Алгоритмы и структуры данных

Лекция 15
Графы. MST. Кратчайшие пути.

Сергей Леонидович Бабичев

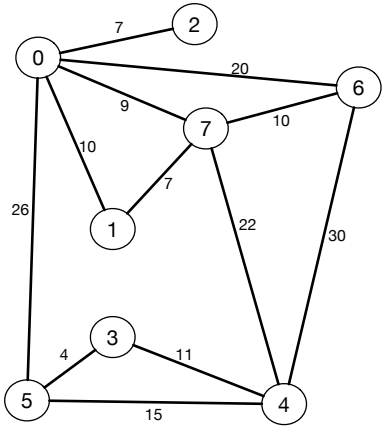
Алгоритм Краскала

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

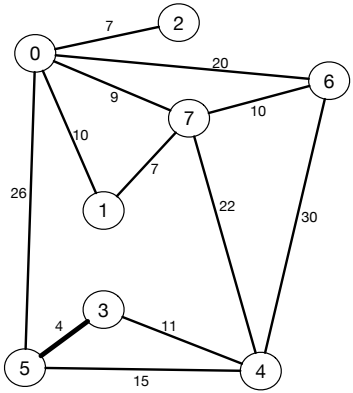
Алгоритм Краскала

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

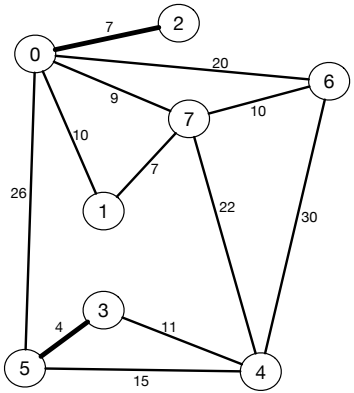
Два подходящих ребра с одинаковым весом:

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

Алгоритм Краскала:вторая итерация

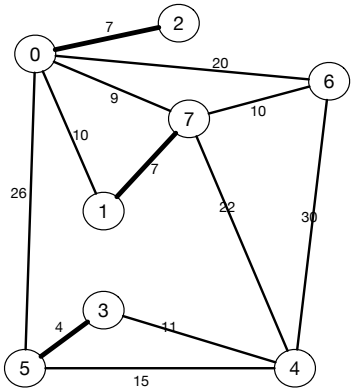
Выберем ребро (0,2) и поместим вершину 2 в множество номер 0.



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

Алгоритм Краскала:третья итерация

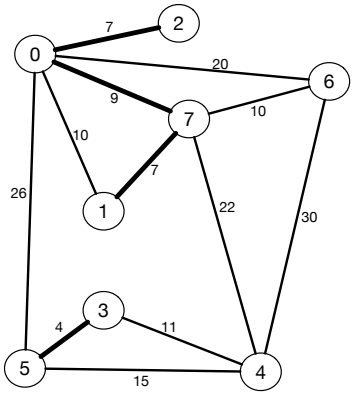
Ребро (1,7) привело к слиянию множеств 1 и 7.



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: `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

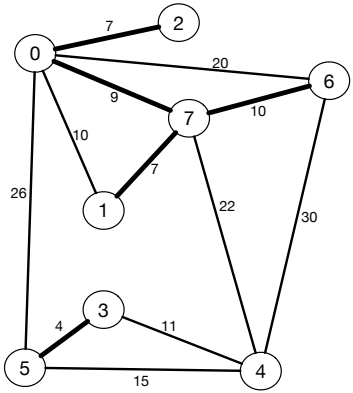
- Определяем, каким деревьям принадлежат концы ребра `(0,7)`.
- `find_root(0)` вернёт 0 как номер множества.
- `find_root(7)` сначала убедится, что в `p[7]` лежит 1 и вызовет `find_root(1)`, после чего, возможно, заменит `p[7]` на 1 и вернёт 1.

