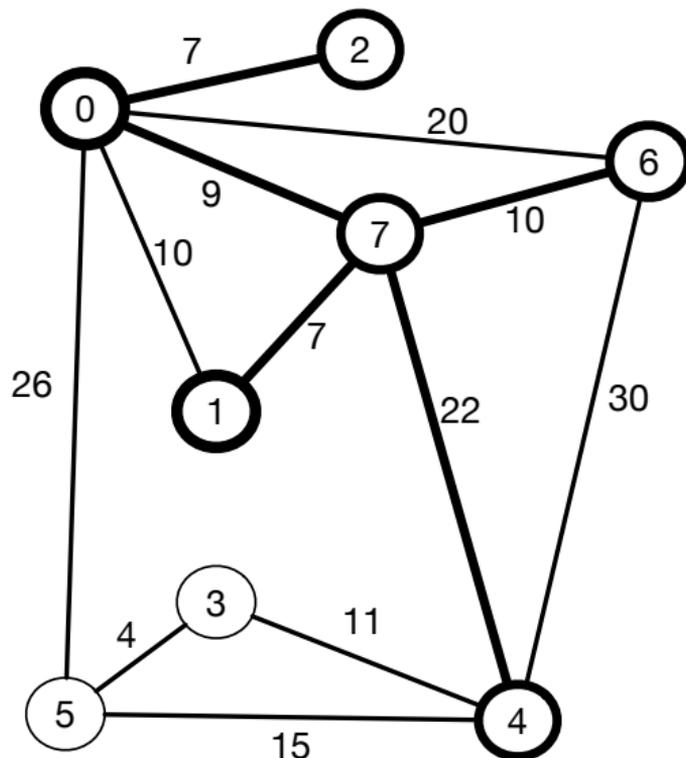
# Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



