Топологическая сортировка, пути в графе

Лекция 13

Топологическая сортировка

Определение. *Частичным порядком* на множестве A называется отношение R, определенное на A и такое, что

- R транзитивно,
- для всех $a \in A$ утверждение aRa ложно, т.е. отношение R иррефлексивно.

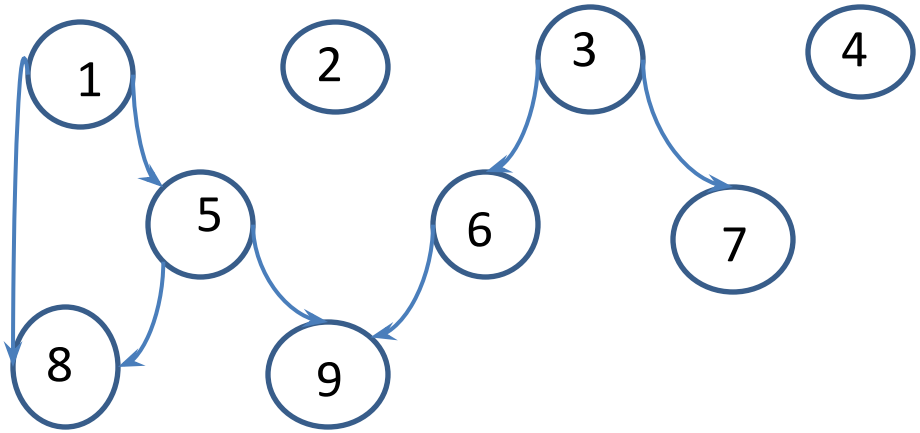
Из свойств (1) и (2) следует, что если aRb истинно, то bRa ложно (асимметричность).

Примеры частичного порядка:

- решение большой задачи разбивается на ряд подзадач, над которыми установлен частичный порядок: без решения одной задачи нельзя решить несколько других;
- последовательность чтения курсов в учебных программах: один курс основывается на другом;
- выполнение работ: одну работу следует выполнить раньше другой.

Если R — частичный порядок на множестве A, то (A, R) — ациклический граф.

Если (A, R') — ациклический граф и R' — отношение "являться потомком", определенное на A, то R' — частичный порядок на A.



Определение. *Линейный порядок R* на множестве *A* — это такой частичный порядок, что если *a* и *b* принадлежат *A*, то либо *aRb*, либо *bRa*, либо *a = b*.

Если *A* — конечное множество, то линейный порядок *R* удобно представлять , считая все элементы множества *A* расположенными в виде последовательности

$$a_1, a_2, \ldots, a_n,$$

для которой имеет место $a_i R a_j$ тогда и только тогда, когда $i < j$.

Если задан частичный порядок R на множестве A, часто бывает нужен линейный порядок, содержащий этот частичный порядок.

Эта проблема вложения частичного порядка в линейный называется *топологической сортировкой*.