Алгоритм Краскала

- Концы ребра 0 принадлежат разным множествам $\rightarrow$ 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

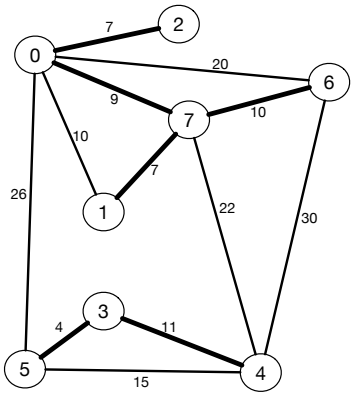


Следующее ребро — (6,7). `find_root(7)` установит `p[7]=0`. Это же значение будет присвоено и `p[6]`.

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

Алгоритм Краскала

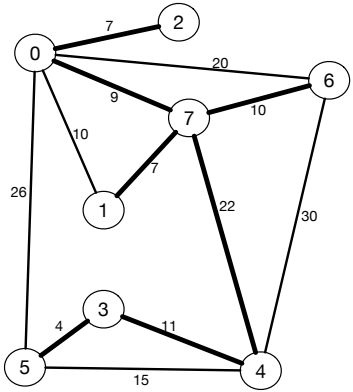
На следующем этапе ребро (3,4) окажется самым коротким.