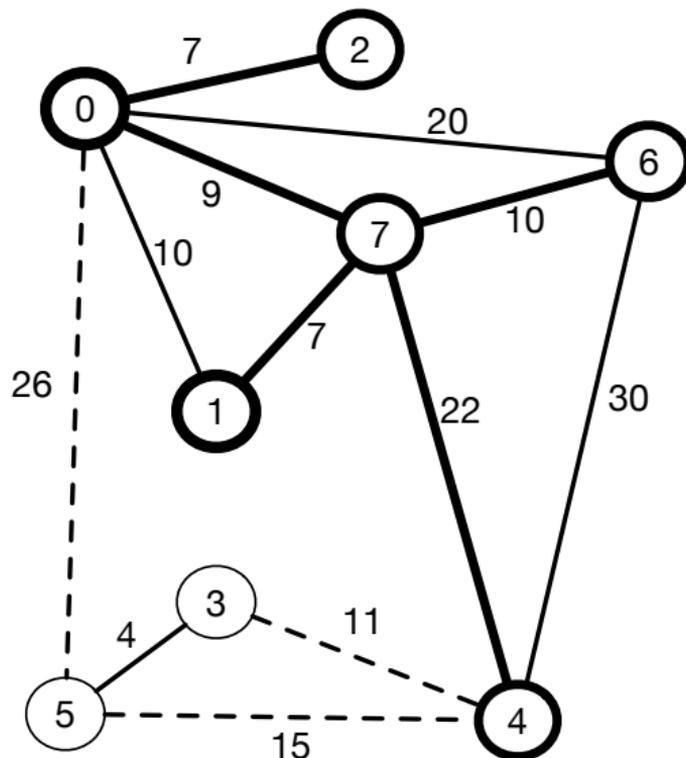
# Алгоритм Прима

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



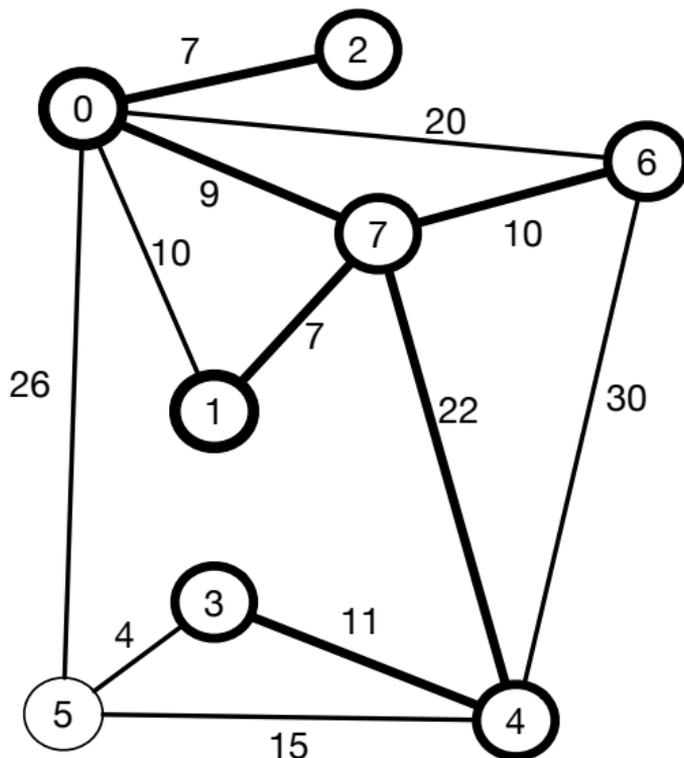
# Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



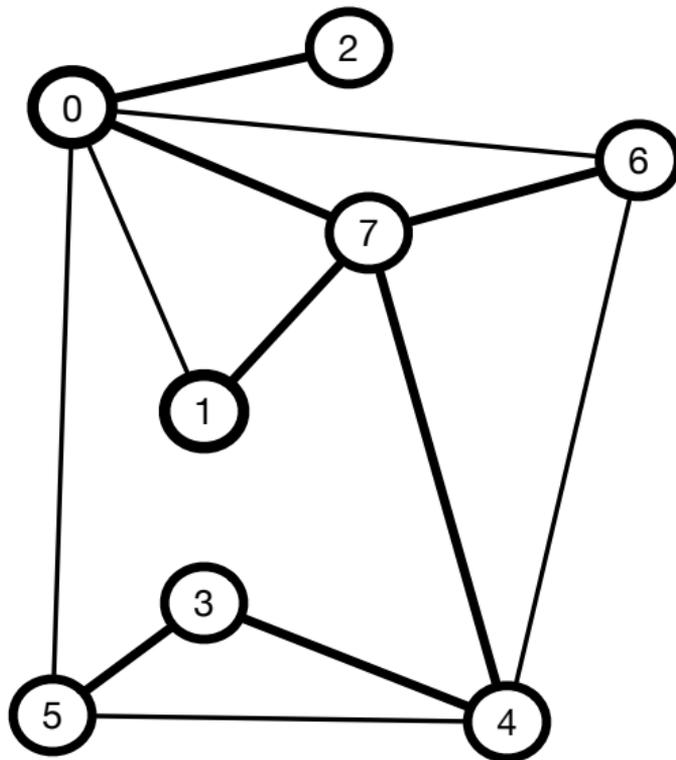
# Алгоритм Прима

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, которые уже проверяли.
- Введём понятие *накопителя*.
- Накопитель содержит множество рёбер-кандидатов.
- Каждый раз в MST включается самое лёгкое ребро.