Формально можно сказать, что

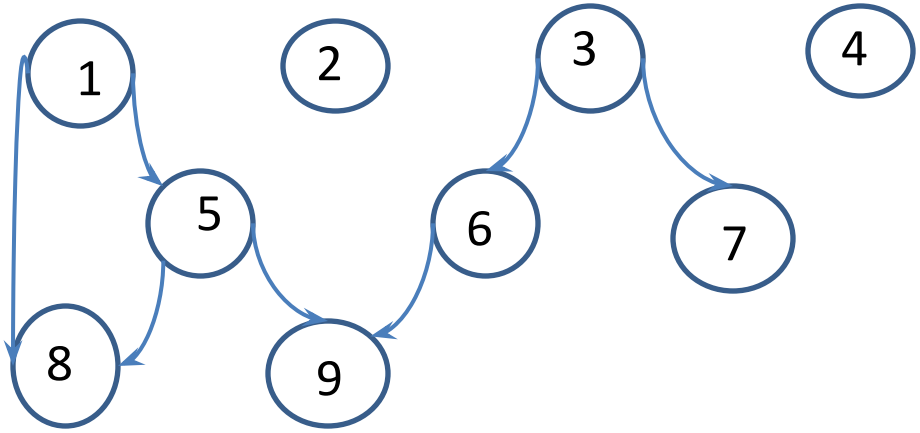
частичный порядок R на множестве A

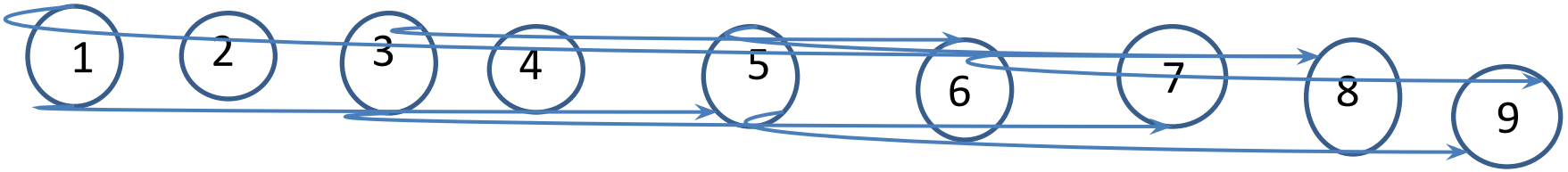
вложен в линейный порядок R',

если R' — линейный порядок и $R \subseteq R'$,

т. е. aRb влечет $aR'b$ для всех a и b из A.

Топологическая сортировка. Пример





Алгоритм. Топологическая сортировка

Вход. Частичный порядок R на конечном множестве A.

Выход. Линейный порядок R' на A, *для* которого $R \subseteq R'$.

Метод. Так как A — конечное множество, линейный порядок R' на A можно представить в виде списка $O_n = a_1, a_2, ..., a_n$, для которого $a_i \, R' \, a_j$, если $i < j$, и $A = \{a_1, a_2, ..., a_n\}$.

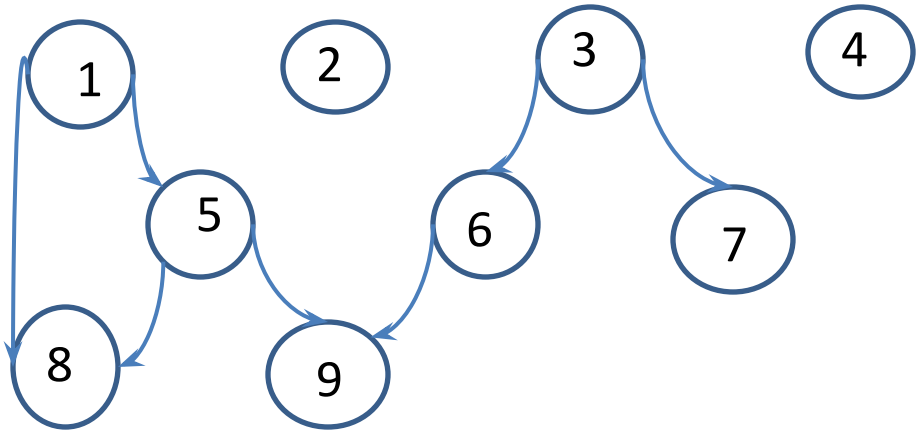
Эта последовательность элементов строится с помощью следующих шагов:

(1) Положить i=1, A_i=A и R_i=R.

