

# Алгоритмы и структуры данных

Лекция 12

Графы - 2

Кандауров Геннадий



# Напоминание отметиться на портале

+ ОСТАВИТЬ ОТЗЫВ

The screenshot shows the VK Education website interface. At the top, there's a navigation bar with links like 'Блоги' (Blogs), 'Люди' (People), 'Программа' (Program), 'Вакансии' (Jobs), and 'Расписание' (Schedule). A yellow banner at the top right says 'Открыт приём заявок!' (Applications are open!). Below the banner, a weekly schedule is displayed:

Чт, 8 сентября	пт, 9 сентября	сб, 10 сентября	вс, 11 сентября	пон, 12 сентября
Нет занятий	18:00 Углублённый Py... Введение в Python, основные понятия, тестирование Г. Кандауров	Нет занятий	Нет занятий	Нет занятий

Below the schedule, there's a blog post titled 'Углубленный Python' (Advanced Python) with the subtitle 'Blog for materials on the "Advanced Python" course'. It has 57 readers and 2 topics. A search bar and a 'Найти' (Find) button are below the post. To the right, there's a sidebar for 'Прямой эфир' (Live Stream) and a list of recent comments from users like Геннадий Кандауров, Екатерина Черкасова, and Дарья Вовченко.

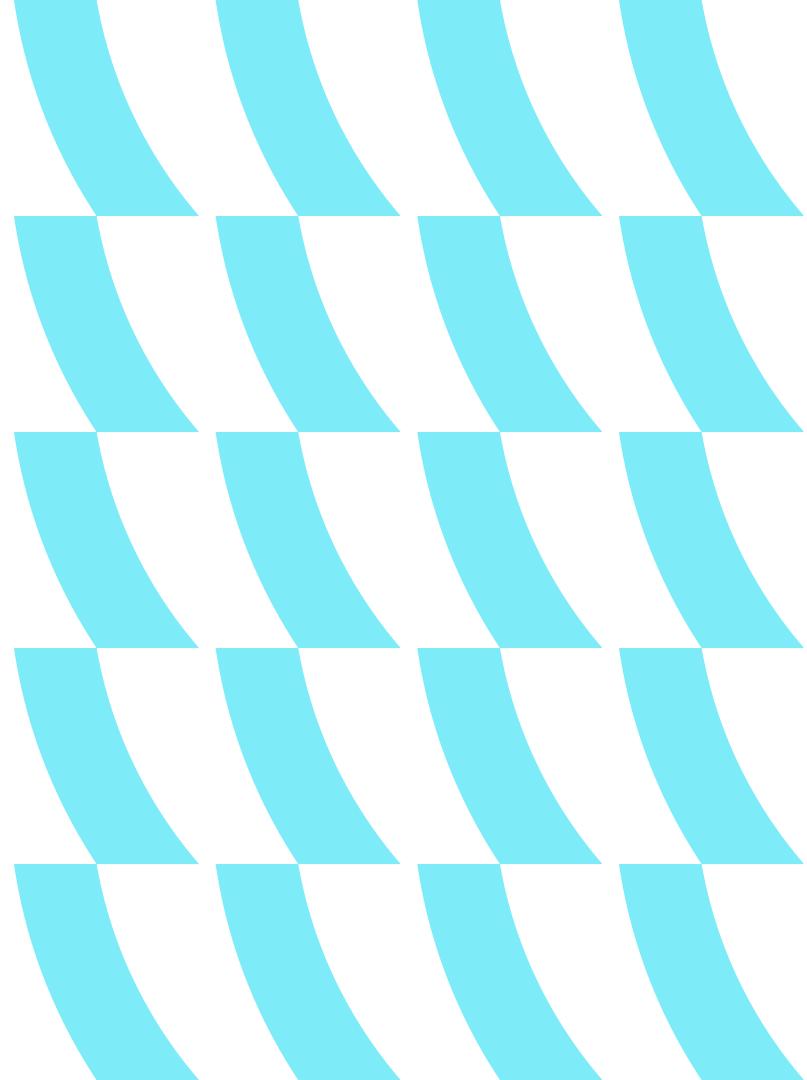
Квиз про прошлой лекции



## Содержание занятия

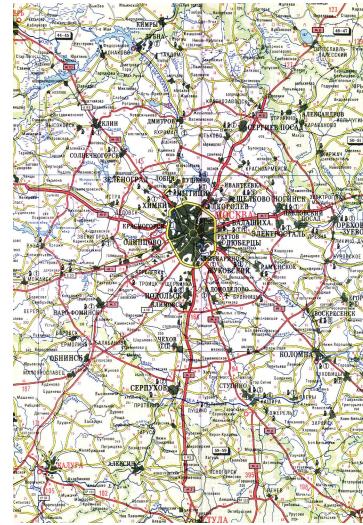
1. Алгоритм Дейкстры, A\*
2. Остовные деревья, алгоритм Прима
3. Задача коммивояжера

# Поиск расстояний



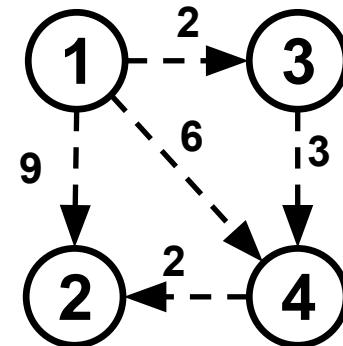
# Взвешенный граф

Во **взвешенном графе** с каждым ребром связано некоторое число (вес), которое обычно интерпретируется как расстояние либо стоимость.



В **евклидовом** графе узлы соответствуют точкам на плоскости, а вес рёбер равен евклидову расстоянию между вершинами.

Пример: карта дорог.

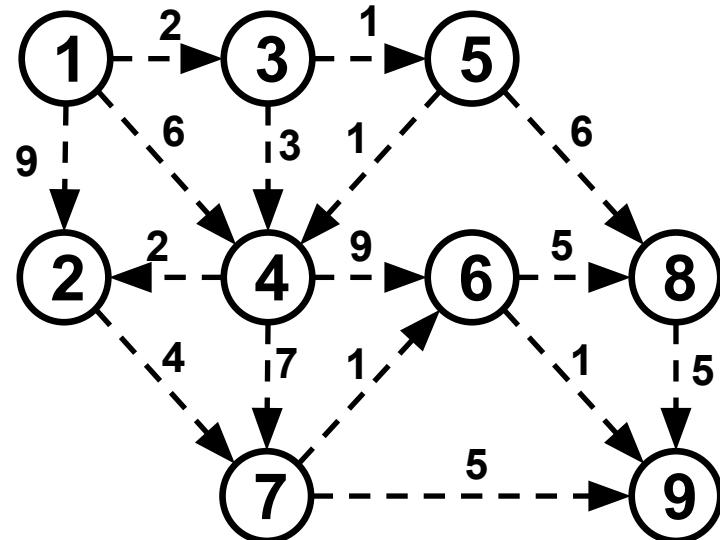


# Взвешенный граф: поиск кратчайших путей

$G$  – ориентированный взвешенный граф

**Кратчайшим путём** между двумя вершинами  $u$  и  $v$  в графе  $G$  называется такой направленный простой путь из  $u$  в  $v$ , что никакой другой путь не имеет меньшего веса.

- Веса рёбер не обязательно пропорциональны их длинам.
- Вес – это количество рёбер.
- Может существовать несколько кратчайших путей.
- А может не существовать ни одного пути.

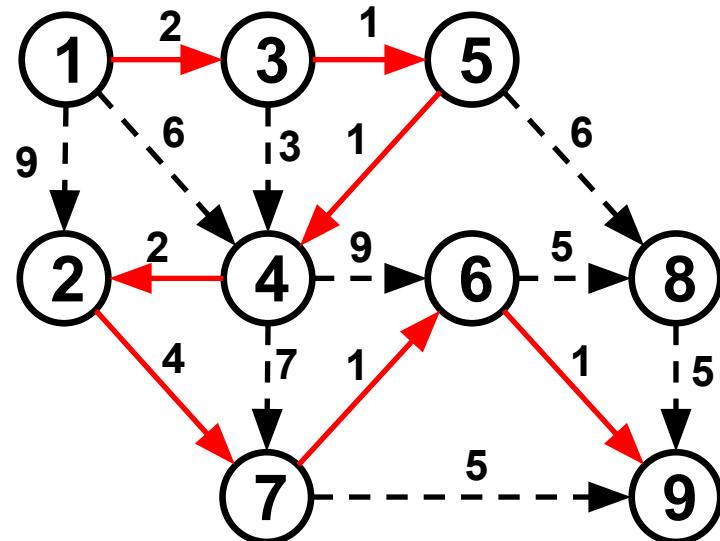


# Взвешенный граф: поиск кратчайших путей

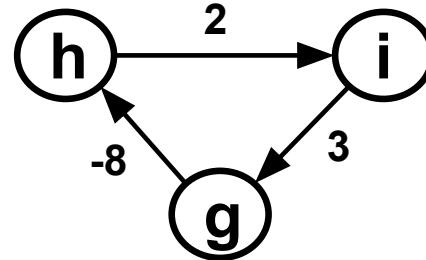
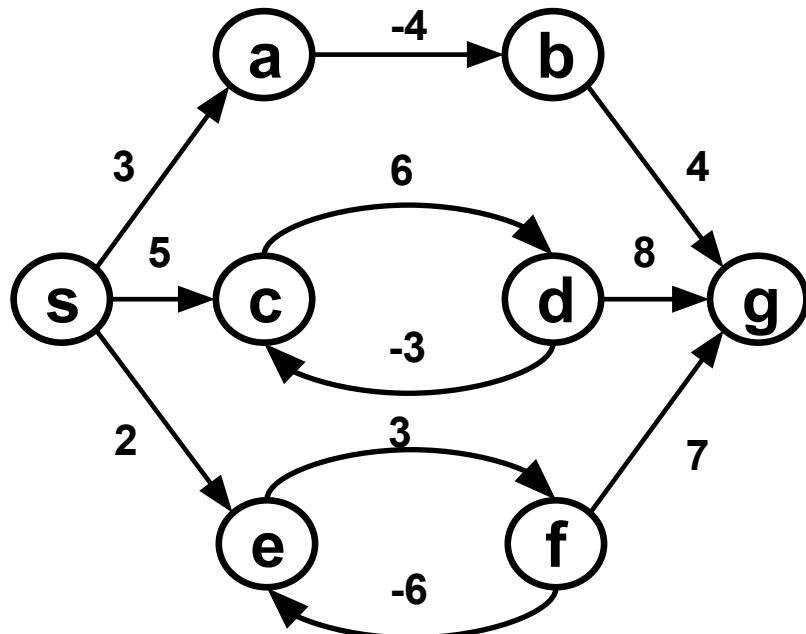
$G$  – ориентированный взвешенный граф

**Кратчайшим путём** между двумя вершинами  $u$  и  $v$  в графе  $G$  называется такой направленный простой путь из  $u$  в  $v$ , что никакой другой путь не имеет меньшего веса.

- Веса рёбер не обязательно пропорциональны их длинам.
- Вес – это количество рёбер.
- Может существовать несколько кратчайших путей.
- А может не существовать ни одного пути.



## Ребра с отрицательным весом



Рёбра с отрицательным весом допустимы –  
 $(s, a, b)$  – кратчайший путь из  $s$  в  $b$ .

Циклы с положительным весом допустимы  
–  $w(c, d) + w(d, c) = 3 > 0$ . Путь  $(s, c, d)$  –  
кратчайший из  $s$  в  $d$ .

При наличии на пути цикла с отрицательным весом –  $w(e, f) + w(f, e) = 3 - 6 = -3 < 0$   
кратчайшего пути между вершинами  $s$  и  $f$  не существует.

# Задачи поиска кратчайших путей

$G$  – ориентированный взвешенный граф.

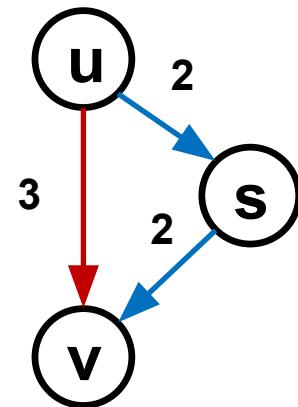
1. SPSP (Single Pair Shortest Path problem) – поиск кратчайшего пути между двумя вершинами.  
*Алгоритмы Дейкстры, A\*, IdaStar.*
2. SSSP (Single Source Shortest Paths problem) – поиск кратчайших путей из выделенной вершины до всех остальных.  
*Алгоритмы Дейкстры, Беллмана-Форда.*
3. APSP (All Pairs Shortest Paths problem) – поиск кратчайших путей между всеми парами вершин.  
*Алгоритмы Флойда, Джонсона.*

# Релаксация

**Релаксация ребра** – это проверка того, даёт ли продвижение по данному ребру новый кратчайший путь к конечной вершине.

*Процедура релаксации ребра*

```
bool Relax( u, v ) {  
    if( d[v] > d[u] + w( u, v ) ) {  
        d[v] = d[u] + w( u, v );  
        pi[v] = u;  
        return true;  
    }  
    return false;  
}
```



$$d[v] = 4$$

При релаксации  $(u, v)$   $d[v]$  получит новое значение 3.

# Алгоритм Дейкстры

Алгоритм Дейкстры похож на BFS, но вместо обычной очереди используется очередь с приоритетом.

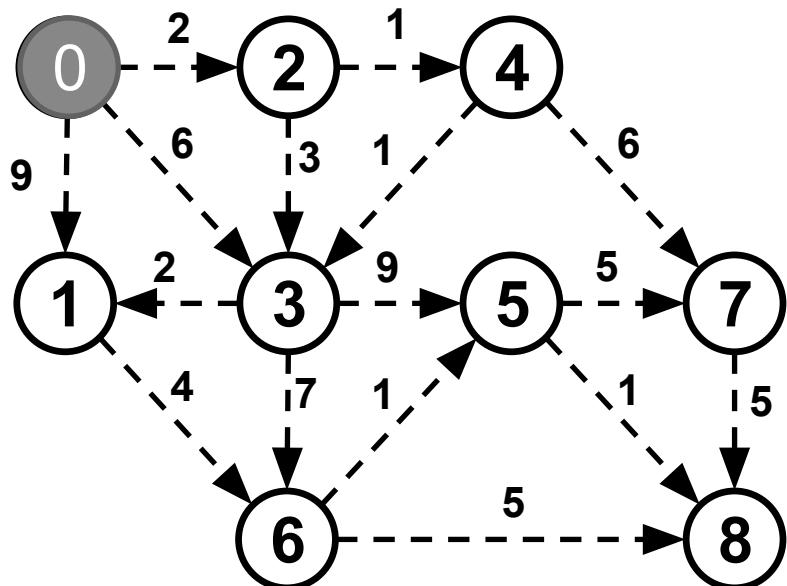
Веса рёбер больше 0.