# 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

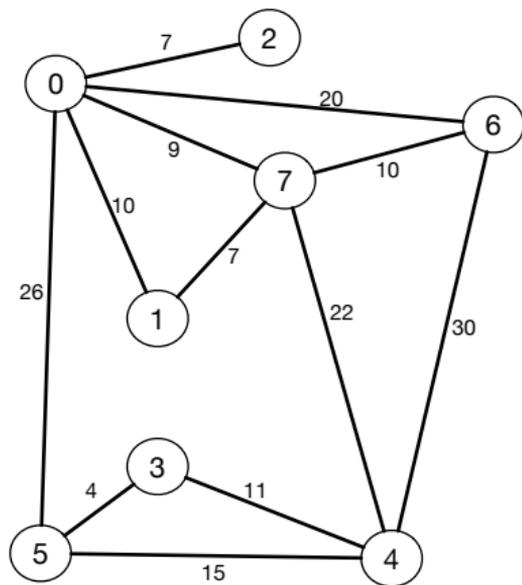
- Алгоритм Прима — обобщение поиска на графе.
- Накопитель представляется очередью с приоритетами.
- Используется операция «извлечь минимальное».
- Используется операция «увеличить приоритет».
- Такой поиск на графе называется PFS — поиск по приоритету.
- Сложность алгоритма  $O(|E| \log |V|)$ .

# 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

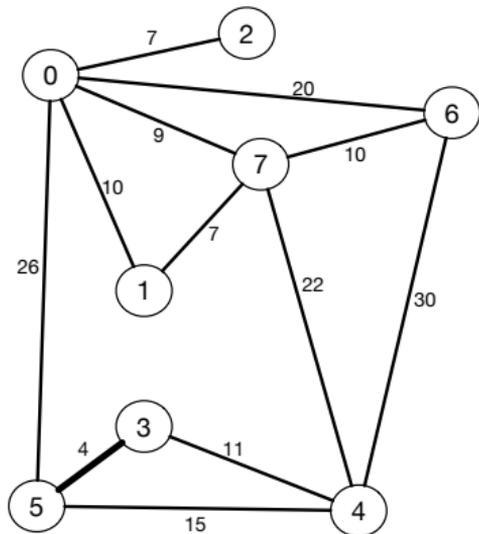
# Алгоритм Краскала

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 → 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

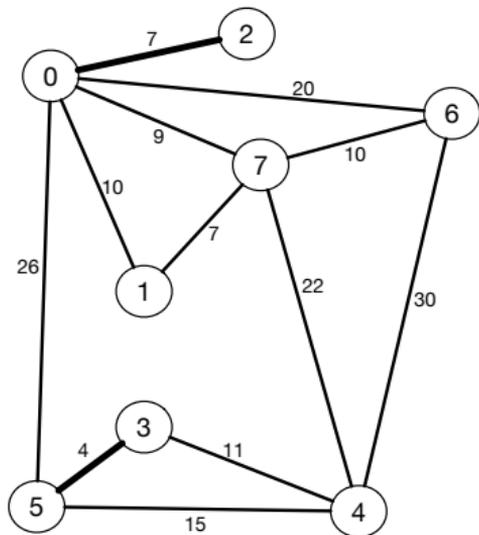
**Лемма.** При равных подходящих рёбрах можно выбирать произвольное.

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

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

# Алгоритм Краскала: вторая итерация

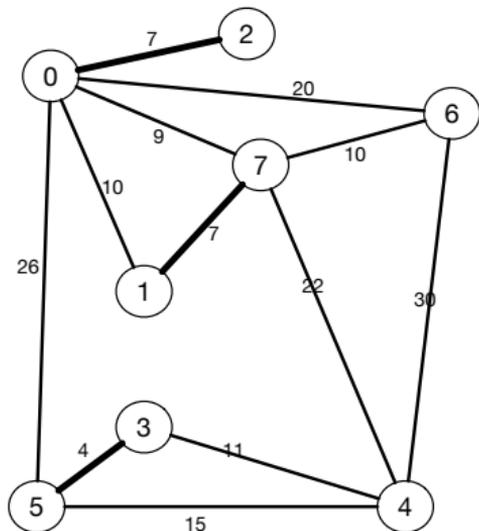
Выберем ребро (0,2) и поместим вершину 2 в множество номер 0.