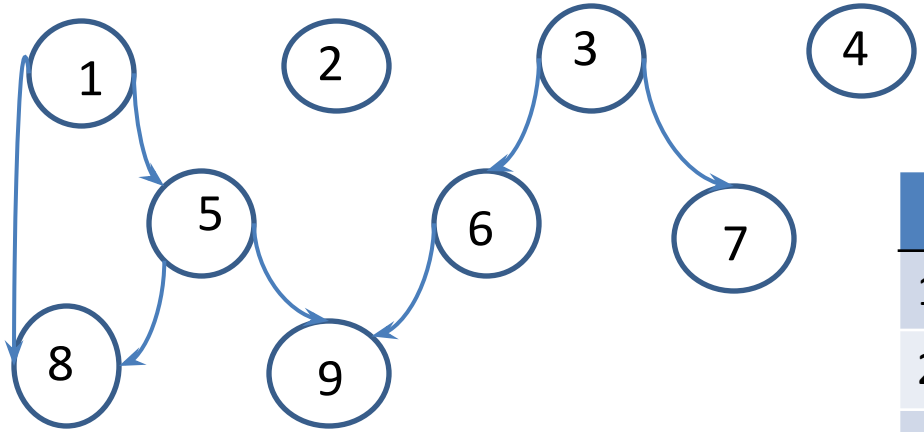
- Если A_i пусто, остановиться и выдать $O_i = a_1, ..., a_i$, в качестве искомого линейного порядка. В противном случае выбрать в A_i такой элемент a_{i+1} что $a' \, R \, a_{i+1}$, ложно для всех $a' \in A_i$.

- Положить $A_{i+1} = A_i \setminus \{a_{i+1}\}$ и $R_{i+1} = R_i \setminus (\{a_{i+1}\} \times A_{i+1})$. Затем увеличить i на единицу и повторить шаг 2.

Топологическая сортировка. Пример



Топологическая сортировка. Реализация на матрице смежности



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

1. Найти вершину, в которую не входит ни одна дуга (это нулевой столбец).
   Удалить все выходящие из нее дуги (обнулить соответствующую строку)
2. Пока не перебрали все вершины, повторять шаг 1.

Топологическая сортировка. Реализация на иерархических списках

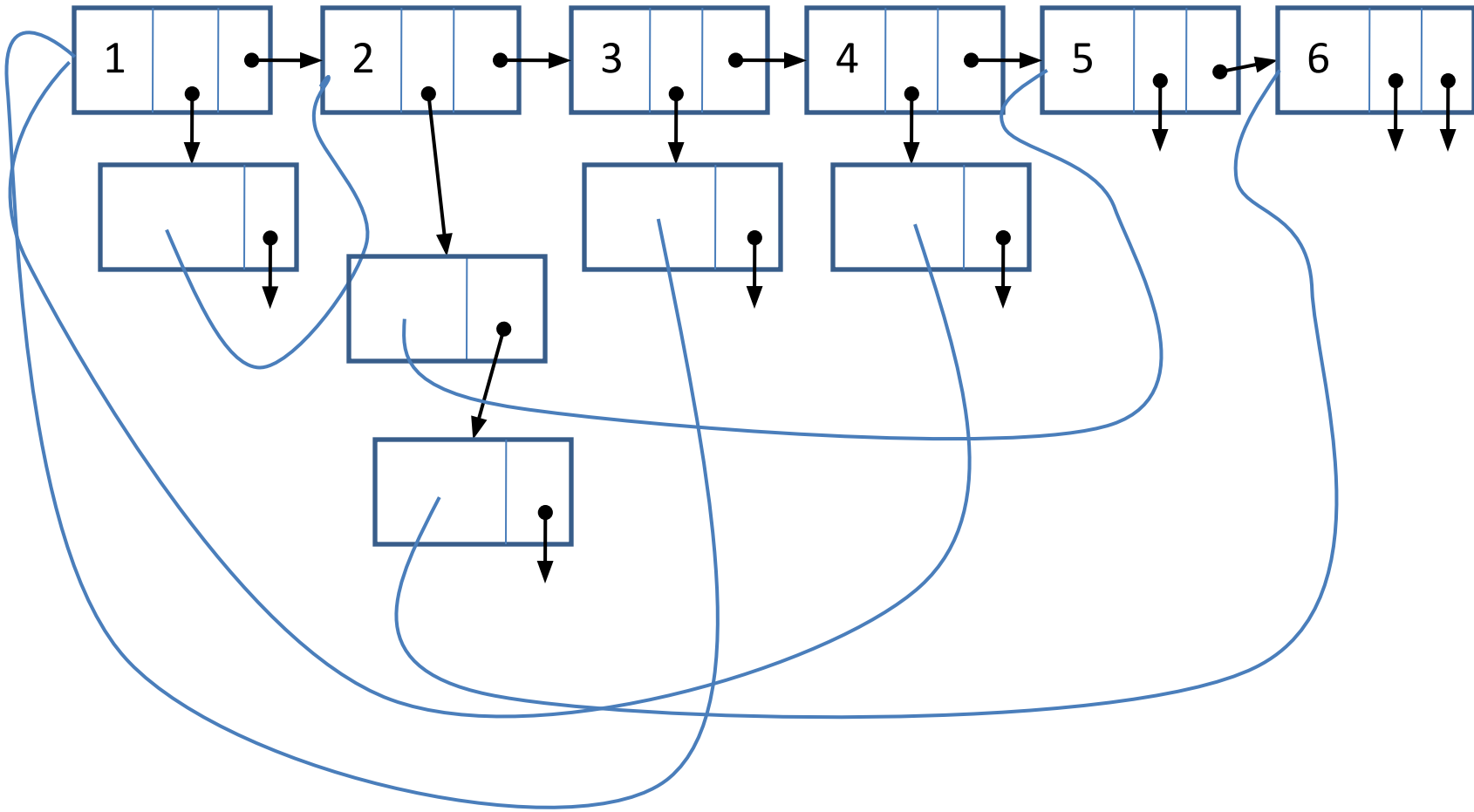
1< 2;