Встретили вершину в  
первый раз

Если удалось  
релаксировать ребро, то  
надо изменить ее  
позицию в очереди с  
приоритетом.

```
void Dijkstra( G, s ) {  
    pi[V] = -1;  
    d[V] = INT_MAX;  
    d[s] = 0;  
    priority_queue<int> q; q.push( s );  
    while( !q.empty() ) {  
        u = q.top(); q.pop();  
        for( ( u, v ) : ребра из u ) {  
            if( d[v] == INT_MAX ) {  
                d[v] = d[u] + w(u, v);  
                pi[v] = u;  
                q.push( v );  
            } else if( Relax( u, v ) ) {  
                q.DecreaseKey( v, d[v] );  
            } } } }
```

# Алгоритм Дейкстры

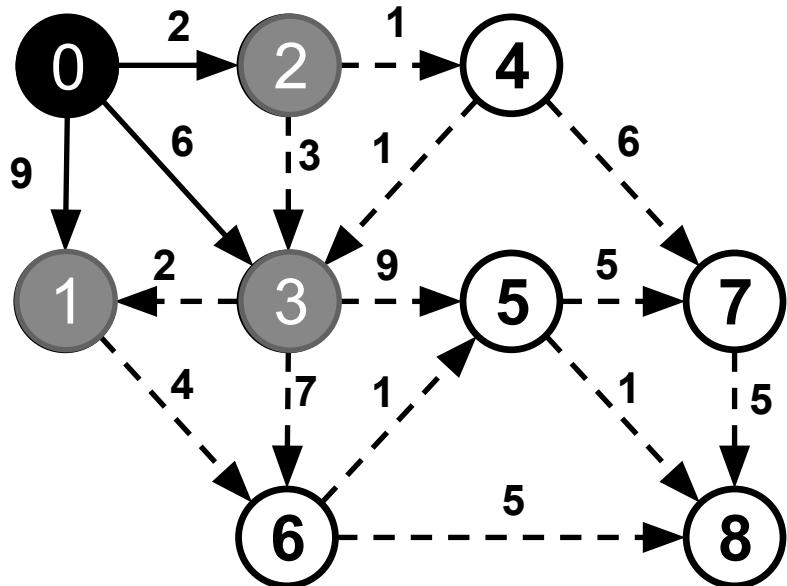


Q: (0, 0)

	pi
0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1
8	-1

	d
0	0
1	
2	
3	
4	
5	
6	
7	
8	

# Алгоритм Дейкстры



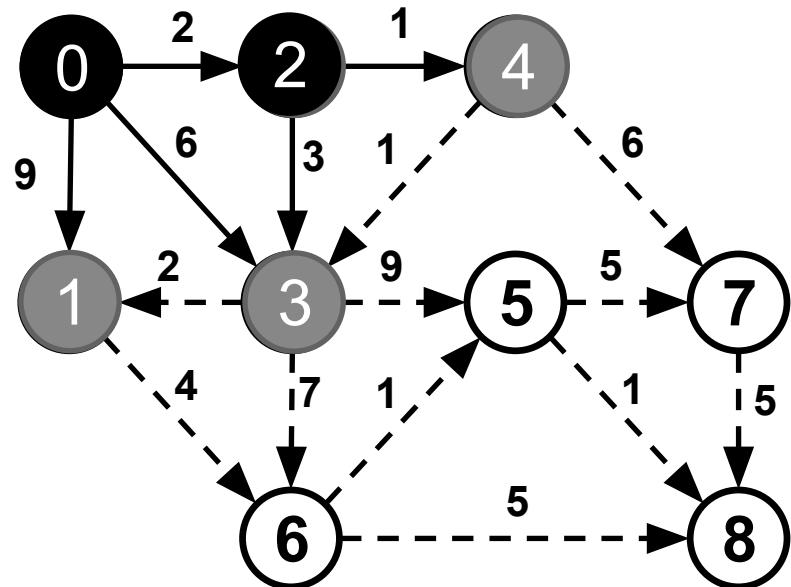
(0, 0)

	pi
0	-1
1	0
2	0
3	0
4	-1
5	-1
6	-1
7	-1
8	-1

	d
0	0
1	9
2	2
3	6
4	
5	
6	
7	
8	

Q: (2, 2) (3, 6) (1, 9)

# Алгоритм Дейкстры

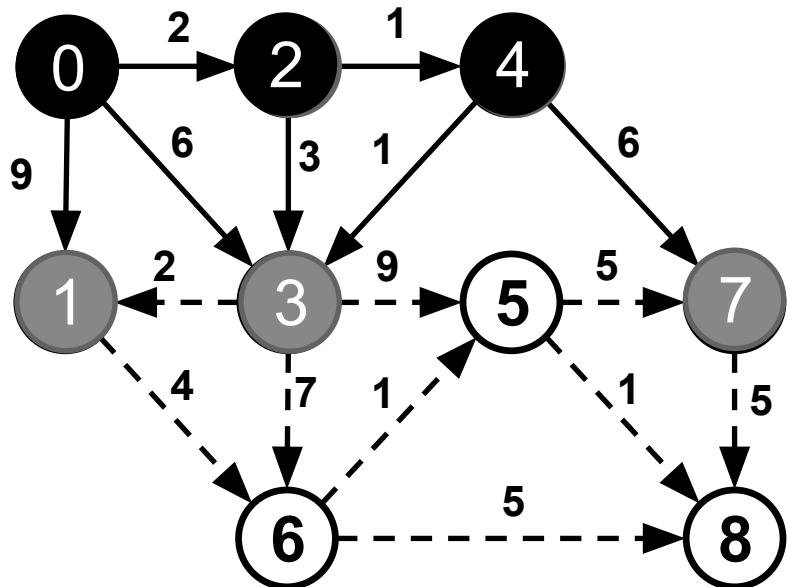


(2, 2)

	pi	d
0	-1	0
1	0	9
2	0	2
3	2	5
4	2	3
5	-1	
6	-1	
7	-1	
8	-1	

Q: (4, 3) (3, 5) (1, 9)

## Алгоритм Дейкстры



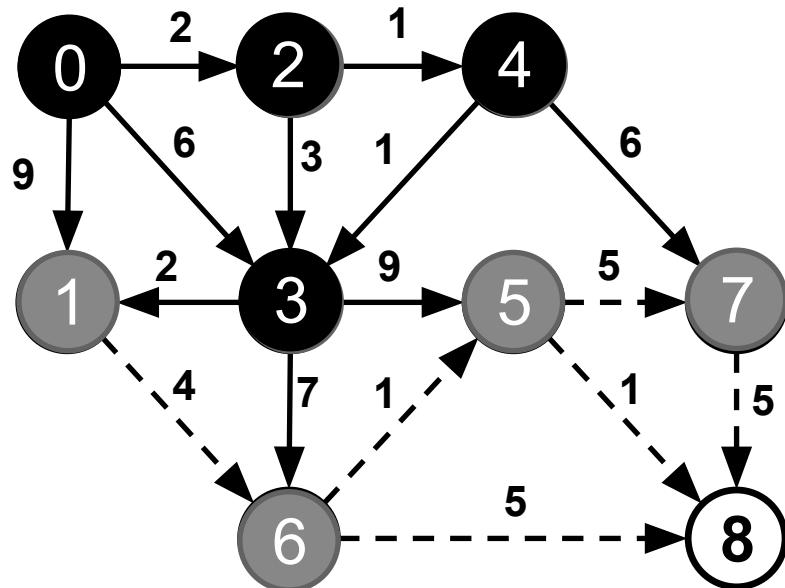
(4, 3)

	pi
0	-1
1	0
2	0
3	4
4	2
5	-1
6	-1
7	4
8	-1

	d
0	0
1	9
2	2
3	4
4	3
5	
6	
7	9
8	

Q: (3, 4) (1, 9) (7, 9)

## Алгоритм Дейкстры



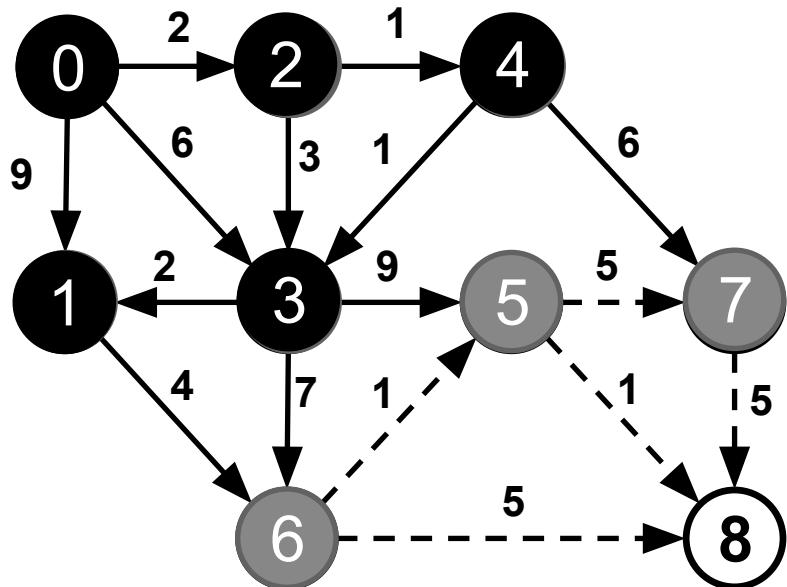
(3, 4)

	pi
0	-1
1	3
2	0
3	4
4	2
5	3
6	3
7	4
8	-1

	d
0	0
1	6
2	2
3	4
4	3
5	13
6	11
7	9
8	

Q: (1, 6) (7, 9) (5, 13) (6, 11)

## Алгоритм Дейкстры



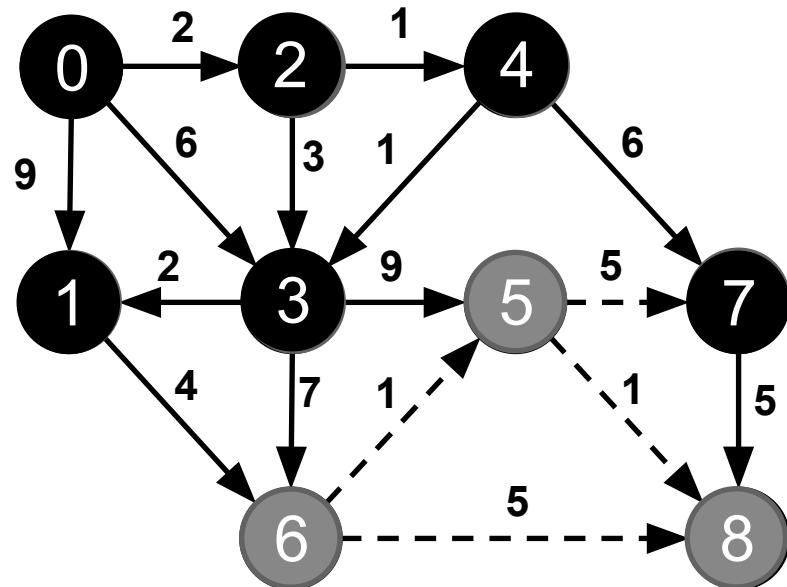
(1, 6)

	pi
0	-1
1	3
2	0
3	4
4	2
5	3
6	1
7	4
8	-1

	d
0	0
1	6
2	2
3	4
4	3
5	13
6	10
7	9
8	

Q: (7, 9) (5, 13) (6, 10)

## Алгоритм Дейкстры



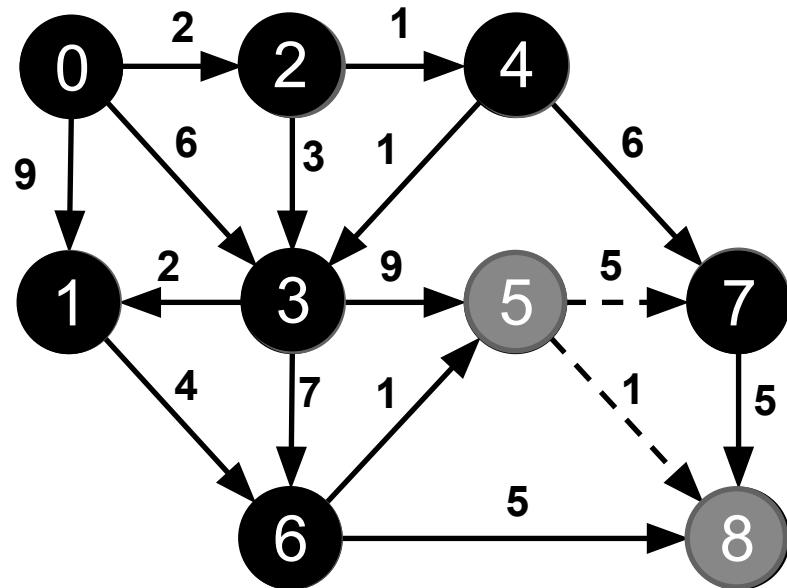
(7, 9)

	pi
0	-1
1	3
2	0
3	4
4	2
5	3
6	1
7	4
8	7

	d
0	0
1	6
2	2
3	4
4	3
5	13
6	10
7	9
8	14

Q: (6, 10) (5, 13) (8, 14)

## Алгоритм Дейкстры



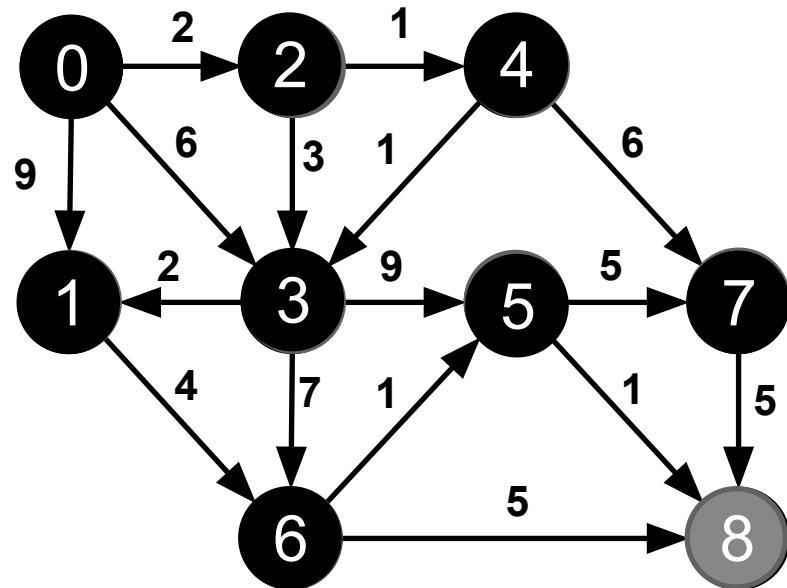
(6, 10)

	pi
0	-1
1	3
2	0
3	4
4	2
5	6
6	1
7	4
8	7

	d
0	0
1	6
2	2
3	4
4	3
5	11
6	10
7	9
8	14

Q: (5, 11) (8, 14)

## Алгоритм Дейкстры



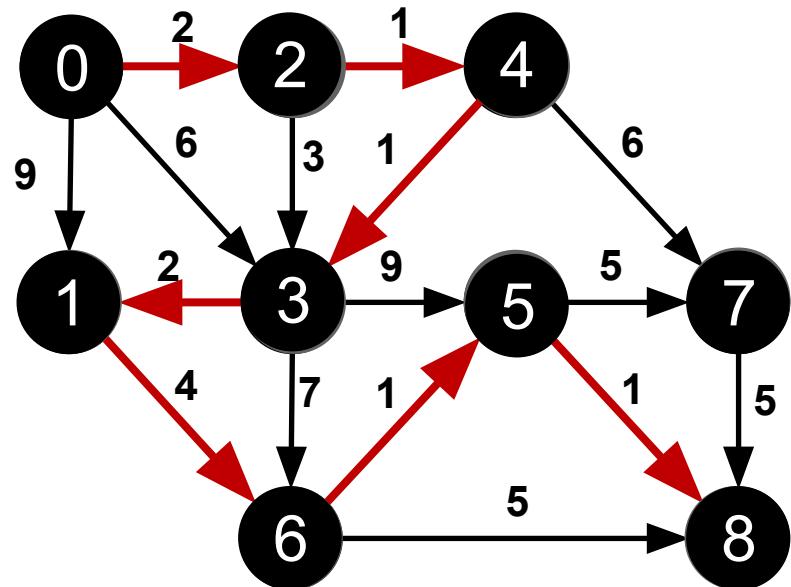
(5, 11)

	pi
0	-1
1	3
2	0
3	4
4	2
5	6
6	1
7	4
8	5

	d
0	0
1	6
2	2
3	4
4	3
5	11
6	10
7	9
8	12

Q: (8, 12)

## Алгоритм Дейкстры

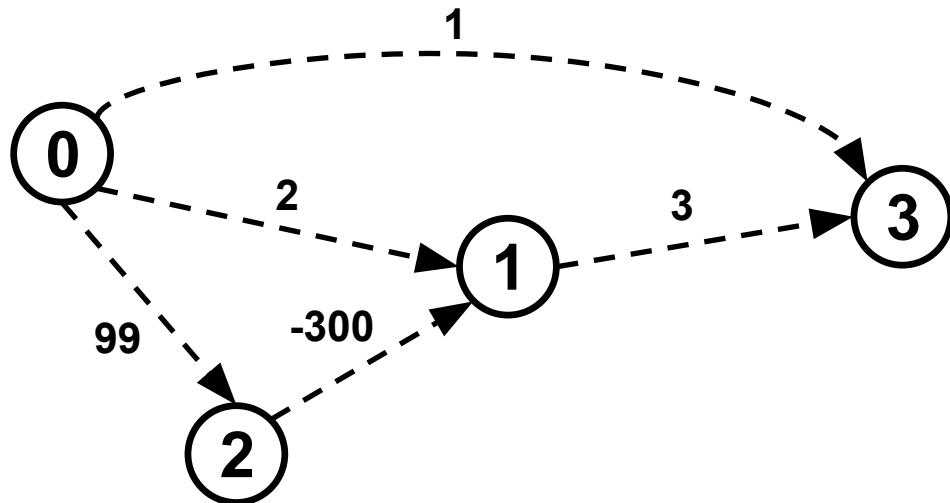


(8, 12)

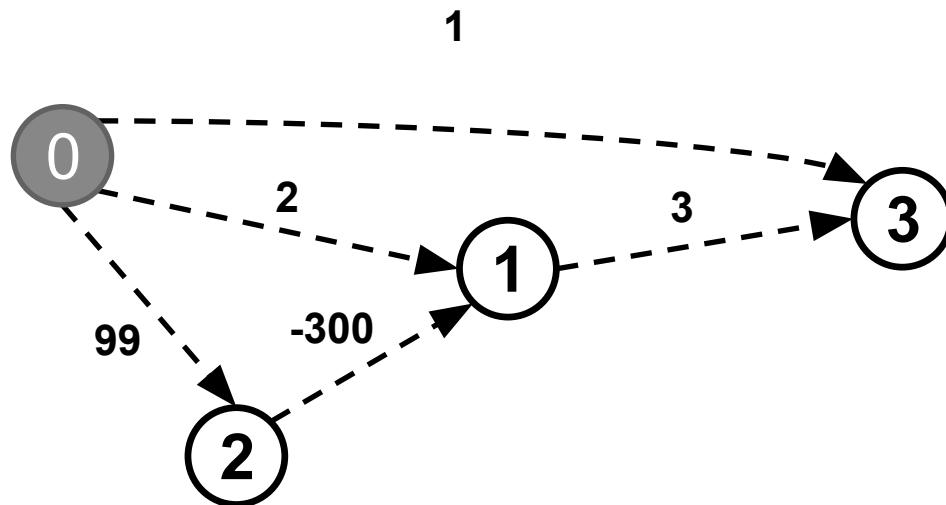
	pi	d
0	-1	0
1	3	6
2	0	2
3	4	4
4	2	3
5	6	11
6	1	10
7	4	9
8	5	12