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

# Алгоритм Краскала: третья итерация

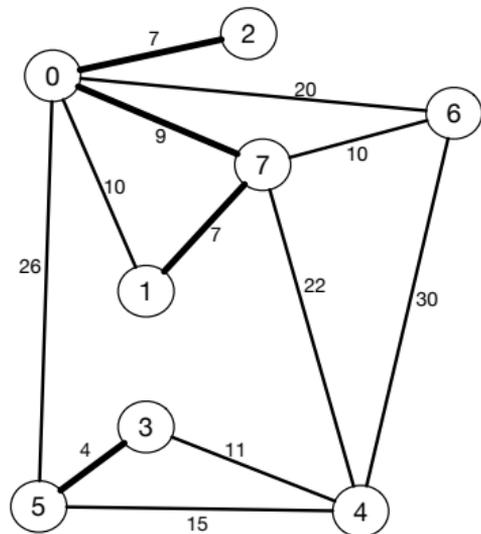
Ребро (1,7) привело к слиянию множеств 1 и 7.



$V_i$	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: `find_root(x)`

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

- $p[7]==1$  — 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.

# Система непересекающихся множеств

- 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; // A trivial case  
    return p[r] = find_root(p[r]); A recursive case  
}
```

# 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() % 2) p[l] = r;  
    else p[r] = l;  
}
```

- Важно: операция слияния начинается с операции 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

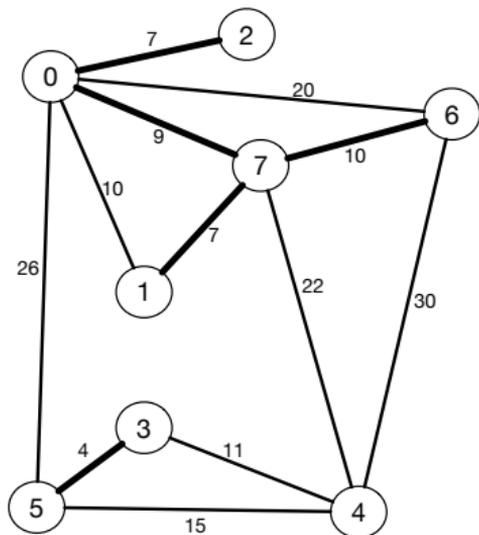
- Определяем, каким деревьям принадлежат концы ребра  $(0,7)$ .
- $\text{find\_root}(0)$  вернёт 0 как номер множества.
- $\text{find\_root}(7)$  сначала убедится, что в  $p[7]$  лежит 1 и вызовет  $\text{find\_root}(1)$ , после чего, возможно, заменит  $p[7]$  на 1 и вернёт 1.

# Алгоритм Краскала

- Концы ребра 0 принадлежат разным множествам  $\rightarrow$  сливаем множества, вызвав  $\text{merge}(0, 7)$ .
- Операция  $\text{merge}$  — 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

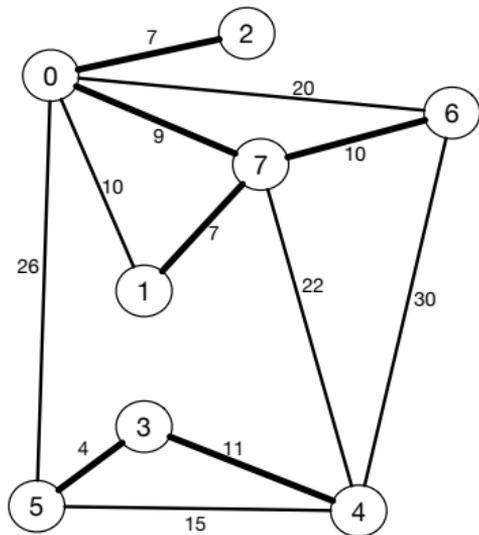


Следующее ребро —  $(6,7)$ .  $\text{find\_root}(7)$  установит  $p[7]=0$ . Это же значение будет присвоено и  $p[6]$ .