3< 1;

4<1;

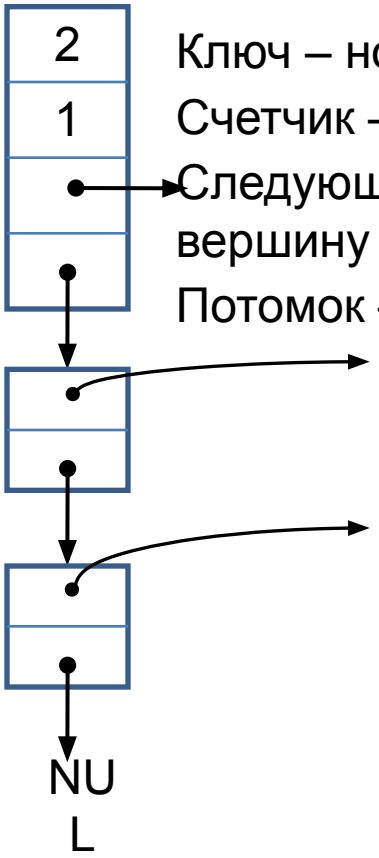
2< 5;

2< 6;



Топологическая сортировка. Реализация на иерархических списках

Элемент списка вершин графа:



Ключ – номер вершины

Счетчик – количество входящих дуг

Следующий – указатель на следующую вершину

Потомок – список указателей на потомков

Работа алгоритма(построение)