Кратчайший путь из 0 в 8 восстанавливаем по массиву pi:  
0, 2, 4, 3, 1, 6, 5, 8

## Алгоритм Дейкстры: недопустимость отрицательных ребер



## Алгоритм Дейкстры: отрицательные ребра

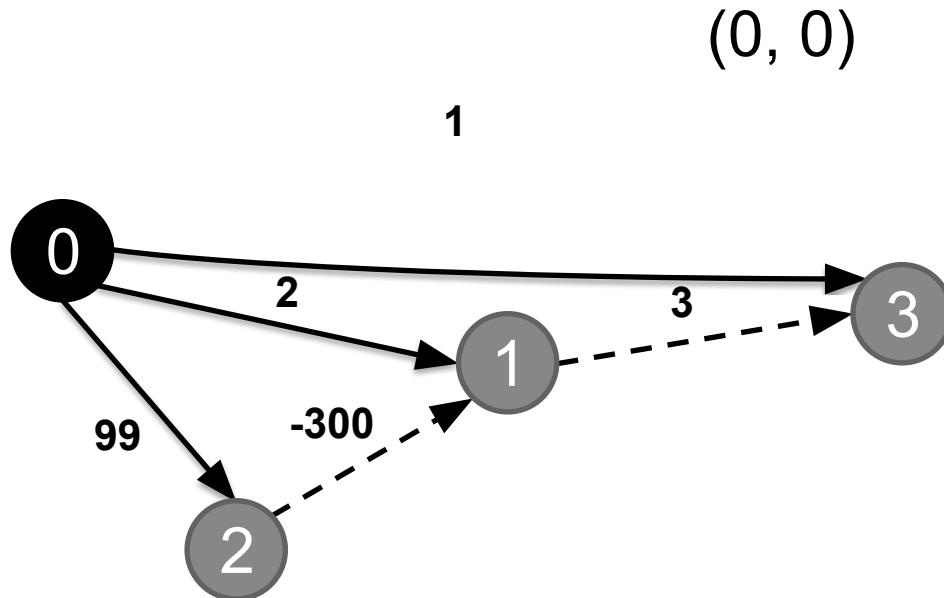


	pi
0	-1
1	-1
2	-1
3	-1

	d
0	0
1	1
2	2
3	3

Q: (0, 0)

## Алгоритм Дейкстры: отрицательные ребра



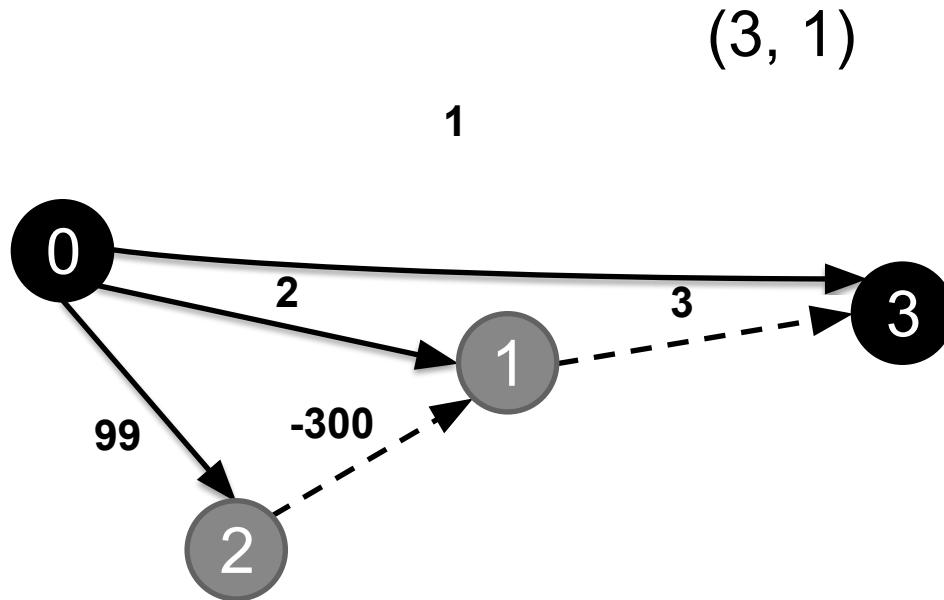
(0, 0)

	pi
0	-1
1	0
2	0
3	0

	d
0	0
1	2
2	99
3	1

Q: (3, 1) (1, 2) (2, 99)

## Алгоритм Дейкстры: отрицательные ребра



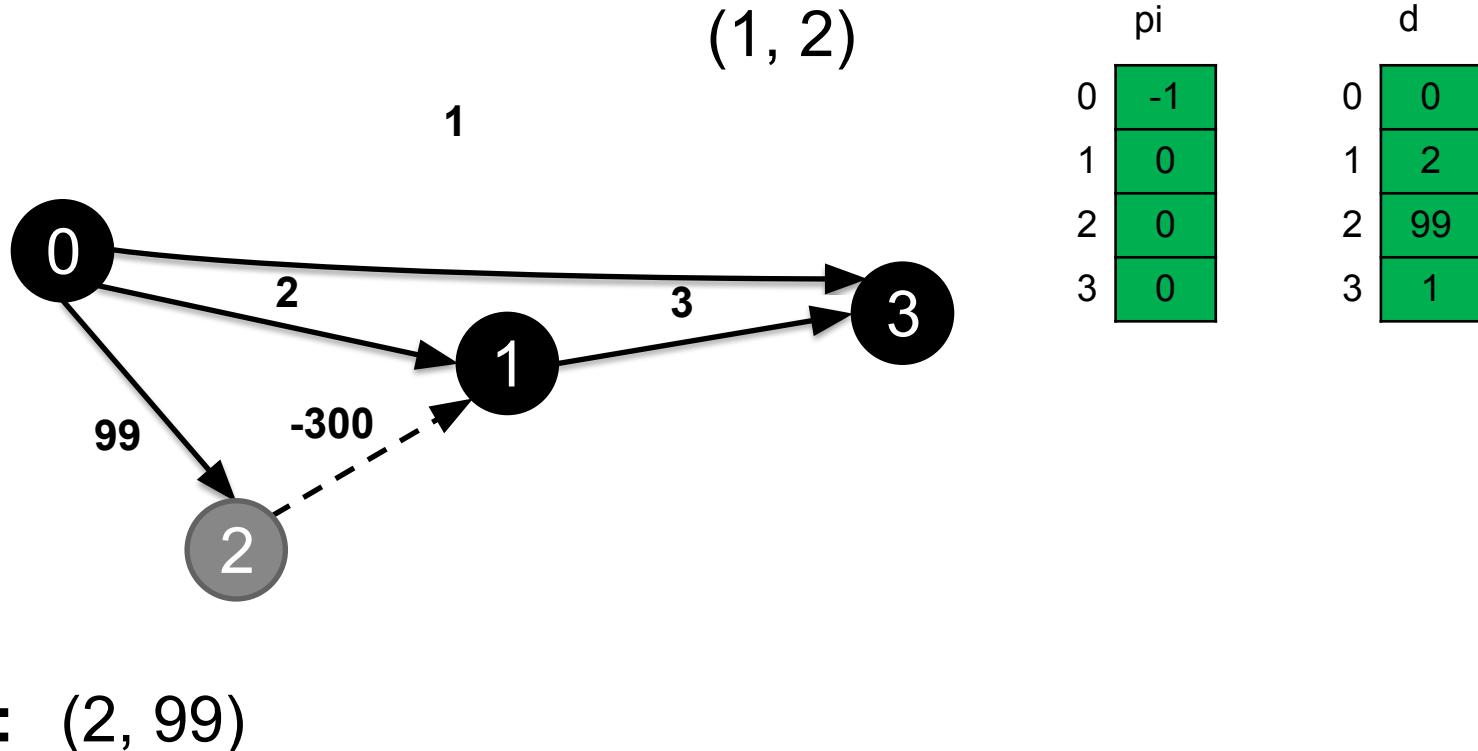
(3, 1)

	pi
0	-1
1	0
2	0
3	0

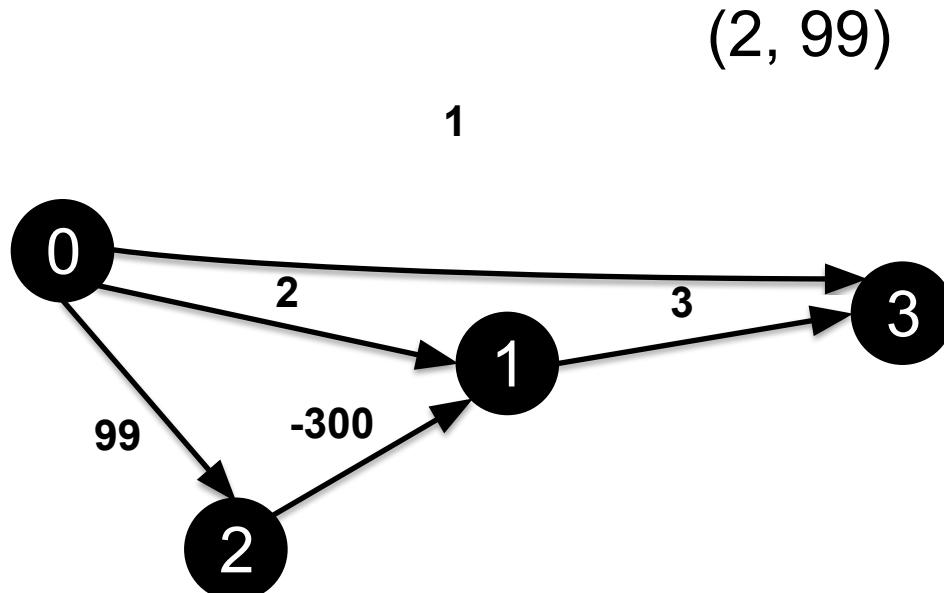
	d
0	0
1	2
2	99
3	1

Q: (1, 2) (2, 99)

## Алгоритм Дейкстры: отрицательные ребра



## Алгоритм Дейкстры: отрицательные ребра

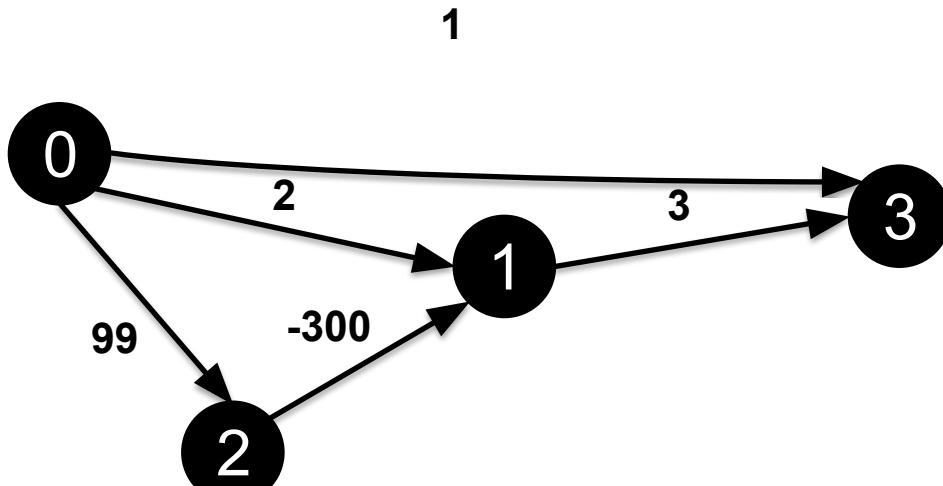


(2, 99)

	$\pi$
0	-1
1	2
2	0
3	0

	d
0	0
1	-201
2	99
3	1

## Алгоритм Дейкстры: отрицательные ребра



	pi
0	-1
1	2
2	0
3	0

	d
0	0
1	-201
2	99
3	1

Очередь опустела — алгоритм завершился. Но кратчайший путь из 0 в 3 должен проходить через вершину 1 и иметь вес -198. Ошибочный результат!

## Алгоритм Дейкстры

Оценка времени работы. Будем использовать в качестве реализации очереди с приоритетом двоичную кучу.

- Добавление узла:  $O(\log V)$ . Не более  $V$  операций.
- Уменьшение значения ключа:  $O(\log V)$ . Не более  $E$  операций.

Всего  $T = O((E + V) \cdot \log V)$ .

Используемая память  $M = O(V)$ .

# Алгоритм Дейкстры

Важная тонкость.

Как найти узел  $v$  в двоичной куче, чтобы вызывать  
`q.DecreaseKey(v, d[v])`?

0	1	2	3
(1, 6)	(7, 9)	(6, 11)	(5, 13)

Решение 1. Хранить позицию  $i$  в куче для каждого узла  $v$ . В момент изменения позиции в куче обновлять эту позицию, хранящуюся в описании узла.

Решение 2. Использовать вместо бинарной кучи множество `set<pair<d, v>>`.

Во время релаксации удалять старую пару  $\langle d[v], v \rangle$ , добавлять новую.

## Алгоритм А\*

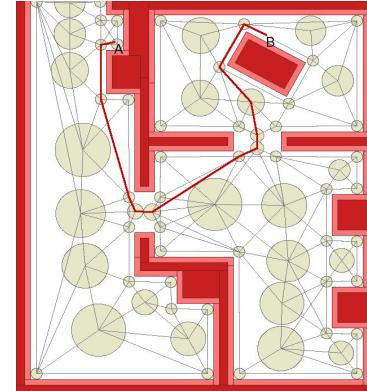
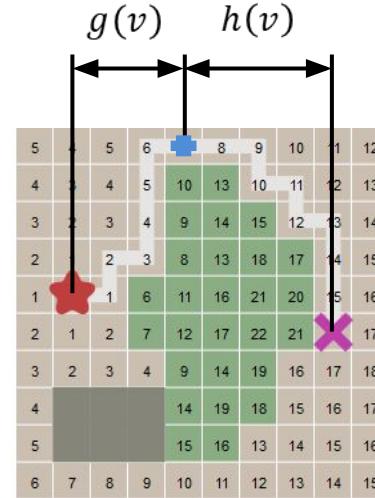
**Алгоритм А\* (A-star)** - это алгоритм поиска кратчайшего пути между двумя вершинами графа.