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

# Алгоритм Краскала

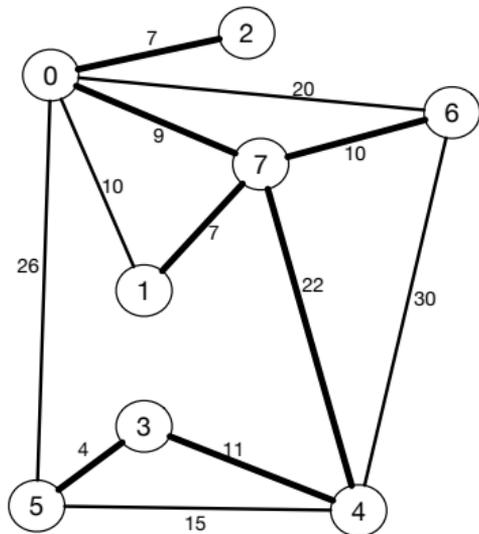
На следующем этапе ребро (3,4) окажется самым коротким.



$V_i$	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. Рёбра (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 амортизированную  $O(1)$ .
- 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 поддереве 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.
- **Обязательное условие:** граф не должен содержать рёбер 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

```
1: procedure DIJKSTRA( $G : Graph; w : weights; s : Vertex$ )
2:   for all  $v \in V$  do
3:      $d[v] \leftarrow \infty$ 
4:      $\pi[v] \leftarrow nil$ 
5:   end for
6:    $d[s] \leftarrow 0$ 
7:    $U \leftarrow \emptyset$ 
8:    $Q \leftarrow V$ 
9:   while  $Q \neq \emptyset$  do
10:     $u \leftarrow Q.extractMin()$ 
11:     $U \leftarrow U \cup \{u\}$ 
12:    for all  $v \in Adj[u], v \notin U$  do
13:      Relax( $u, v$ )
14:    end for
15:  end while
16: end procedure
```

# Алгоритм Дейкстры

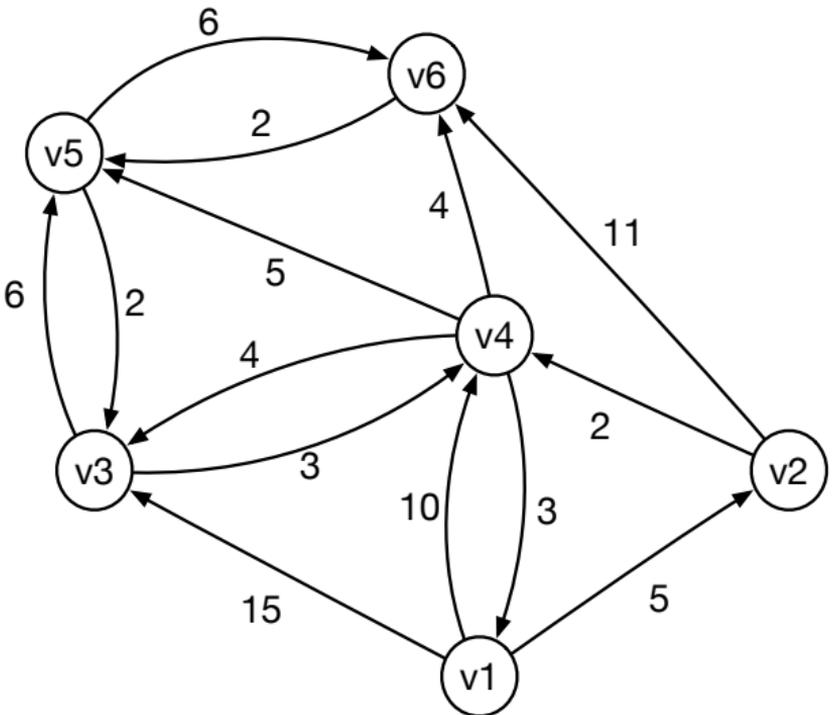
```
1: procedure RELAX( $u, v : Vertex$ )  
2:   if  $d[v] > d[u] + w(u, v)$  then  
3:      $d[v] = d[u] + w(u, v)$   
4:      $\pi[v] \leftarrow u$   
5:   end if  
6: end procedure
```