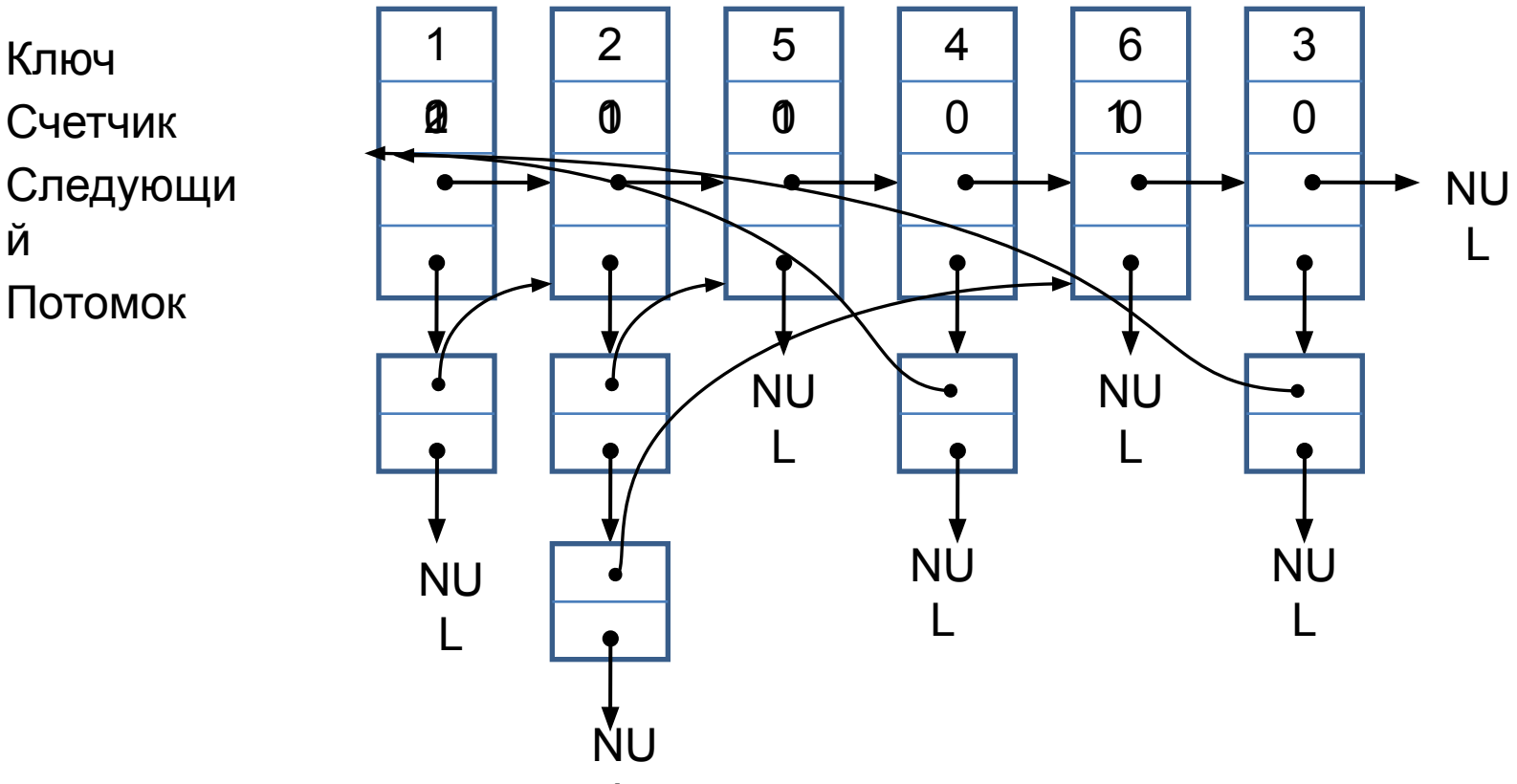
1< 2;

2< 5;