В процессе работы алгоритма рассчитывается функция

$$f(v) = g(v) + h(v), \text{ где}$$

$g(v)$  - наименьшая стоимость пути из стартовой вершины в  $v$

$h(v)$  - эвристическое приближение стоимости пути от  $v$  к конечной цели



# Алгоритм A\*

Сравнение как и в алгоритме Дейкстры

Значение эвристики используем только для изменения порядка в очереди с приоритетом

```
void aStar( G, s ) {  
    pi[V] = -1;  
    d[V] = INT_MAX;  
    d[s] = 0;  
    priority_queue<int> q; q.push( s );  
    while( !q.empty() ) {  
        u = q.top(); q.pop();  
        for( ( u, v ) : E ) {  
            if( d[v] == INT_MAX ) {  
                q.push( v );  
            }  
            if( d[v] > d[u] + w(u, v) ) {  
                d[v] = d[u] + w(u, v);  
                pi[v] = u;  
                q.DecreaseKey( v, d[v] + h(v) );  
            }  
        }  
    }  
}
```

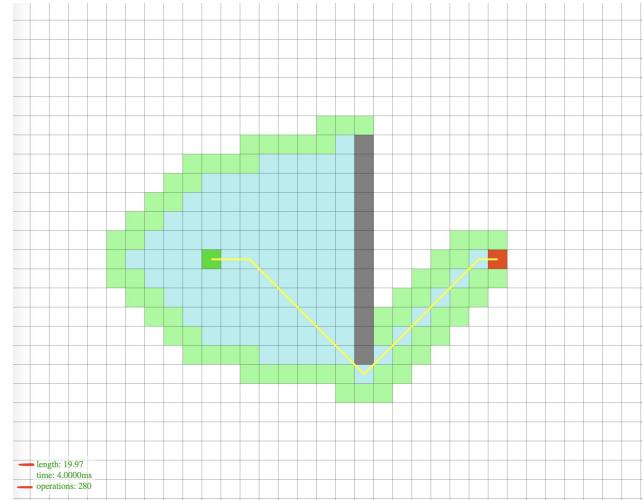
# Алгоритм А\*

## Дейкстры



Длина пути – 19.97  
Кол-во итераций – 1664

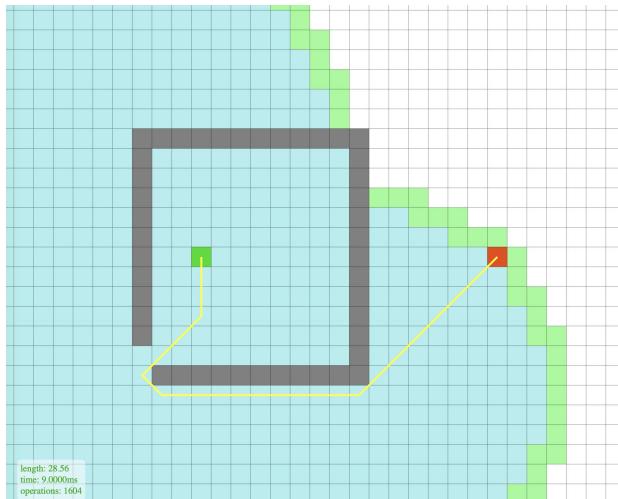
## А\*



Длина пути – 19.97  
Кол-во итераций – 280

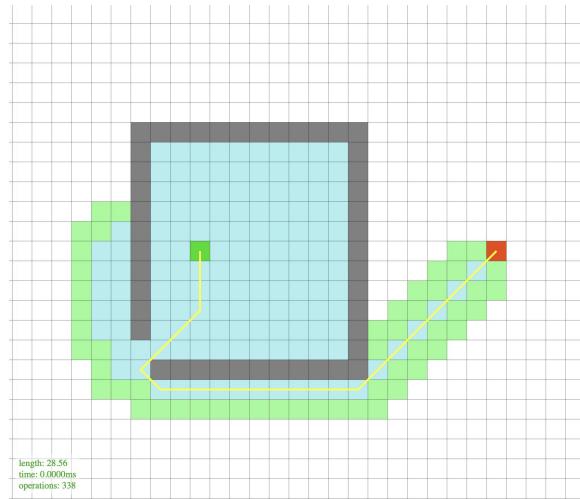
# Алгоритм А\*

## Дейкстра



Длина пути – 28.56  
Кол-во итераций – 1604

## А\*



Длина пути – 28.56  
Кол-во итераций – 338

# Алгоритм А\*

Эвристика  $h(v)$  должна быть:

- **Допустимой** – для любой вершины  $v$  значение  $h(v)$  меньше или равно весу кратчайшего пути от  $v$  до цели.
- **Монотонной** – для любой вершины  $v_1$  и ее потомка  $v_2$  разность  $h(v_1)$  и  $h(v_2)$  не превышает фактического веса ребра  $w(v_1, v_2)$  от  $v_1$  до  $v_2$ , а эвристическая оценка целевого состояния равна нулю.

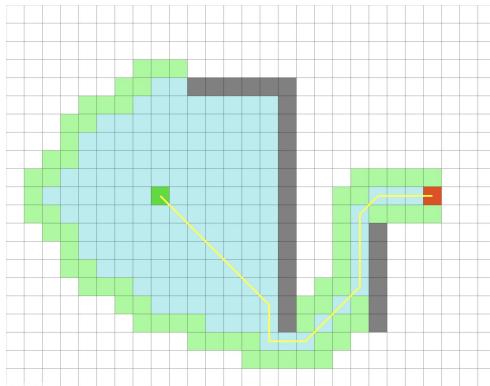
Любая монотонная эвристика допустима, но обратное неверно.

Примеры эвристик:

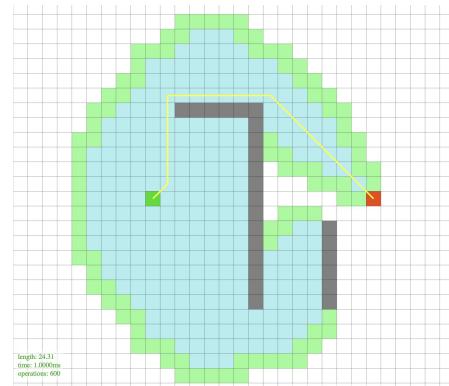
- Манхэттенское расстояние -  $h(v) = |v.x - u.x| + |v.y - u.y|$
- Расстояние Чебышёва -  $h(v) = \max(|v.x - u.x|, |v.y - u.y|)$
- Евклидово расстояние -  $h(v) = \sqrt{(v.x - u.x)^2 + (v.y - u.y)^2}$

# Алгоритм А\*

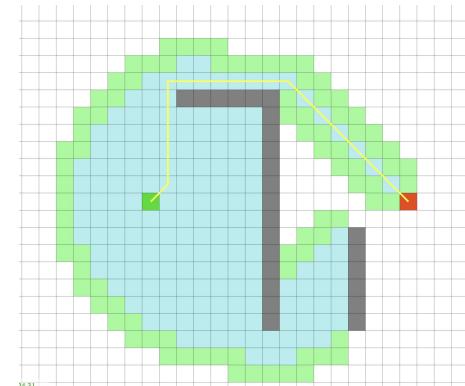
Манхеттенское  
расстояние



Расстояние  
Чебышёва



Евклидово  
расстояние



Длина пути во всех примерах одинаковая – 24.31

<http://qiao.github.io/PathFinding.js/visual/>

# Алгоритм Беллмана-Форда

Поиск кратчайших путей из выделенной вершины до всех остальных (SSSP)

Если в графе нет циклов отрицательного веса, достижимых из  $s$ , то алгоритм Беллмана-Форда находит все кратчайшие пути из  $s$  до остальных вершин.

Если в графе есть достижимый из  $s$  цикл отрицательного веса, то алгоритм Беллмана-Форда сообщит о его наличии. (и может выдать его).

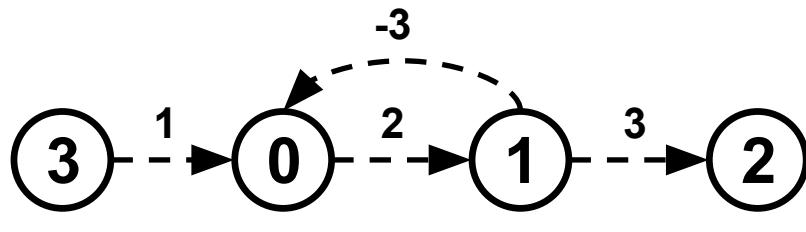
```
bool BellmanFord( G, s ) {
    d[s] = 0;
    for( int i = 0; i < V - 1; ++i ) {
        for( (u, v) : E ) Relax( u, v );
    }
    // Детектирование цикла.
    for( (u, v) : E ) {
        if( Relax( u, v ) )
            return false;
    }
    return true;
}
```

Сложность:  $O(|V| \cdot |E|)$

# Алгоритм Беллмана-Форда

Релаксируем все рёбра  $V - 1$  раз

Стартовая вершина 3, инициализируем  $d[3] = 0$



	pi	d
0	-1	inf
1	-1	inf
2	-1	inf
3	-1	0

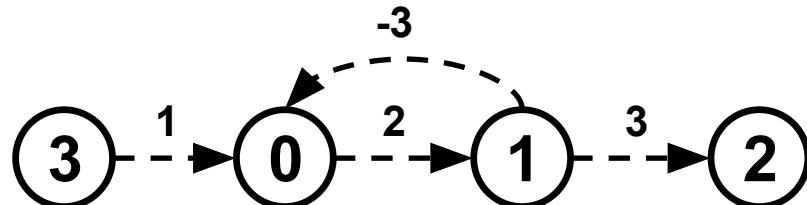
Процедура релаксации ребра

```
bool Relax( u, v ) {  
    if( d[v] > d[u] + w( u, v ) )  
    {  
        d[v] = d[u] + w( u, v );  
        pi[v] = u;  
        return true;  
    }  
    return false;  
}
```

# Алгоритм Беллмана-Форда

Релаксируем все рёбра  $V - 1$  раз

Итерация 1



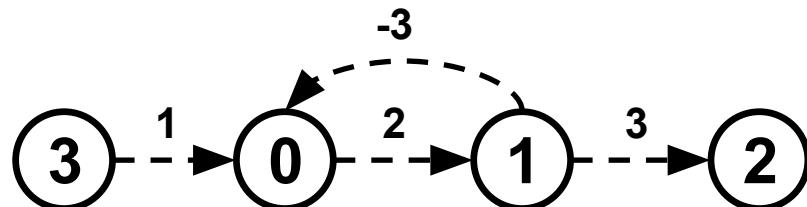
pi
0
1
2
3

d
0
1
inf
inf
0

# Алгоритм Беллмана-Форда

Релаксируем все рёбра  $V - 1$  раз

Итерация 2



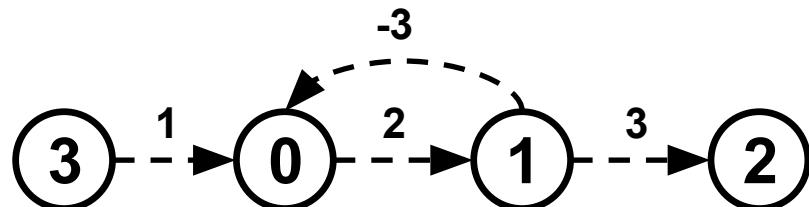
$p_i$
0
1
2
3

$d$
0
1
2
3

# Алгоритм Беллмана-Форда

Релаксируем все рёбра  $V - 1$  раз

Итерация 3

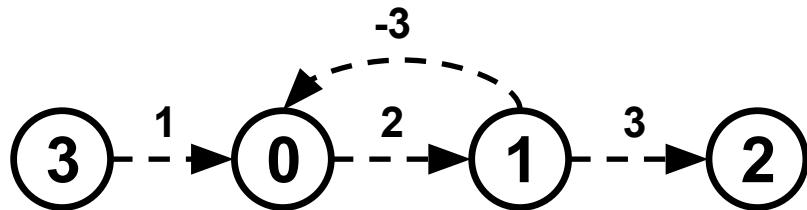


	pi	d
0	1	-1
1	0	2
2	1	5
3	-1	0

# Алгоритм Беллмана-Форда

Если отрицательный цикл есть, то на  $V$  – ой итерации должна произойти релаксация

Итерация 4

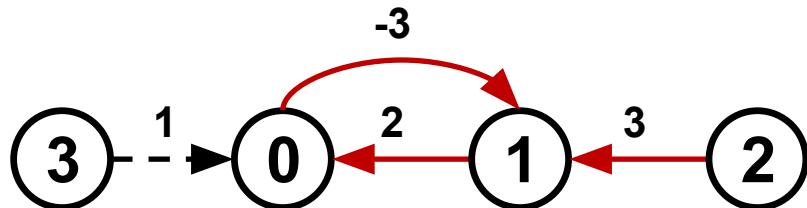


	$p_i$	$d$
0	1	-2
1	0	1
2	1	4
3	-1	0

Нужно запомнить номер вершины, на которой произошла релаксация.

## Алгоритм Беллмана-Форда

Для поиска вершины, лежащей на цикле, проходим по предкам  $V$  раз от запомненной на предыдущем шаге вершины.

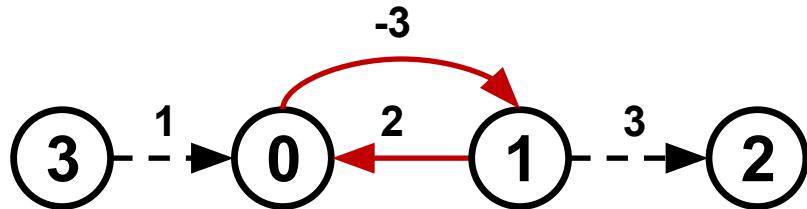


	pi	d
0	1	-2
1	0	1
2	1	4
3	-1	0

После этого мы точно окажемся внутри цикла.

# Алгоритм Беллмана-Форда

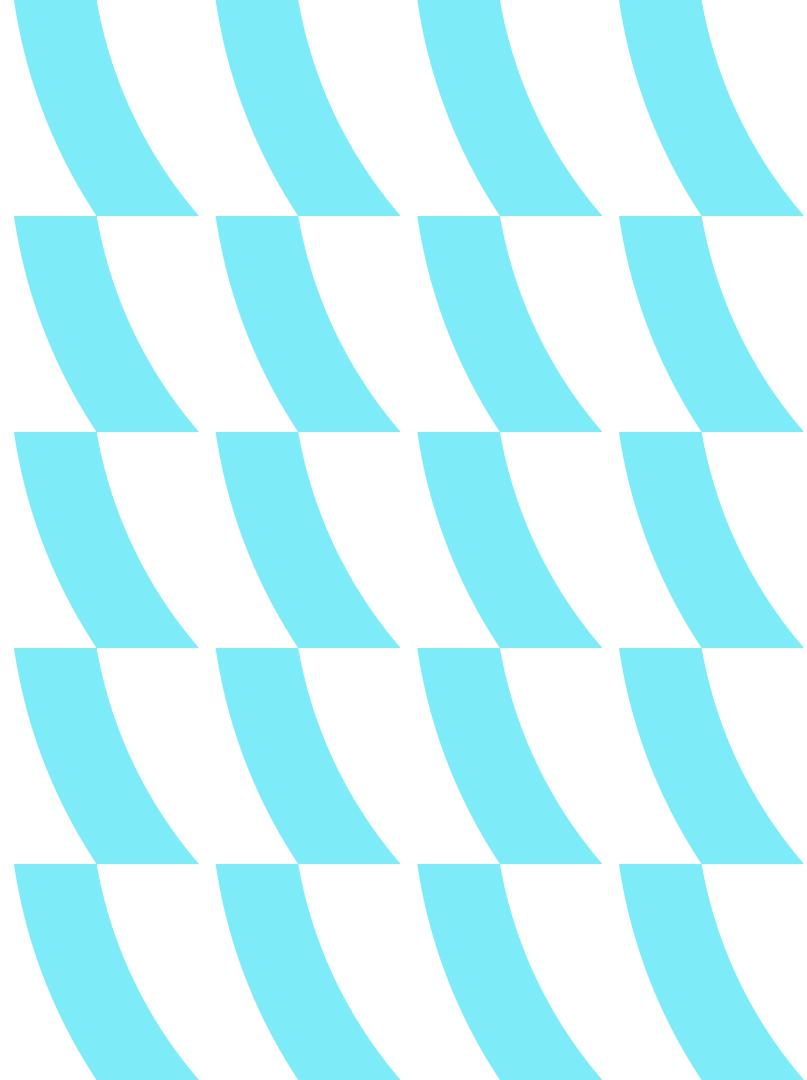
Для поиска цикла от найденной вершины проходим по предкам, пока не встретим эту же вершину.



	pi	d
0	1	-2
1	0	1
2	1	4
3	-1	0

А эту же вершину встретим обязательно, потому что находимся внутри цикла.

# **Остовные деревья**



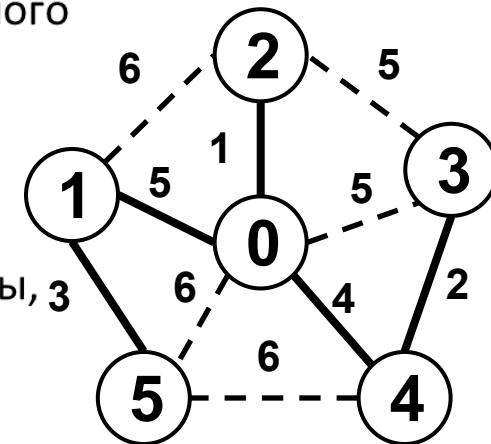
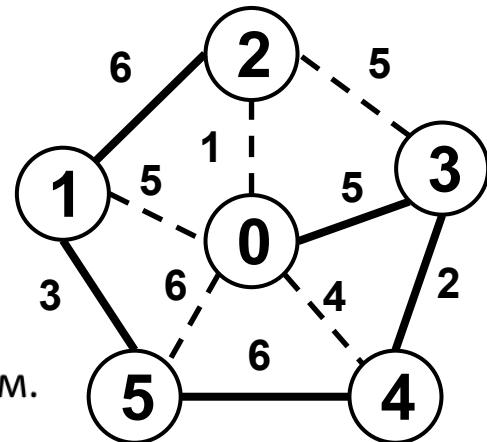
# Остовные деревья

Пусть  $G$  – связный неориентированный граф.

**Остовное дерево** графа  $G$  состоит из минимального подмножества рёбер графа, таких, что из любой вершины графа можно попасть в любую другую вершину, двигаясь по этим рёбрам.

**Остовное дерево** графа  $G$  — ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины.

**Минимальное остовное дерево (MST)** графа  $G$  — это его ациклический связный подграф, в который входят все его вершины, 3 обладающий минимальным суммарным весом ребер.



## Алгоритм Прима

Алгоритм Прима – алгоритм поиска минимального оствовного дерева.