# 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.

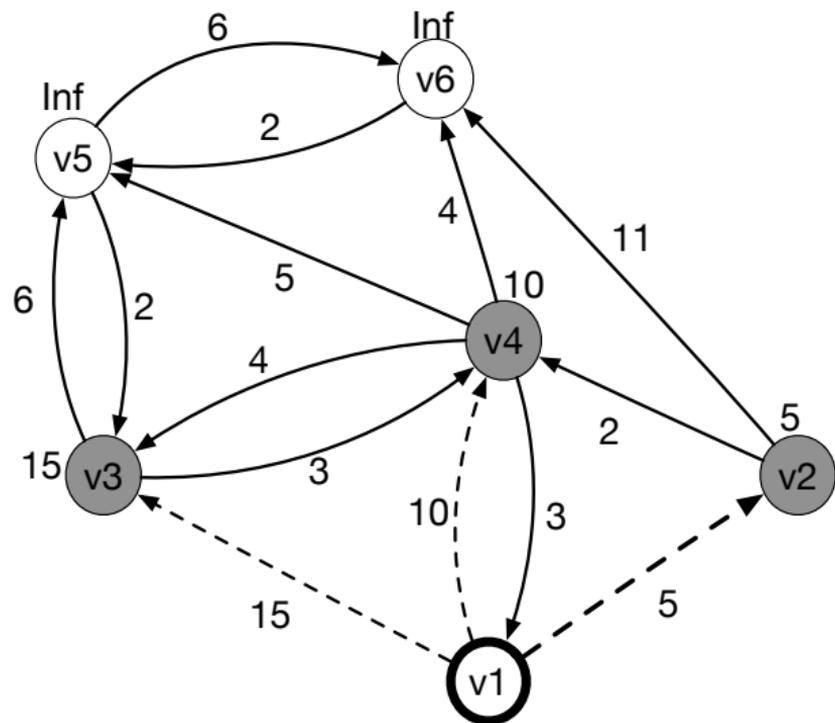
# Алгоритм Дейкстры

Исходный граф



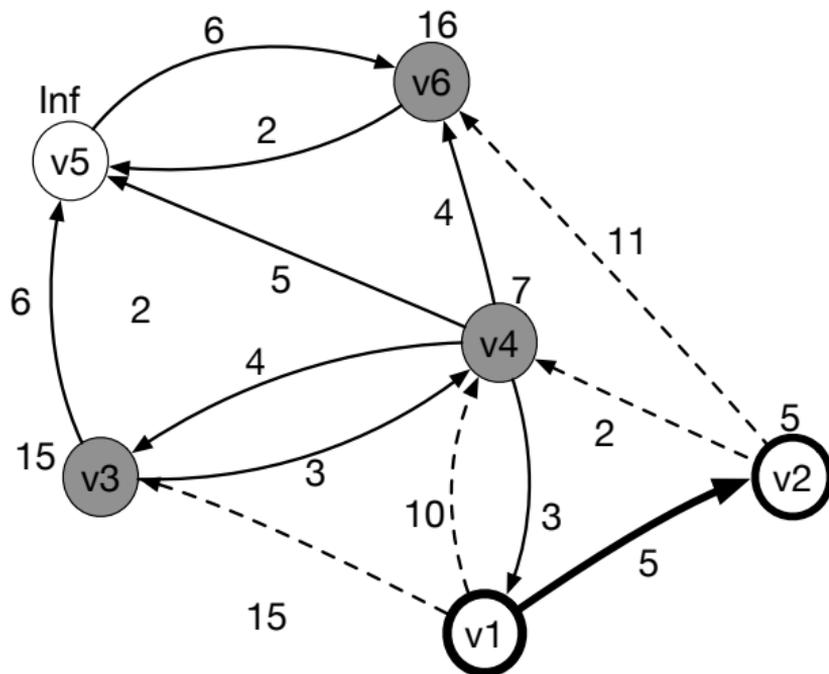
# Алгоритм Дейкстры

$v_1$  в SPT,  $v_2$ ,  $v_3$  и  $v_4$  — в накопителе.