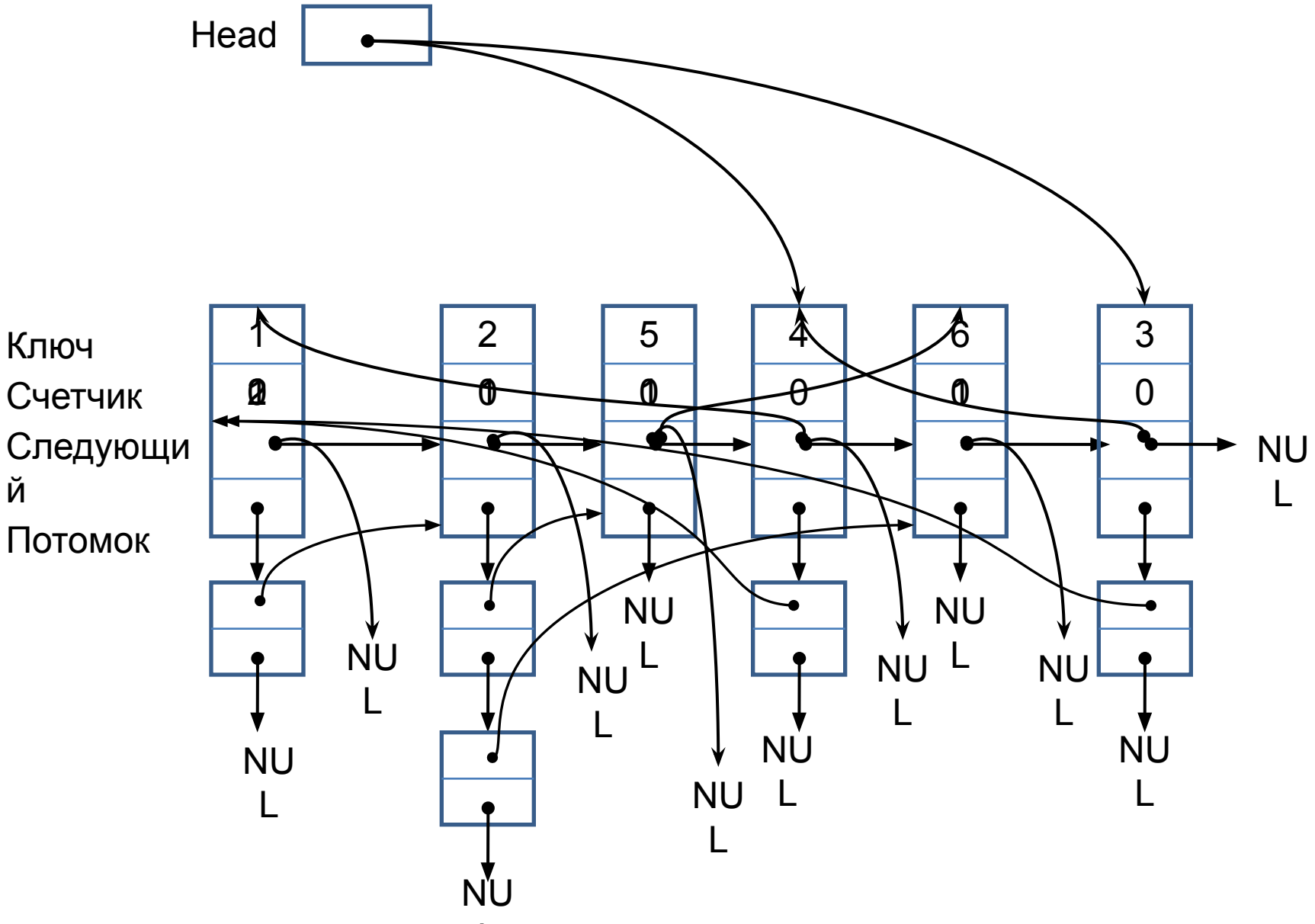
4 < 1;

2 < 6;

3 < 1;



Работа алгоритма (перестройка списка)



Кратчайшие пути

Пусть $G = (V, E)$ – ориентированный граф. Поставим в оответствие

каждому ребру $e \in E$ в графе G неотрицательную стоимость $w(e)$.

$$w: E \longrightarrow R^{+} \text{ - } \textit{функция стоимости}$$

Стоимость (вес) пути $p(v_0, v_1, \ldots, v_k)$ определяется как сумма стоимостей ребер, входящих в этот путь: $w(p) = \sum_i w(v_{i-1}, v_i)$.