1. Выбираем произвольную стартовую вершину – начало строящегося дерева. Строим очередь с приоритетом вершин, до которых есть ребро от стартовой. Приоритет - длина ребра до стартовой.
2. Итеративно добавляем к дереву вершину из очереди с ребром до дерева, имеющим вес = приоритет вершины. Добавляем новые вершины, доступные из добавленной вершины по ребрам графа с соответствующими приоритетами. Либо обновляем приоритет вершины в очереди, если от добавленной вершины к ней ведет более легкое ребро.

# Алгоритм Прима

Алгоритм Прима похож на алгоритм Дейкстры, но в качестве весовой функции используем длину кратчайшего ребра от текущей вершины до дерева.

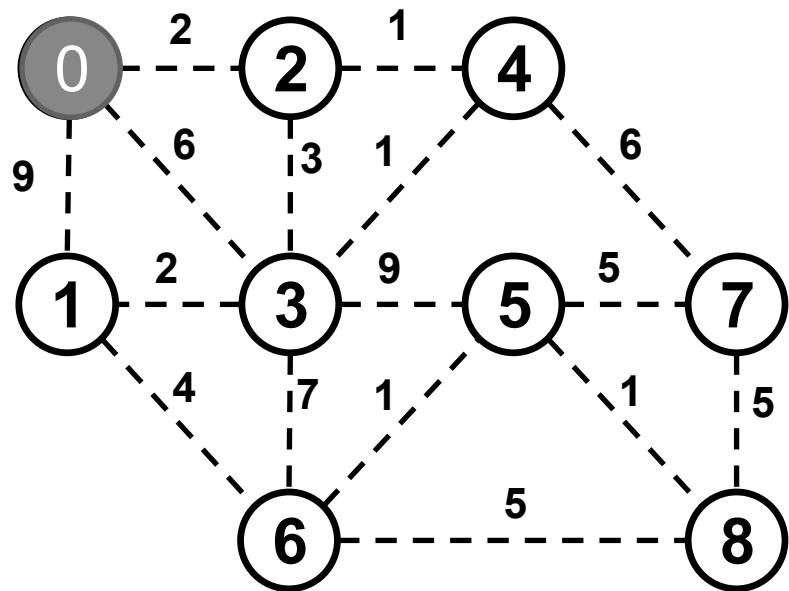
Веса рёбер больше 0.

Встретили вершину в  
первый раз

Если удалось  
релаксировать  
вершину, то надо  
изменить ее позицию в  
очереди с  
приоритетом.

```
void Prima( G, s ) {  
    pi[V] = -1;  
    min_e[V] = INT_MAX;  
    min_e[s] = 0;  
    priority_queue<int> q; q.push( s );  
    while( !q.empty() ) {  
        u = q.top(); q.pop();  
        for( ( u, v ) : ребра из u ) {  
            if( min_e[v] == INT_MAX ) {  
                min_e[v] = w(u, v);  
                pi[v] = u;  
                q.push( v );  
            } else if(Relax( u, v )) {  
                q.DecreaseKey( v, min_e[v] );  
            } } } }
```

# Алгоритм Прима

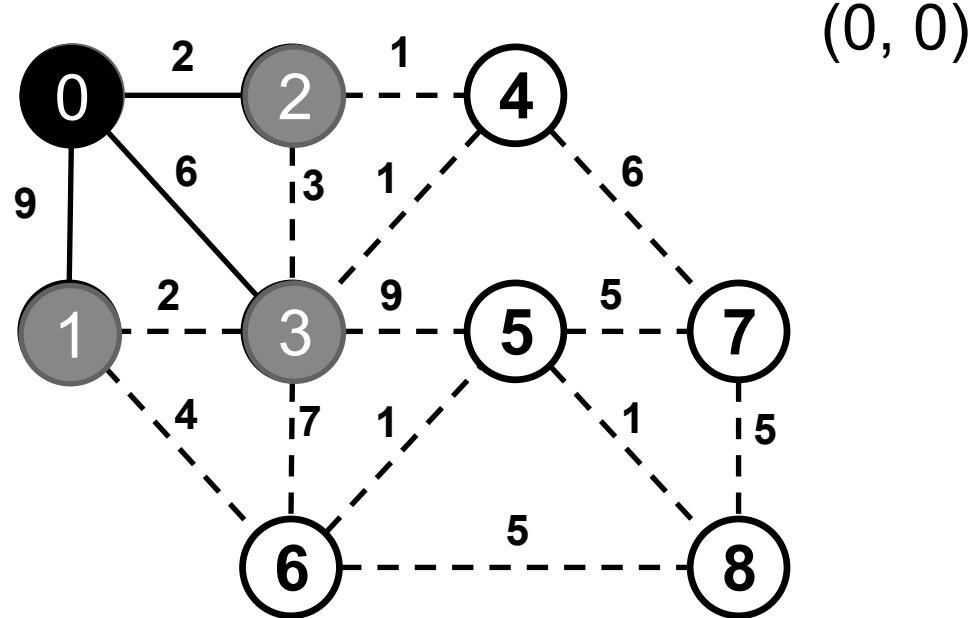


Q: (0, 0)

	pi
0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1
8	-1

	min_e
0	0
1	
2	
3	
4	
5	
6	
7	
8	

# Алгоритм Прима



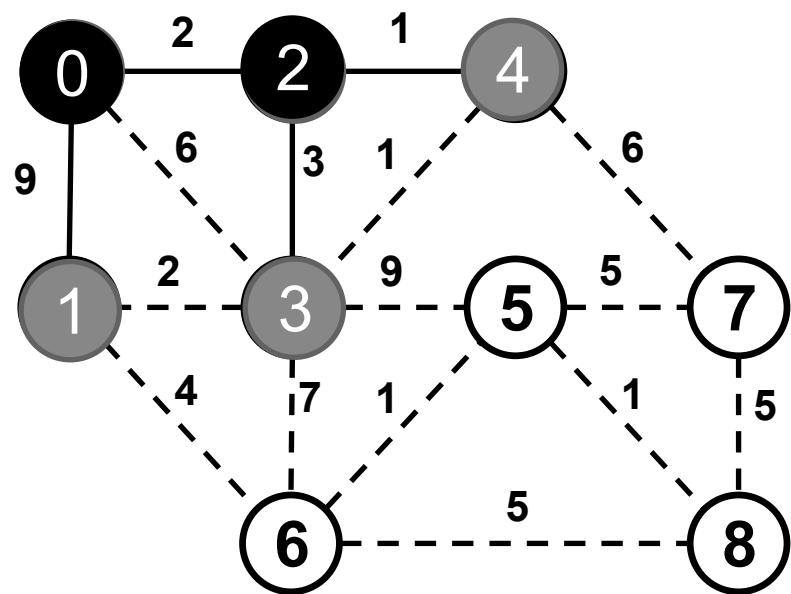
(0, 0)

Q: (2, 2) (3, 6) (1, 9)

	pi
0	-1
1	0
2	0
3	0
4	-1
5	-1
6	-1
7	-1
8	-1

	min_e
0	0
1	9
2	2
3	6
4	
5	
6	
7	
8	

# Алгоритм Прима



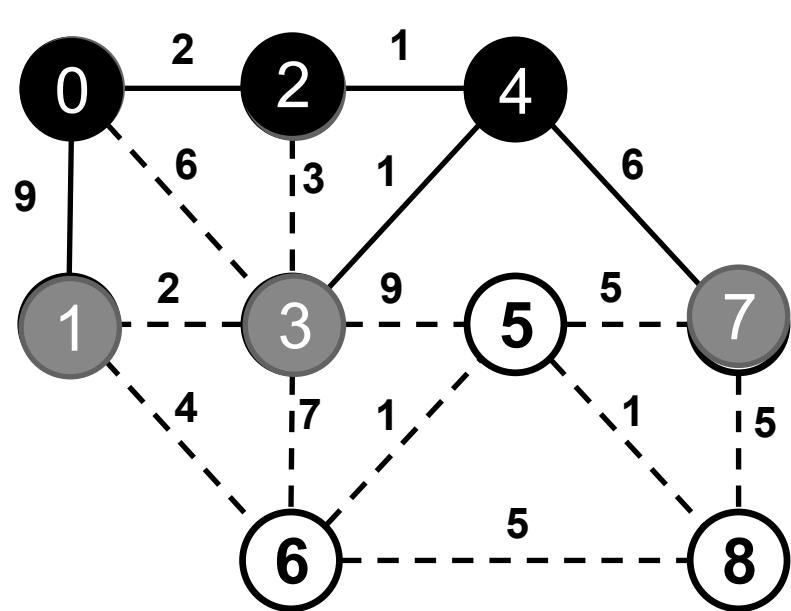
(2, 2)

Q: (4, 1) (3, 3) (1, 9)

	pi
0	-1
1	0
2	0
3	2
4	2
5	-1
6	-1
7	-1
8	-1

	min_e
0	0
1	9
2	2
3	3
4	1
5	
6	
7	
8	

# Алгоритм Прима



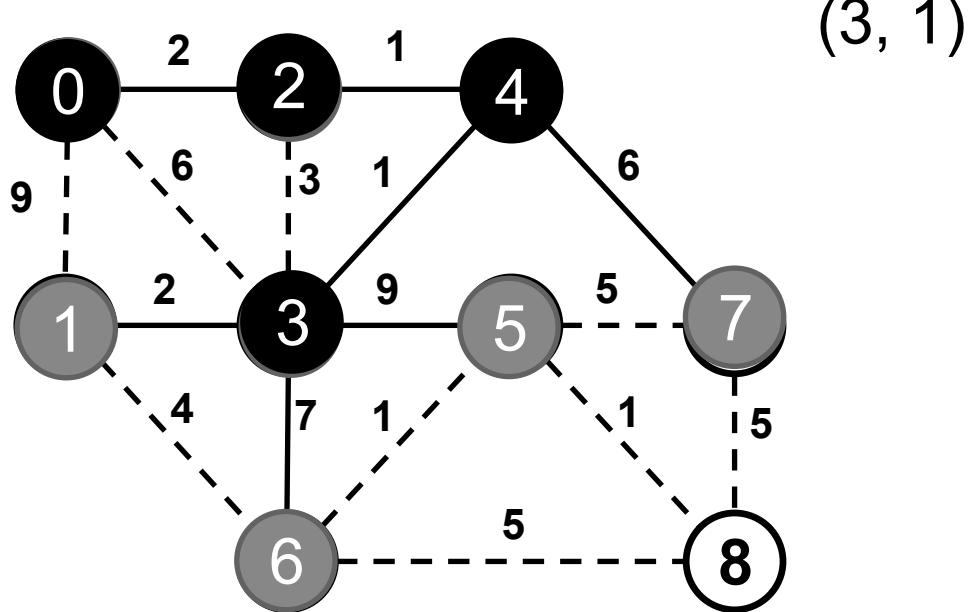
(4, 1)

Q: (3, 1) (1, 9) (7, 6)

	pi
0	-1
1	0
2	0
3	4
4	2
5	-1
6	-1
7	4
8	-1

	min_e
0	0
1	9
2	2
3	1
4	1
5	
6	
7	6
8	

# Алгоритм Прима

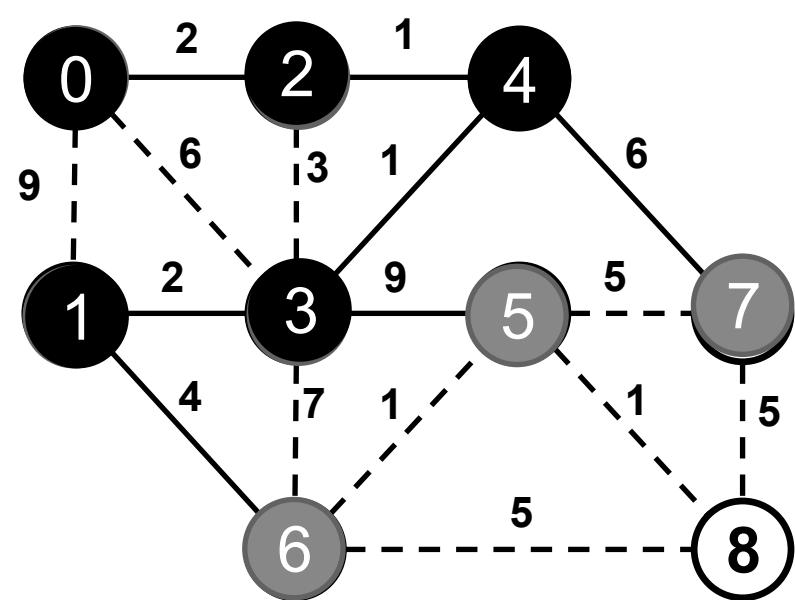


Q: (1, 2) (7, 6) (6, 7) (5, 9)

pi
-1
3
0
4
2
3
2
4
-1

min_e
0
2
2
1
1
9
7
6

# Алгоритм Прима



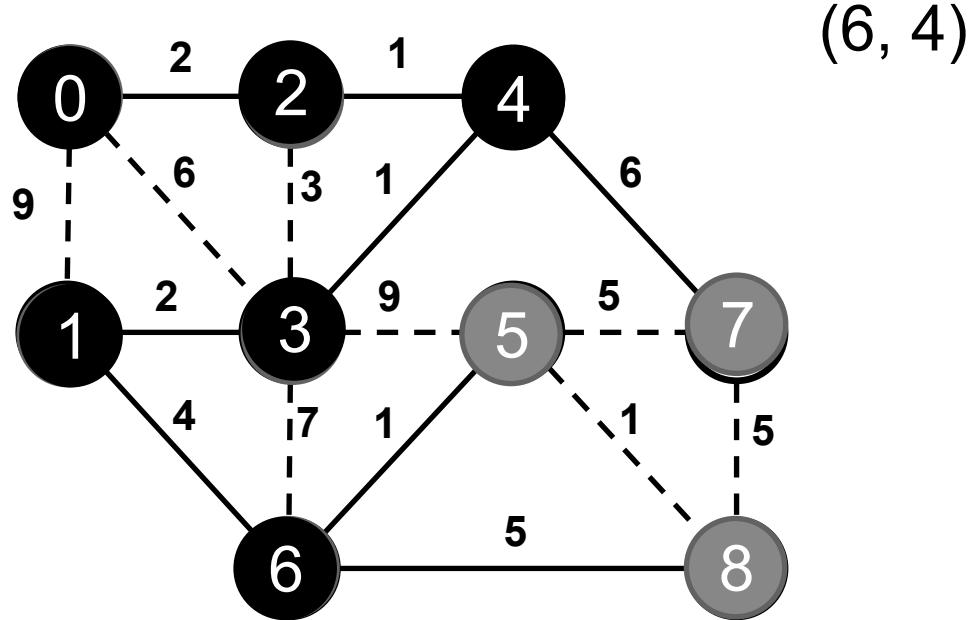
(1, 2)

Q: (6, 4) (7, 6) (5, 9)

	pi
0	-1
1	3
2	0
3	4
4	2
5	3
6	1
7	4
8	-1

	min_e
0	0
1	2
2	2
3	1
4	1
5	9
6	4
7	6
8	

# Алгоритм Прима



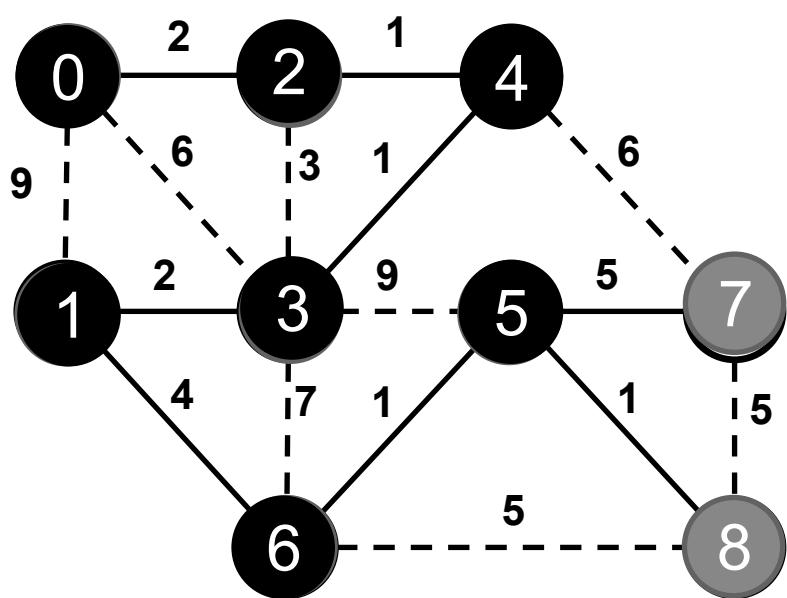
(6, 4)