# Алгоритм Дейкстры

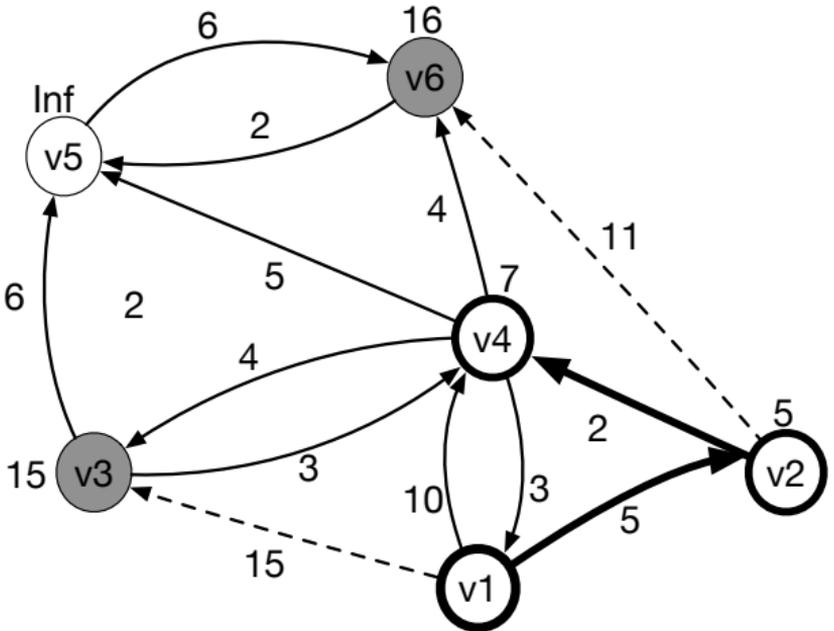
Выбран узел  $v_2$ . Корректируются расстояния от него. Релаксация:  $(1 \rightarrow 4)$



заменён на  $(1 \rightarrow 2 \rightarrow 4)$ .

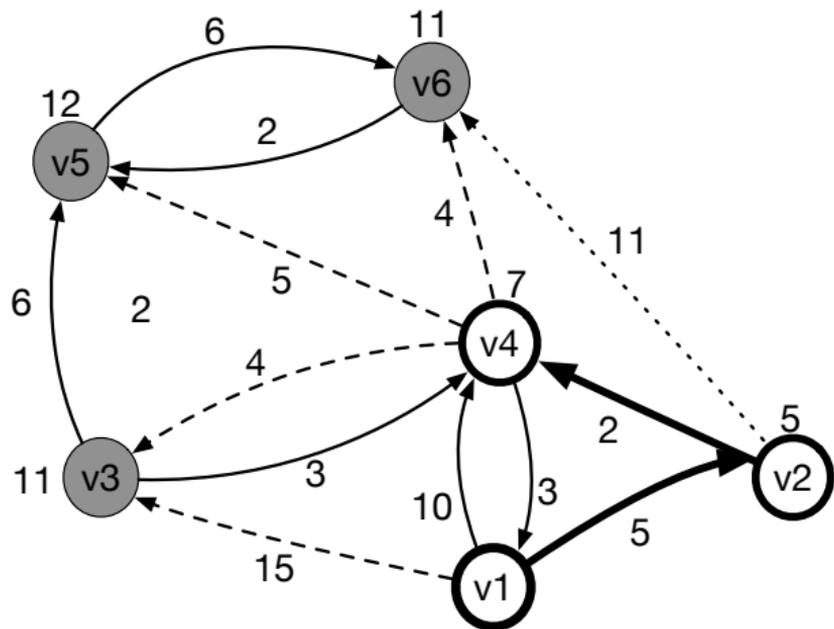
# Алгоритм Дейкстры

Выбран узел v3.