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 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 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.
- **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 ∈ V do
3:          d[v] ← ∞
4:          π[v] ← nil
5:      end for
6:      d[s] ← 0
7:      U ← ∅
8:      Q ← V
9:      while Q ≠ ∅ do
10:         u ← Q.extractMin()
11:         U ← U ∪ {u}
12:         for all v ∈ Adj[u], v ∉ U do
13:             Relax(u,v)
14:         end for
15:     end while
16: end procedure
```

Алгоритм Дейкстры

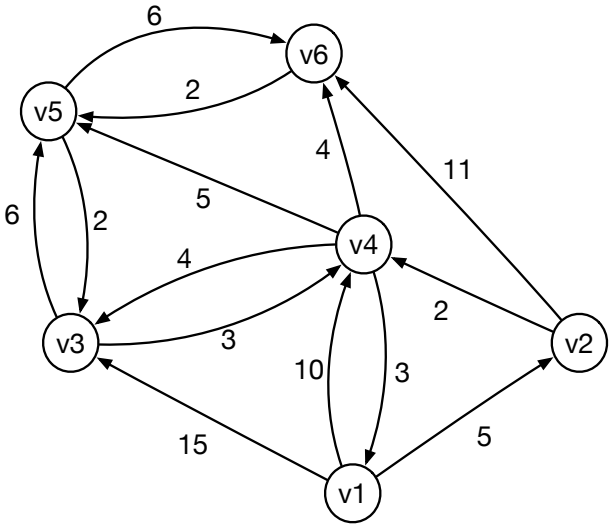
1: **procedure** $\textsc{Relax}(u, v : Vertex)$
2: **if** $d[v] > d[u] + w(u, v)$ **then**
3: $d[v] = d[u] + w(u, v)$
4: $\pi[v] \leftarrow u$
5: **end if**
6: **end procedure**

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.

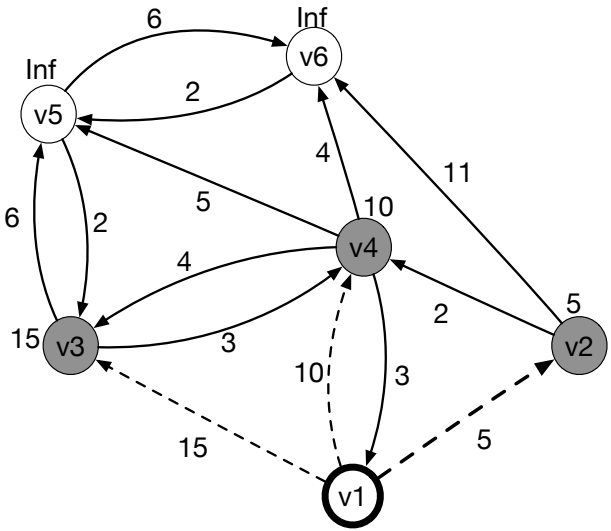
Алгоритм Дейкстры

Исходный граф



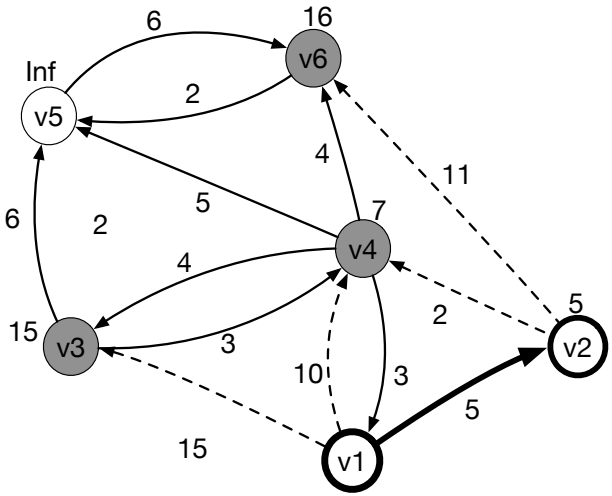
Алгоритм Дейкстры

v1 в SPT, v2, v3 и v4 — в накопителе.



Алгоритм Дейкстры

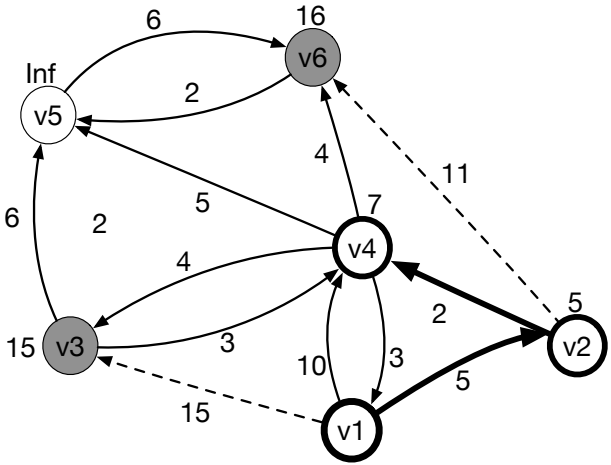
Выбран узел v2. Корректируются расстояния от него. Релаксация: $(1 \to 4)$



заменён на $(1 \to 2 \to 4)$.

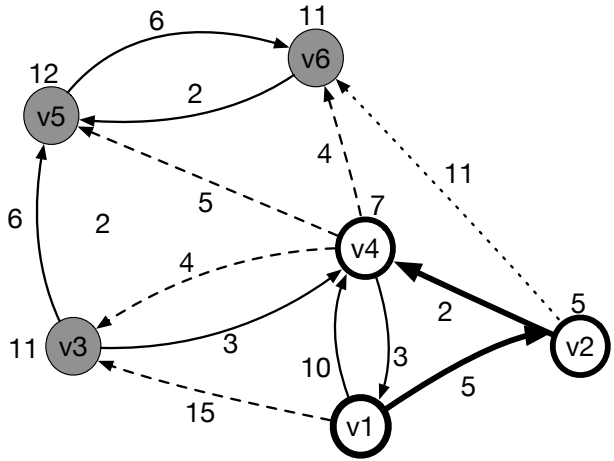
Алгоритм Дейкстры

Выбран узел v3.

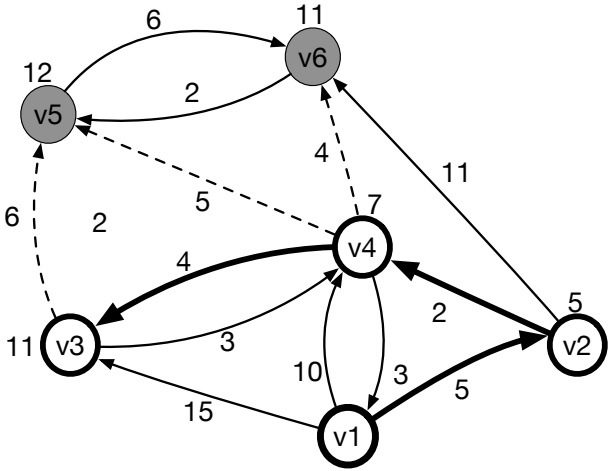


Алгоритм Дейкстры

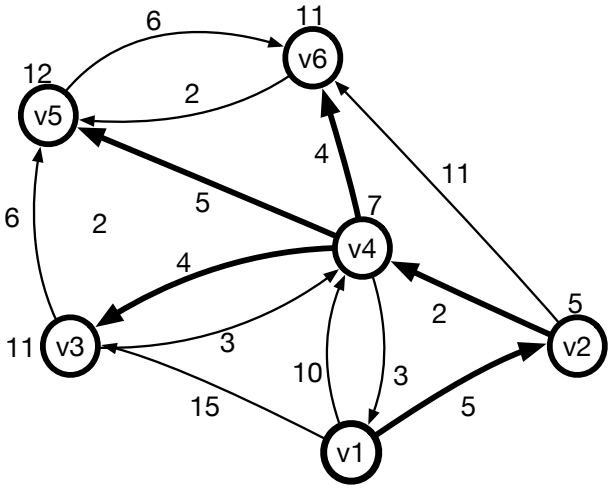