Q: (5, 1) (8, 5) (7, 6)

	pi
0	-1
1	3
2	0
3	4
4	2
5	6
6	1
7	4
8	6

	min_e
0	0
1	2
2	2
3	1
4	1
5	1
6	4
7	6
8	5

# Алгоритм Прима



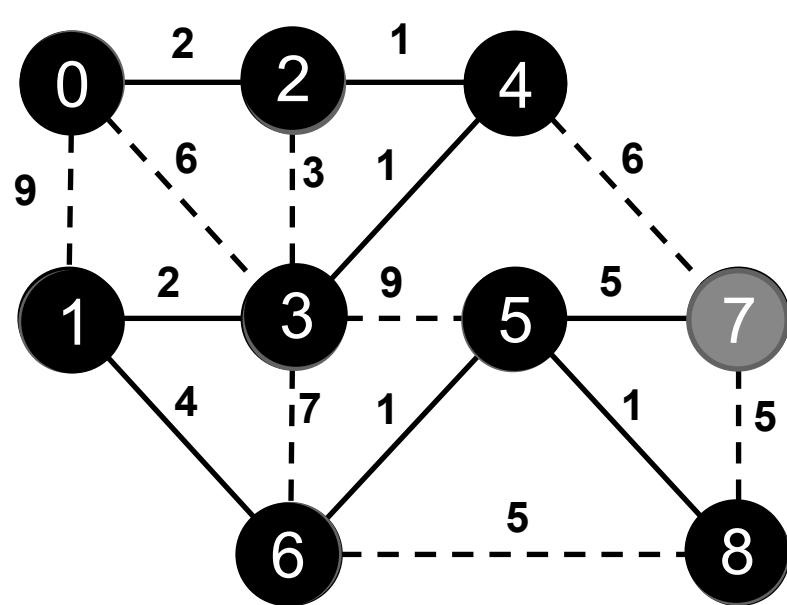
(5, 1)

Q: (8, 1) (7, 5)

	pi
0	-1
1	3
2	0
3	4
4	2
5	6
6	1
7	5
8	5

	min_e
0	0
1	2
2	2
3	1
4	1
5	1
6	4
7	5
8	1

# Алгоритм Прима



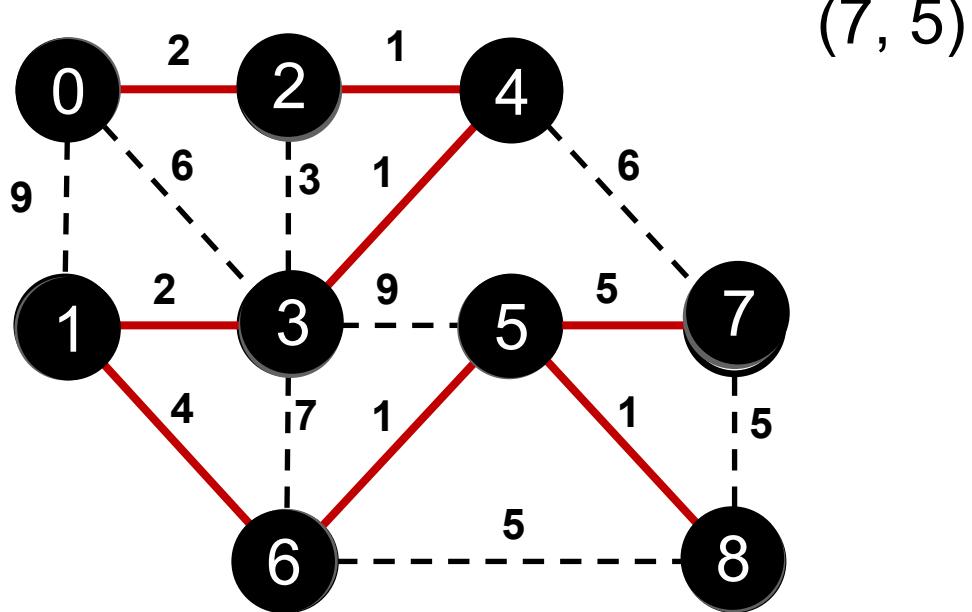
(8, 1)

Q: (7, 5)

	pi
0	-1
1	3
2	0
3	4
4	2
5	6
6	1
7	5
8	5

	min_e
0	0
1	2
2	2
3	1
4	1
5	1
6	4
7	5
8	1

# Алгоритм Прима



Q:

	pi
0	-1
1	3
2	0
3	4
4	2
5	6
6	1
7	5
8	5

	min_e
0	0
1	2
2	2
3	1
4	1
5	1
6	4
7	5
8	1

## Алгоритм Прима: оценка

В качестве очереди с приоритетом для вершин можно использовать кучу или сбалансированное дерево поиска, как в алгоритме Дейкстры.

В этом случае одна итерация состоит из

- 1) Извлечение вершины из очереди с приоритетом –  $O(\log V)$
- 2) Обработка ребер, инцидентных извлеченной вершине –  $O(\log V)$  на каждое ребро.

Пункт 1) будет выполнять  $V$  раз.

Пункт 2) будет выполняться  $E$  раз.

Общая оценка времени работы:  $T = O((E + V) \cdot \log V)$

Оценка доп.памяти:  $M = O(V)$ .

# Алгоритм Крускала

Алгоритм Крускала – еще один алгоритм поиска минимального остовного дерева.

1. Сортируем все ребра графа по весу.
2. Инициализируем лес деревьев. Изначально каждая вершина – маленькое дерево (пень).
3. Последовательно рассматриваем ребра графа в порядке возрастания веса.
  - Если очередное ребро соединяет два разных дерева из леса, то объединяем эти два дерева этим ребром в одно дерево.
  - Если очередное ребро соединяет две вершины одного дерева из леса, то пропускаем такое ребро.

Повторяем 3), пока в лесу не останется одно дерево.

## Алгоритм Крускала

$$w(2, 4) = 1$$

$$w(3, 4) = 1$$

$$w(5, 6) = 1$$

$$w(5, 8) = 1$$

$$w(0, 2) = 2$$

$$w(1, 3) = 2$$

$$w(2, 3) = 3$$

$$w(1, 6) = 4$$

$$w(5, 7) = 5$$

$$w(6, 8) = 5$$

$$w(7, 8) = 5$$

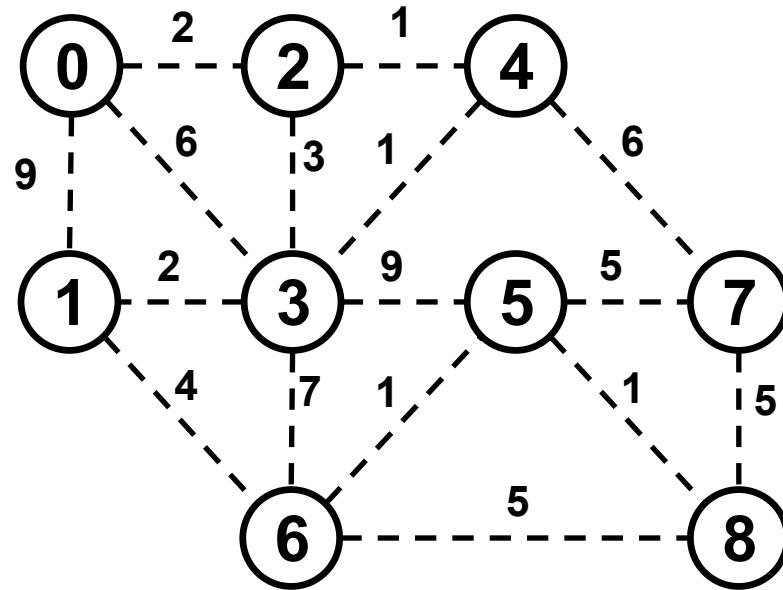
$$w(0, 3) = 6$$

$$w(4, 7) = 6$$

$$w(3, 6) = 7$$

$$w(0, 1) = 9$$

$$w(3, 5) = 9$$



## Алгоритм Крускала

$$w(2, 4) = 1$$

$$w(3, 4) = 1$$

$$w(5, 6) = 1$$

$$w(5, 8) = 1$$

$$w(0, 2) = 2$$

$$w(1, 3) = 2$$

$$w(2, 3) = 3$$

$$w(1, 6) = 4$$

$$w(5, 7) = 5$$

$$w(6, 8) = 5$$

$$w(7, 8) = 5$$

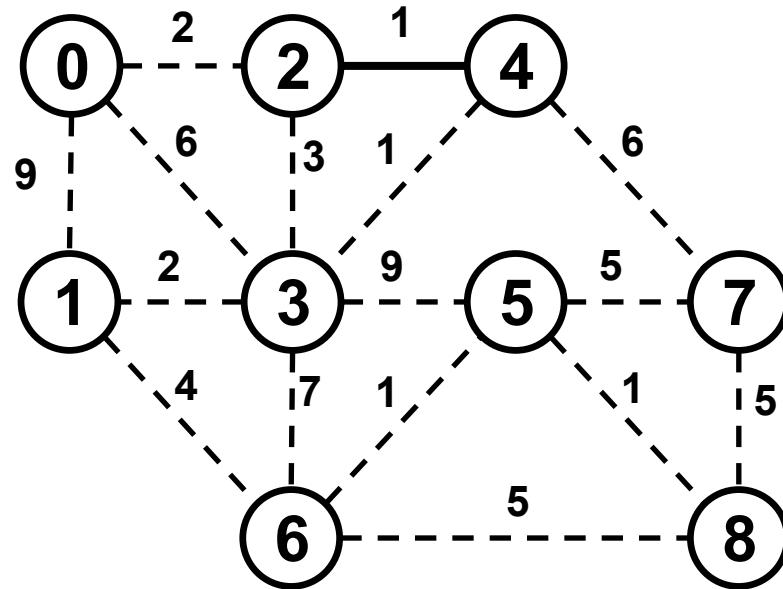
$$w(0, 3) = 6$$

$$w(4, 7) = 6$$

$$w(3, 6) = 7$$

$$w(0, 1) = 9$$

$$w(3, 5) = 9$$



## Алгоритм Крускала

$$w(2, 4) = 1$$

$$w(3, 4) = 1$$

$$w(5, 6) = 1$$

$$w(5, 8) = 1$$

$$w(0, 2) = 2$$

$$w(1, 3) = 2$$

$$w(2, 3) = 3$$

$$w(1, 6) = 4$$

$$w(5, 7) = 5$$

$$w(6, 8) = 5$$

$$w(7, 8) = 5$$

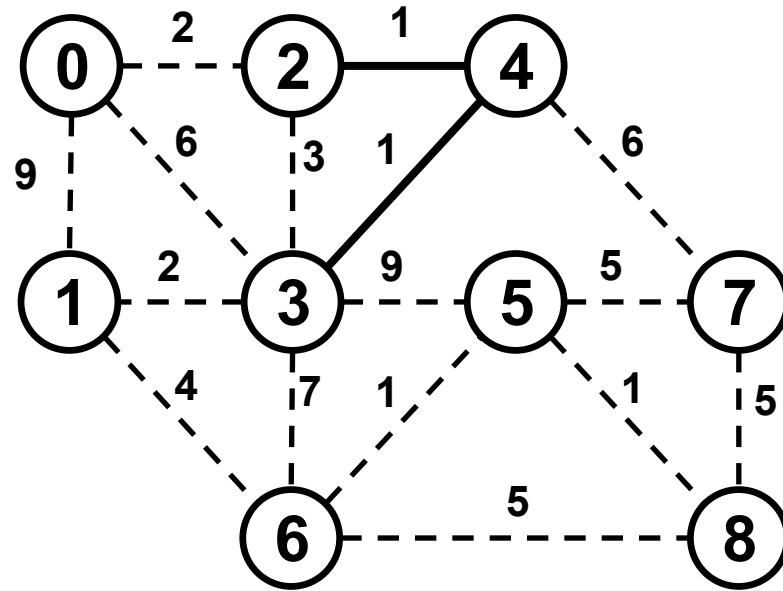
$$w(0, 3) = 6$$

$$w(4, 7) = 6$$

$$w(3, 6) = 7$$

$$w(0, 1) = 9$$

$$w(3, 5) = 9$$



## Алгоритм Крускала

$$w(2, 4) = 1$$

$$w(3, 4) = 1$$

$$w(5, 6) = 1$$

$$w(5, 8) = 1$$

$$w(0, 2) = 2$$

$$w(1, 3) = 2$$

$$w(2, 3) = 3$$

$$w(1, 6) = 4$$

$$w(5, 7) = 5$$

$$w(6, 8) = 5$$

$$w(7, 8) = 5$$

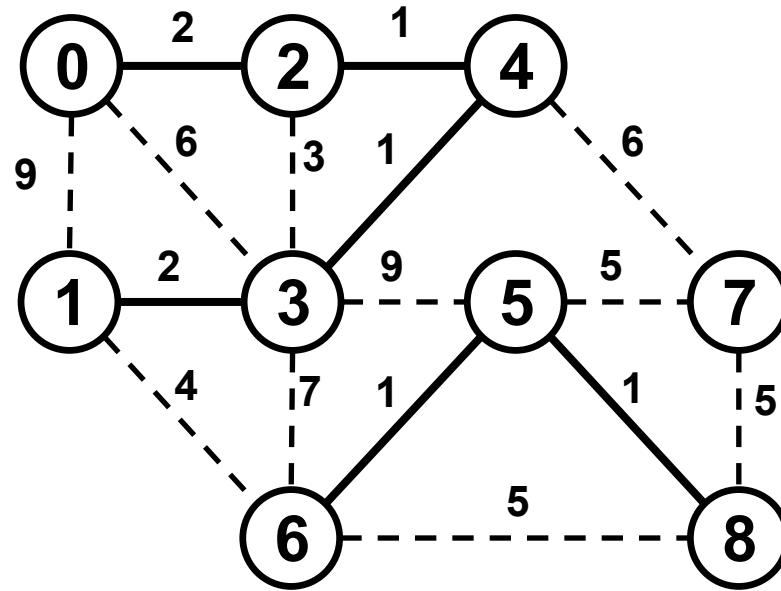
$$w(0, 3) = 6$$

$$w(4, 7) = 6$$

$$w(3, 6) = 7$$

$$w(0, 1) = 9$$

$$w(3, 5) = 9$$



## Алгоритм Крускала

$$w(2, 4) = 1$$

$$w(3, 4) = 1$$

$$w(5, 6) = 1$$

$$w(5, 8) = 1$$

$$w(0, 2) = 2$$

$$w(1, 3) = 2$$

$$w(2, 3) = 3$$

$$w(1, 6) = 4$$

$$w(5, 7) = 5$$

$$w(6, 8) = 5$$

$$w(7, 8) = 5$$

$$w(0, 3) = 6$$

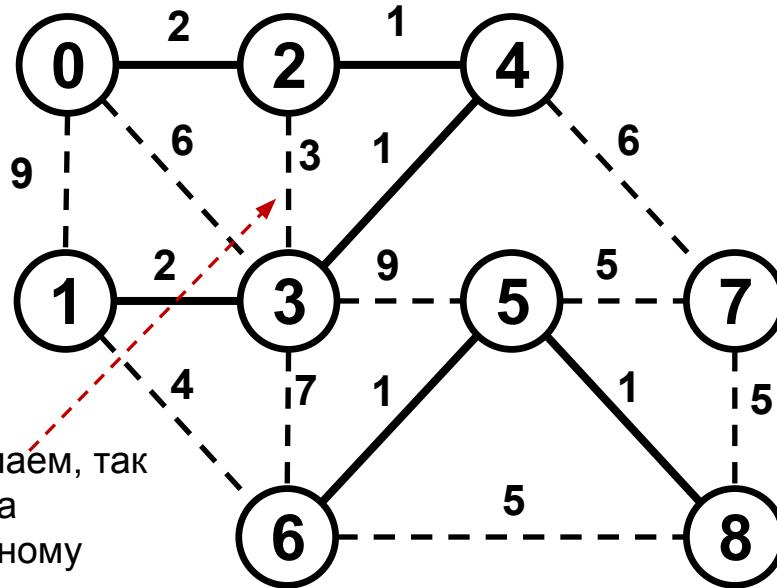
$$w(4, 7) = 6$$

$$w(3, 6) = 7$$

$$w(0, 1) = 9$$

$$w(3, 5) = 9$$

Ребро (2, 3) скипаем, так как оба его конца принадлежат одному дереву.