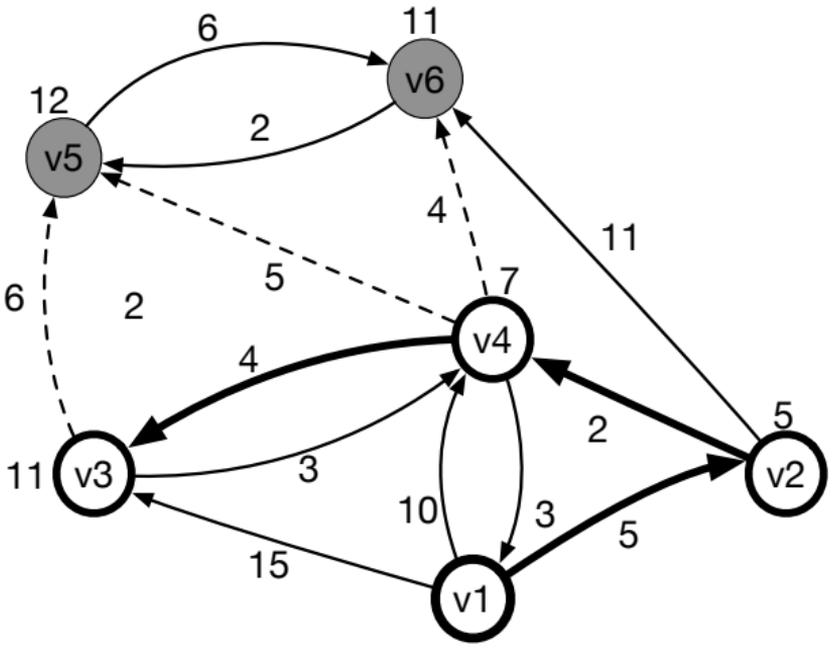


# Алгоритм Дейкстры

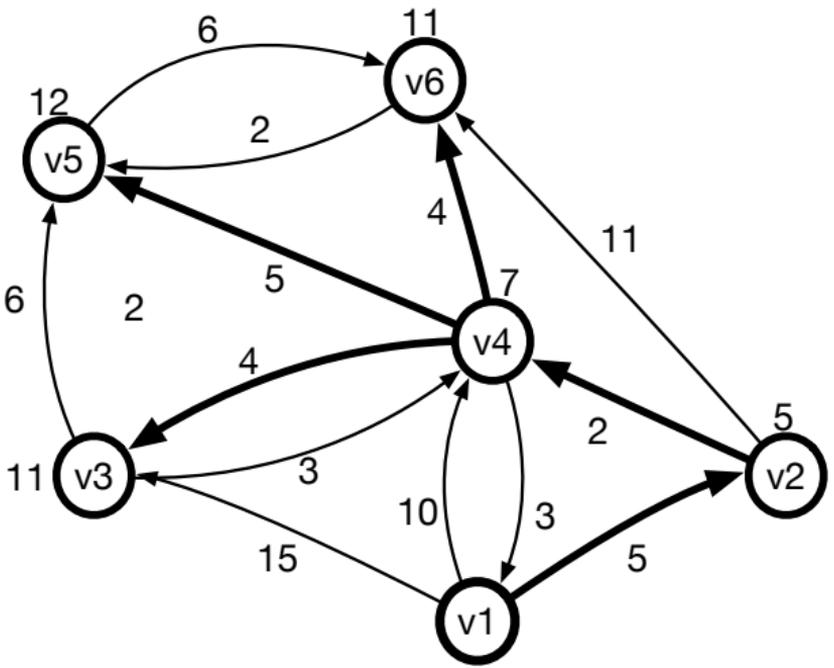
В накопитель отправляется v5. Релаксация:  $(1 \rightarrow 2 \rightarrow 6)$  заменено на  $(1 \rightarrow 2 \rightarrow 4 \rightarrow 6)$ ,  $(1 \rightarrow 3)$  на  $(1 \rightarrow 2 \rightarrow 4 \rightarrow 3)$



# Алгоритм Дейкстры



# Алгоритм Дейкстры



# Алгоритм Дейкстры: сложность

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

Спасибо за внимание.