В накопитель отправляется v5. Релаксация: $(1 \to 2 \to 6)$ заменено на $(1 \to 2 \to 4 \to 6)$, $(1 \to 3)$ на $(1 \to 2 \to 4 \to 3)$



Алгоритм Дейкстры



Алгоритм Дейкстры



Алгоритм Дейкстры: сложность

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

Множественный алгоритм Дейкстры

- Если мы хотим построить таблицу минимальных расстояний от каждого до каждого, то вычисление таблиц для каждого узла в отдельности имеет сложность $O(N^2 \log N)$.
- Вычисление таблиц для всех узлов имеет сложность $N \cdot O(N^2 \log N) = O(N^3 \log N)$
- 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.

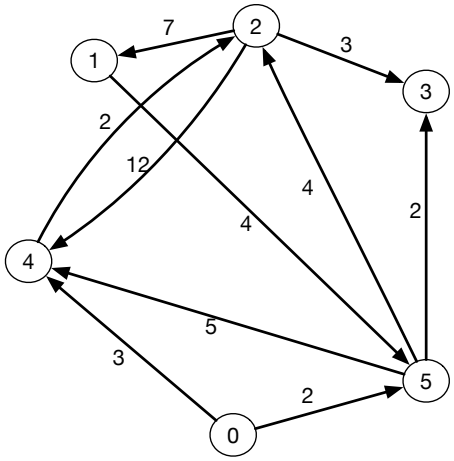
Алгоритм Флойда-Уоршалла

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 смежности:

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

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

Алгоритм Флойда-Уоршалла

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

$D^{(2)} =$

	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	∞	∞	4	2	5	0

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

Результат четвёртой и пятой итерации совпадает с результатом третьей. Шестая, последняя итерация:

$D^{(6)} =$

	P_0	P_1	P_2	P_3	P_4	P_5
P_0	0	**13**	**6**	**4**	3	2
P_1	∞	0	**8**	**6**	**9**	4
P_2	∞	7	0	3	12	11
P_3	∞	∞	∞	0	∞	∞
P_4	∞	∞	∞	∞	0	∞
P_5	∞	11	4	2	5	0