## Алгоритм Крускала

$$w(2, 4) = 1$$

$$w(3, 4) = 1$$

$$w(5, 6) = 1$$

$$w(5, 8) = 1$$

$$w(0, 2) = 2$$

$$w(1, 3) = 2$$

$$w(2, 3) = 3$$

$$w(1, 6) = 4$$

$$w(5, 7) = 5$$

$$w(6, 8) = 5$$

$$w(7, 8) = 5$$

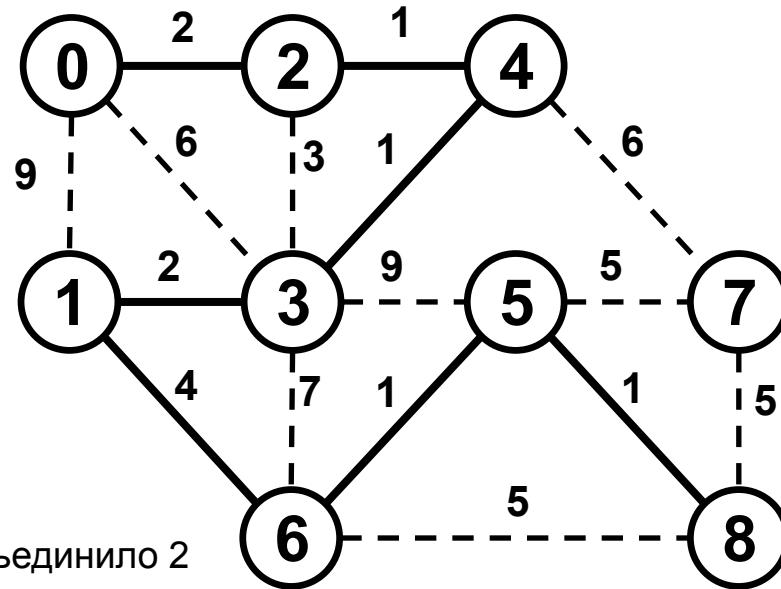
$$w(0, 3) = 6$$

$$w(4, 7) = 6$$

$$w(3, 6) = 7$$

$$w(0, 1) = 9$$

$$w(3, 5) = 9$$



Ребро (1, 6) объединило 2 больших поддерева.

## Алгоритм Крускала

$$w(2, 4) = 1$$

$$w(3, 4) = 1$$

$$w(5, 6) = 1$$

$$w(5, 8) = 1$$

$$w(0, 2) = 2$$

$$w(1, 3) = 2$$

$$w(2, 3) = 3$$

$$w(1, 6) = 4$$

$$w(5, 7) = 5$$

$$w(6, 8) = 5$$

$$w(7, 8) = 5$$

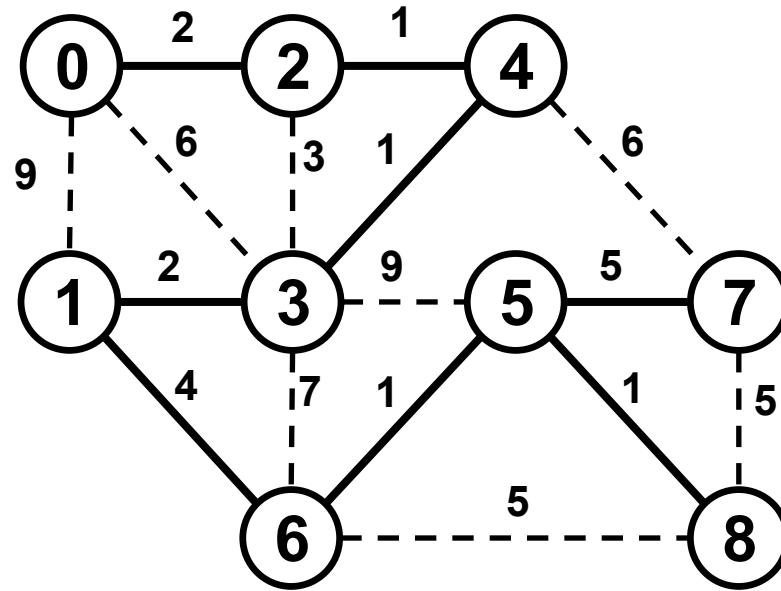
$$w(0, 3) = 6$$

$$w(4, 7) = 6$$

$$w(3, 6) = 7$$

$$w(0, 1) = 9$$

$$w(3, 5) = 9$$



## Алгоритм Крускала

$$w(2, 4) = 1$$

$$w(3, 4) = 1$$

$$w(5, 6) = 1$$

$$w(5, 8) = 1$$

$$w(0, 2) = 2$$

$$w(1, 3) = 2$$

$$w(2, 3) = 3$$

$$w(1, 6) = 4$$

$$w(5, 7) = 5$$

$$w(6, 8) = 5$$

$$w(7, 8) = 5$$

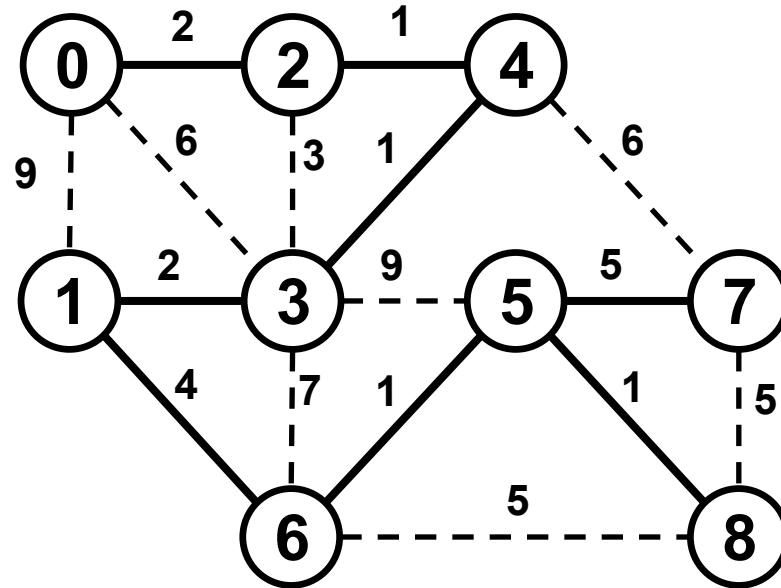
$$w(0, 3) = 6$$

$$w(4, 7) = 6$$

$$w(3, 6) = 7$$

$$w(0, 1) = 9$$

$$w(3, 5) = 9$$



Начиная с ребра (6, 8), у оставшихся рёбер  
концы принадлежат одному поддереву.  
Скипаем их.

## Алгоритм Крускала: оценка

Время работы алгоритма складывается из:

- Сортировка рёбер =  $O(E \log E)$
- Объединение поддеревьев  $O(V^2)$

Общее время работы =  $O(E \log E) + O(V^2)$

## Алгоритм Крускала: СНМ

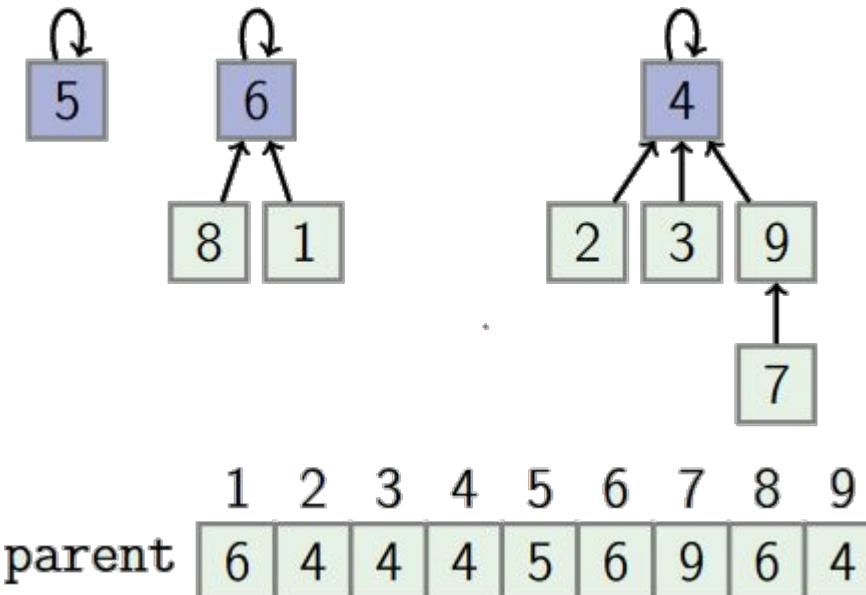
Требуется СД «Система Непересекающихся Множеств» (CHM=DSU) с операциями

1. `make_set(u)` – создать множество с одним элементом u.
2. `find_set(u)` – найти множество по элементу u, чтобы можно было сравнить их (`find_set(u) == find_set(v)`).
3. `union_set(u, v)` – объединить два множества, одно из которых содержит элемент u, а другое – элемент v.

# Алгоритм Крускала: СНМ

Особенности реализации:

- Храним элементы каждого множества в виде дерева. Корень дерева называется **представителем** или **лидером**. Он возвращается методом `find_set`.



## Алгоритм Крускала: СНМ

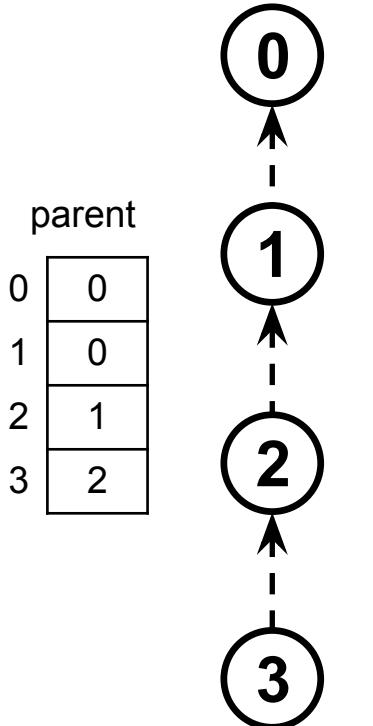
Особенности реализации:

- Сравнение двух деревьев – сравнение представителей.
- Объединение двух деревьев – одному представителю в качестве родителя устанавливается другой представитель.
- Родитель корня (представителя) – сам представитель.

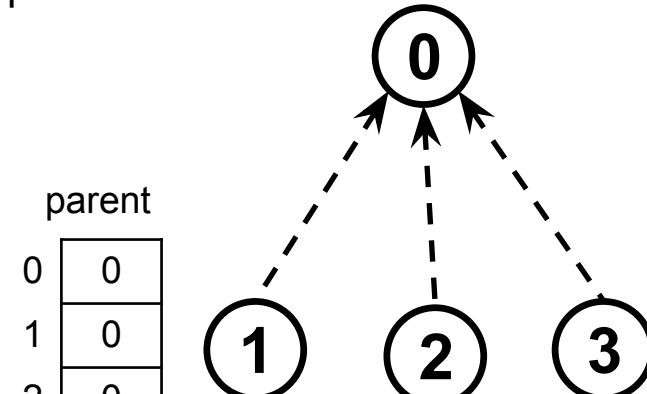


## Алгоритм Крускала: СНМ

При каждом новом поиске все элементы, находящиеся на пути от корня до искомого элемента, вешаются под корень дерева.



`find_set(3)`

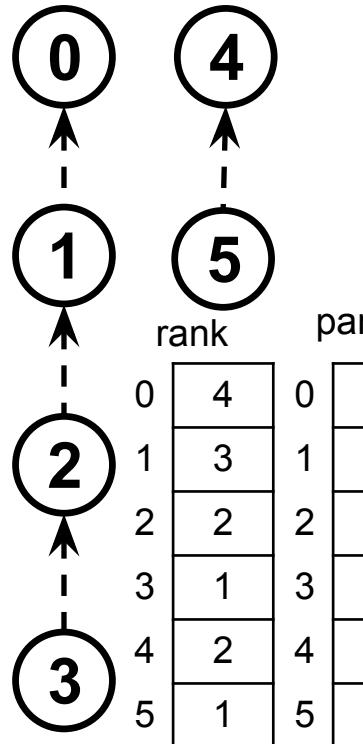


## СНМ: эвристика сжатия пути

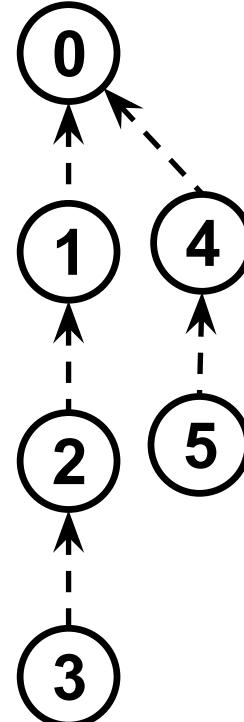
```
// Сжатие пути
int find_set (int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set (parent[v]);
}
```

# СНМ: эвристика объединения по рангу

Идея в том, что мы присоединяем дерево с меньшим рангом к дереву с большим рангом.



Union\_set(0, 4)



В качестве ранга можно брать количество элементов в подмножестве или максимальную глубину дерева.

## Алгоритм Крускала: СНМ, оценка

Время работы алгоритма складывается из:

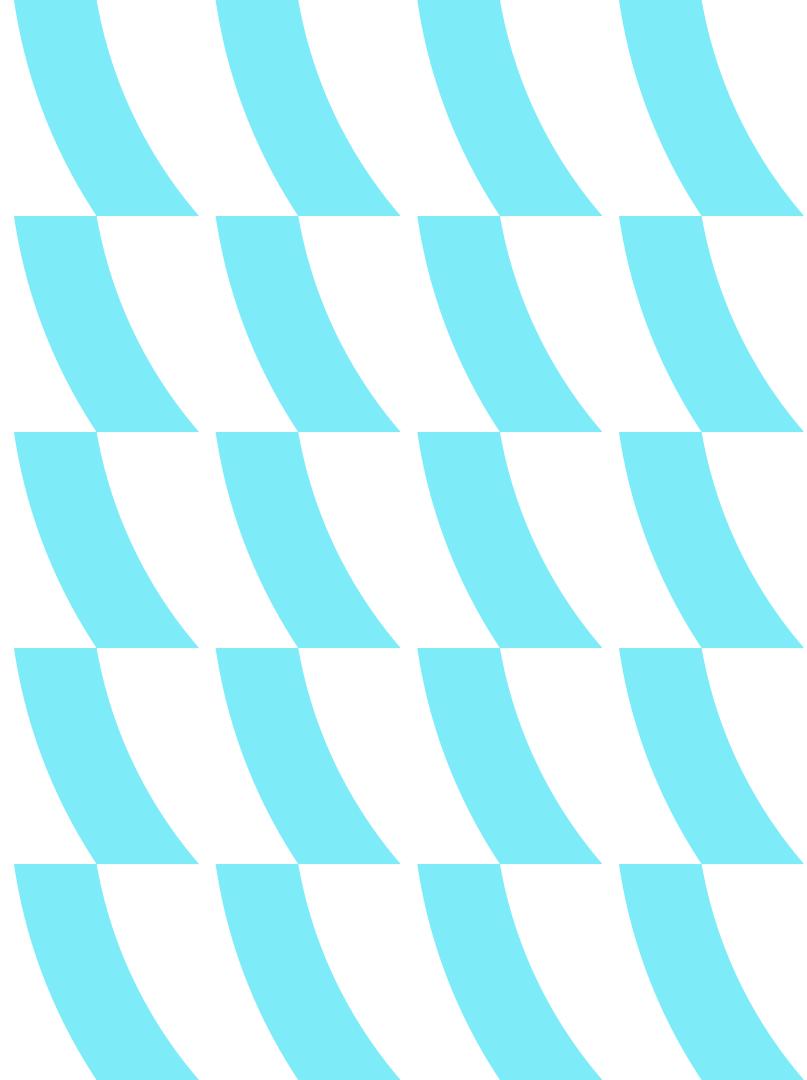
1. Сортировка рёбер =  $O(E \log E)$
2. Объединение поддеревьев с помощью СНМ и двух эвристик  $O(A^{-1}(V)) \approx O(1)$

Общее время работы =  $O(E \log E)$

$A^{-1}(V)$  - обратная функция Аккермана.

Это настолько медленно растущая функция, что для любых мыслимых применений ее значение не превышает 4 (примерно для  $n \leq 10^{600}$ ).

# **NP-полнота**



## NP-полнота

Классы сложности алгоритмов:

- Класс  $P$  – задачу можно решить за полиномиальное время от количества входных данных ( $O(n^k)$ , где  $n$  – кол-во входных данных,  $k$  - константа)
- Класс  $NP$  – задача поддается проверке за полиномиальное время
- Класс  $NP$  - полных задач – подмножество задач класса  $NP$ , к которой можно свести любую задачу из этого класса за полиномиальное время.

Точно понятно, что  $P \subseteq NP$

Проблема равенства классов  $P = NP$  – математическая проблема тысячелетия.

Большинство исследователей склоняются к тому, что  $P \neq NP$

# NP-полнота

Примеры NP-полных задач:

- Задача коммивояжёра
- Задача выполнимости булевых формул
- Кратчайшее решение пятнашек размером  $n \times n$
- Сапёр
- Тетрис

# Задача коммивояжера

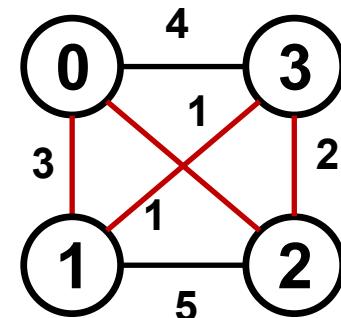
Пусть  $G$  – полный неориентированный взвешенный граф.

Коммивояжеру нужно совершить *кратчайший тур*, или гамильтонов цикл, посетив каждый город ровно по одному разу и завершив путешествие в том же городе, из которого он выехал.



**Метрическая задача коммивояжера** - на матрице стоимостей выполняется неравенство треугольника.

Задача коммивояжера - NP-полная, то есть существование быстрого алгоритма маловероятно.



# Приближенные алгоритмы

Многие задачи, представляющие практический интерес, являются NP-полными.

Варианты решения:

1. Если объем входных данных небольшой, то может подойти алгоритм, время работы которого выражается показательной функцией.
2. Иногда удается выделить важные частные случаи, разрешимые в течение полиномиального времени.
3. Иногда есть возможность найти за полиномиальное время решение, близкое к оптимальному.

Алгоритм, возвращающий решения, близкие к оптимальным, называется **приближенным алгоритмом**.

## Оценка качества приближенных алгоритмов

Говорят, что алгоритм обладает **коэффициентом аппроксимации**  $\rho(n)$ , если

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

$n$  – размер входных данных

$C$  – стоимость решения

$C^*$  – стоимость оптимального решения

**$\rho(n)$ -приближенный алгоритм** – алгоритм, в котором достигается коэффициент аппроксимации  $\rho(n)$

# Задача коммивояжера

## 2-приближенный алгоритм

- $G = (V, E)$  – полный неориентированный граф
- Каждому ребру  $(u, v) \in E$  ставим в соответствие целочисленную стоимость  $c(u, v) \geq 0$
- $c(u, v)$  удовлетворяют неравенству треугольника, то есть для всех  $u, v, w \in V$   
$$c(u, w) \leq c(u, v) + c(v, w)$$

Если неравенство треугольника не выполняется, то приближенного алгоритма с полиномиальным временем работы не существует, если только не справедливо соотношение  $P = NP$ .

## Задача коммивояжера

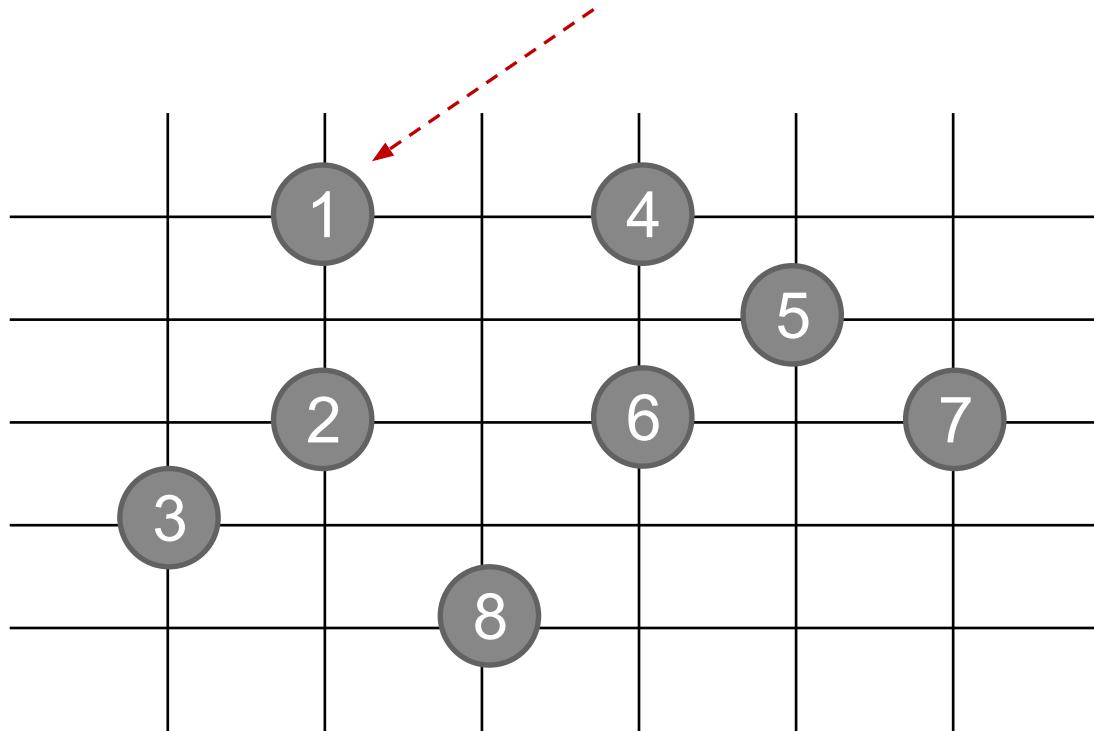
### 2-приближенный алгоритм Approx\_TSP\_Tour

1. Выбирается произвольная вершина, которая будет корневой
2. Из этой вершины строится минимальное остовное дерево  $T$  для графа  $G$
3. Обходим дерево  $T$  в прямом порядке, получаем список вершин  $L$  в порядке прямого обхода дерева  $T$ .
4. Получаем гамильтонов цикл  $H$ , который посещает вершины в порядке их перечисления в списке  $L$ .

# Задача коммивояжера

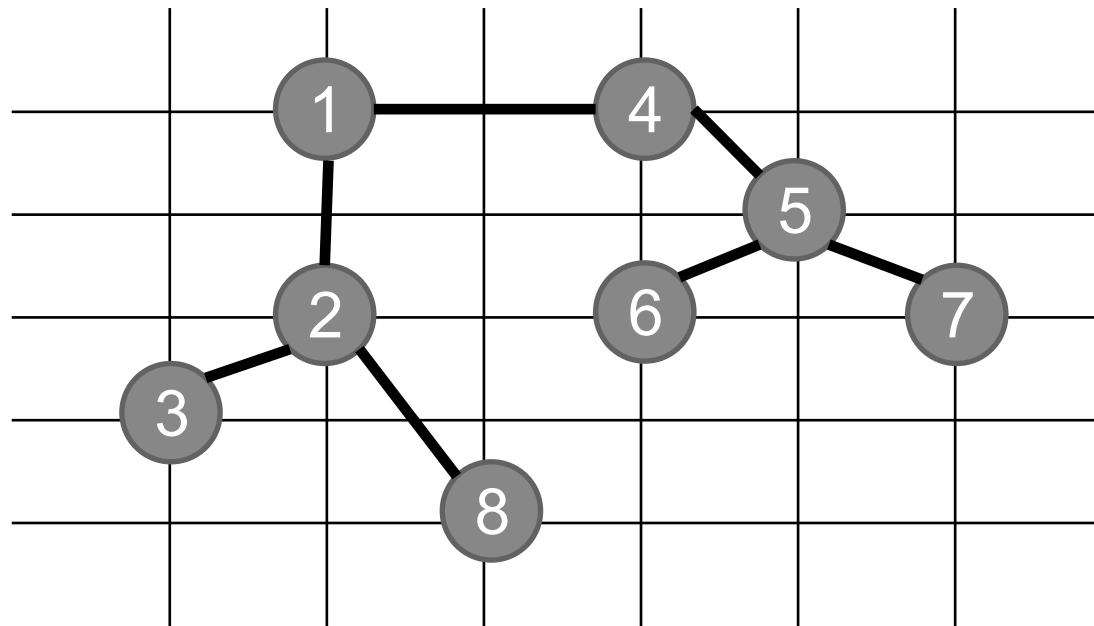
Исходный граф

Возьмем эту вершину в  
качестве исходной



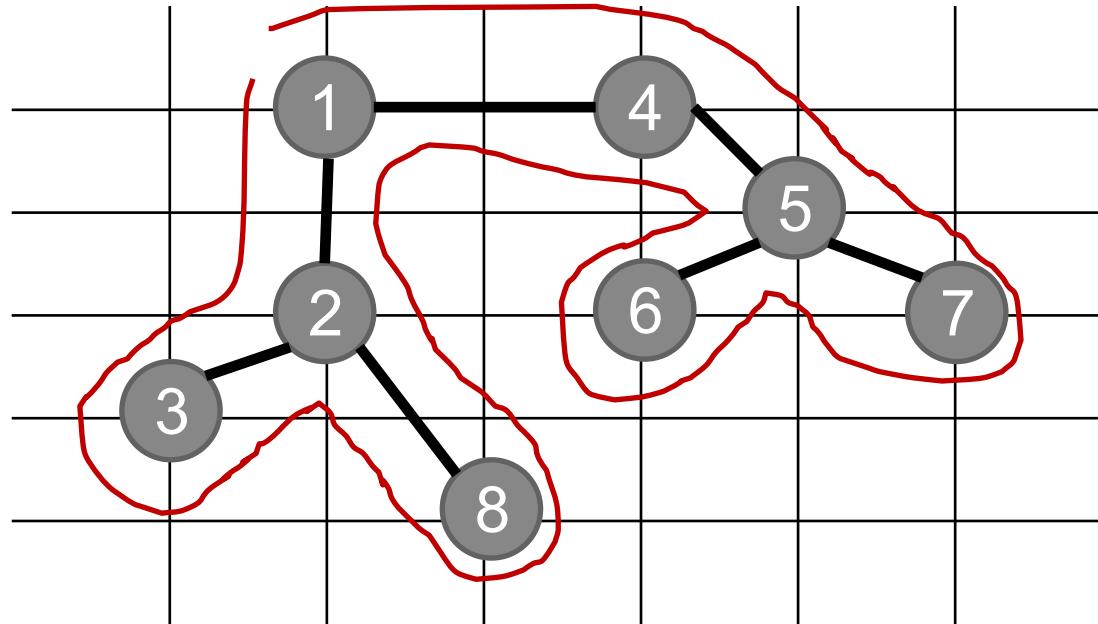
# Задача коммивояжера

Строим минимальное остовное дерево



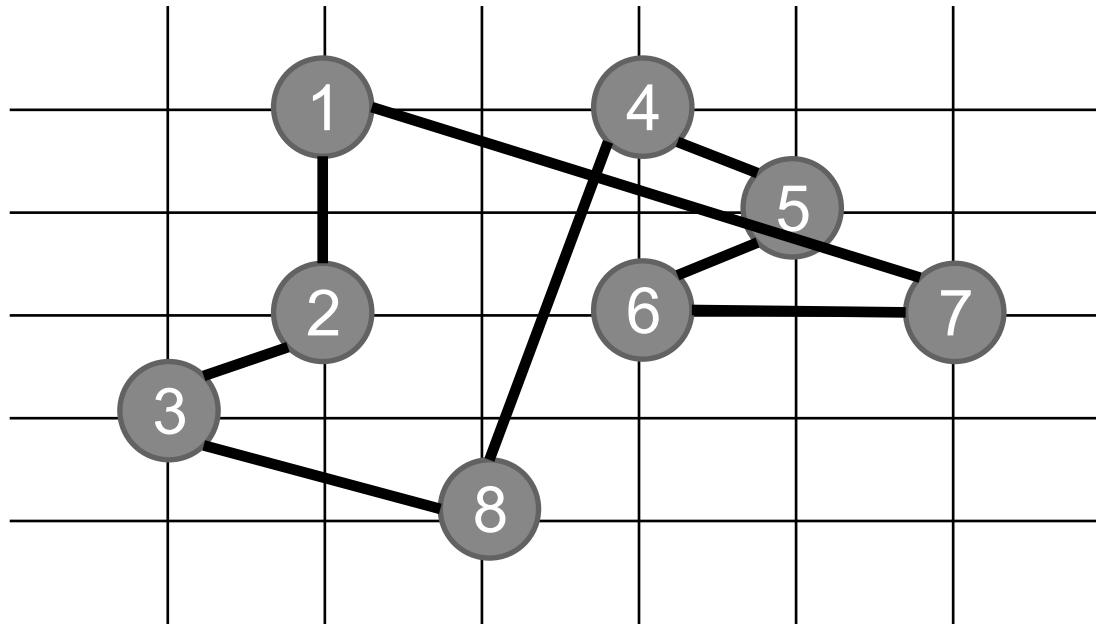
## Задача коммивояжера

Обходим минимальное минимальное оставное дерево в  
прямом порядке, получаем список вершин  $L$



## Задача коммивояжера

Получаем гамильтонов цикл  $H$ , который посещает вершины в порядке их перечисления в списке  $L$ .

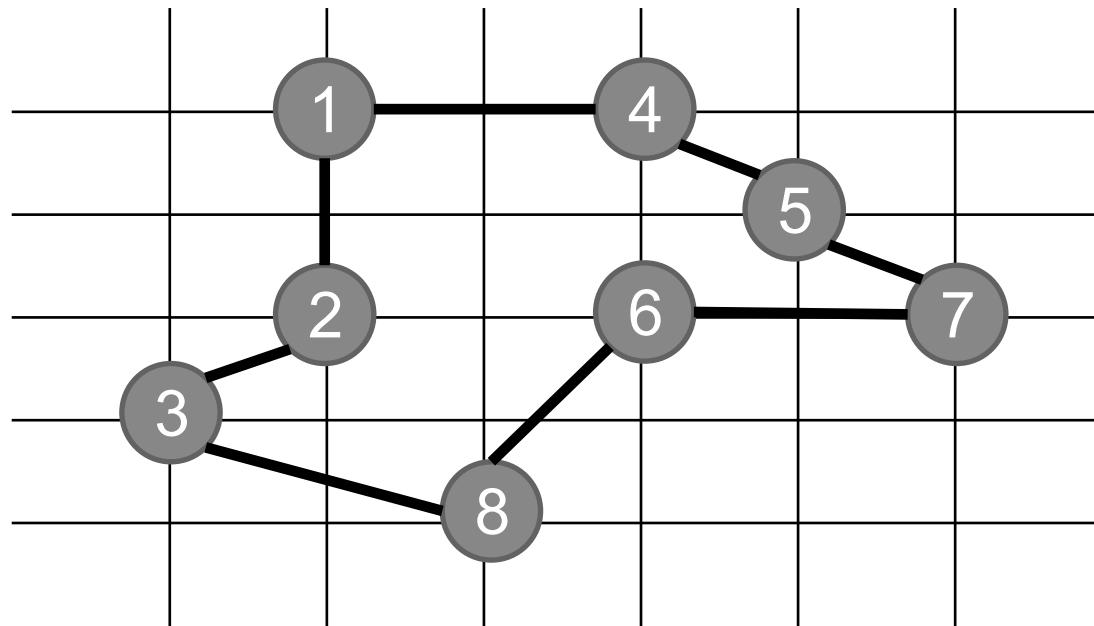


# Задача коммивояжера

Оптимальный тур  $H^*$

Длина тура  $H \sim 19.074$

Длина оптимального тура  $H^* \sim 14.715$



## Задача коммивояжера

Теорема. Алгоритм Approx\_TSP\_Tour является 2-приближенным алгоритмом с полиномиальным временем работы, позволяющим решить задачу коммивояжера, в которой удовлетворяется неравенство треугольника.

Док-во.

$H^*$  - оптимальный тур. Удалив из него одно ребро получим оставное дерево, причем вес минимального оставного дерева  $T$  является нижней границей стоимости оптимального тура, то есть  $c(T) \leq c(H^*)$ .

При полном обходе минимального оставного дерева все ребра посещаются 2 раза, обозначим этот обход через  $W$ . Из этого следует, что  $c(W) = 2c(T) \Rightarrow c(W) \leq 2c(H^*)$ , то есть стоимость обхода  $W$  не более чем в 2 раза больше стоимости оптимального тура.

## Задача коммивояжера

Док-во (продолжение).

Но  $W$  – не тур, он посещает некоторые вершины более 1 раза. Но исходя из неравенства треугольника можно исключить из обхода все посещения вершины, кроме первой, и при этом стоимость пути не возрастет. То есть мы получим цикл  $H$ , он является гамильтоновым, так как вершины посещаются по одному разу. Так как  $H$  получается путем удаления вершин из  $W$ , то

$$c(H) \leq c(W) \Rightarrow c(H) \leq 2c(H^*)$$

## Домашнее задание #08

- Реализация алгоритма APSP (All Pairs Shortest Paths problem):
  - алгоритм Джонсона
  - алгоритм Форда — Фалкерсона
  - алгоритм Флойда — Уоршелла

# Напоминание отметиться на портале Vol 2

+ оставить отзыв после лекции

The screenshot shows the VK Education website interface. At the top, there's a navigation bar with links like 'Блоги' (Blogs), 'Люди' (People), 'Программа' (Program), 'Вакансии' (Jobs), and 'Расписание' (Schedule). A yellow banner at the top right says 'Открыт приём заявок!' (Applications are open!). Below the banner, a weekly schedule is displayed:

Чт, 8 сентября	пт, 9 сентября	сб, 10 сентября	вс, 11 сентября	пон, 12 сентября
Нет занятий	18:00 Углублённый Py... Введение в Python, основные понятия, тестирование Г. Кандауров	Нет занятий	Нет занятий	Нет занятий

Below the schedule, a blog post titled 'Углубленный Python' is shown. It has 57 readers and 2 topics. There are buttons for 'Подписаться' (Subscribe) and 'Создать топик' (Create topic). A search bar and a 'Найти' (Find) button are also present.

On the right side, there's a sidebar for 'Прямой эфир' (Live Stream) with a list of recent comments from users like Геннадий Кандауров, Екатерина Черкасова, and Дарья Вовченко.

# Спасибо за внимание



education