Вес кратчайшего пути из u в v равен по определению

$$\delta(u,v) = \begin{cases} \min \{ w(p): \; u \rightarrow^{p} v \}, & \text{если существует путь из } u \text{ в } v \\ \infty, & \text{иначе} \end{cases}$$

Кратчайший путь из u в v это любой путь из u, для которого

$$w(p) = \delta(u,v)$$

Ребра отрицательного веса

Веса ребер могут ребер могут быть отрицательными.
Важно знать, имеются ли циклы отрицательного веса. Если
из вершины *s* можно добраться до цикла отрицательного
веса, то можно обойти этот цикл сколь угодно раз, при этом
вес все время будет уменьшаться.
Для вершин этого цикла кратчайших путей не существует.

Если циклов отрицательного веса нет, то любой цикл можно
выбросить. Путей без циклов конечное число, так что вес
кратчайшего пути определен корректно.

Пусть $G = (V, E)$ – заданный граф.

Для каждой вершины $v \in V$ мы будем помнить ее предшественника.

Релаксация – постепенное уточнение верхней оценки на вес кратчайшего пути в заданную вершину.

Свойства оптимальности.

Лемма 1. Отрезки кратчайших путей являются кратчайшими: Если $p(v_1, v_2, \ldots, v_k)$ – кратчайший путь из v_1 в v_k и $1 \le i \le j \le k$, то $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ есть кратчайший путь из v_i в v_j

Следствие 1. Рассмотрим кратчайший путь p из s в v. Пусть (u,v) – последнее ребро этого пути. Тогда $\delta(s,v) = \delta(s,u) + w(u,v)$.

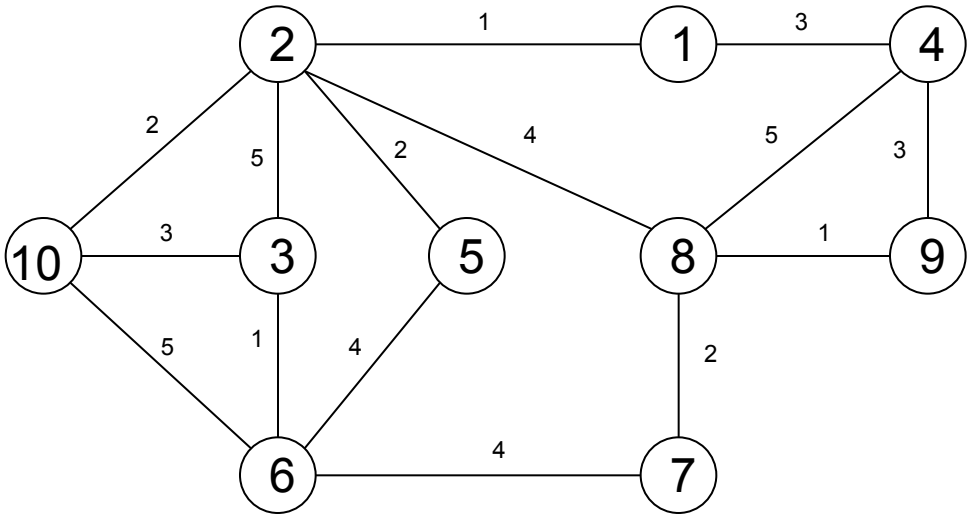
Следствие 2. Для любого ребра $(u,v) \in E$ справедливо $\delta(s,v) \le \delta(s,u) + w(u,v)$.

Идея алгоритма нахождения кратчайших путей

из одной вершины во все другие

Строится множество *S*, содержащее вершины графа, кратчайшие расстояния до которых от источника известны.

На каждом шаге добавляется тот из оставшихся узлов, кратчайшее расстояние до которого меньше всех других оставшихся узлов.



Кратчайшие пути из вершины 10:

v	Длина	Путь через
1	3	2
2	2	
3	4	
4	6	2, 1
5	4	2
6	4	3
7	8	2, 8 или 3, 6
8	6	2
9	7	2, 8

Техника релаксации

Для каждого ребра (u,v) храним $d[v]$ – верхнюю оценку кратчайшего пути из s в v.

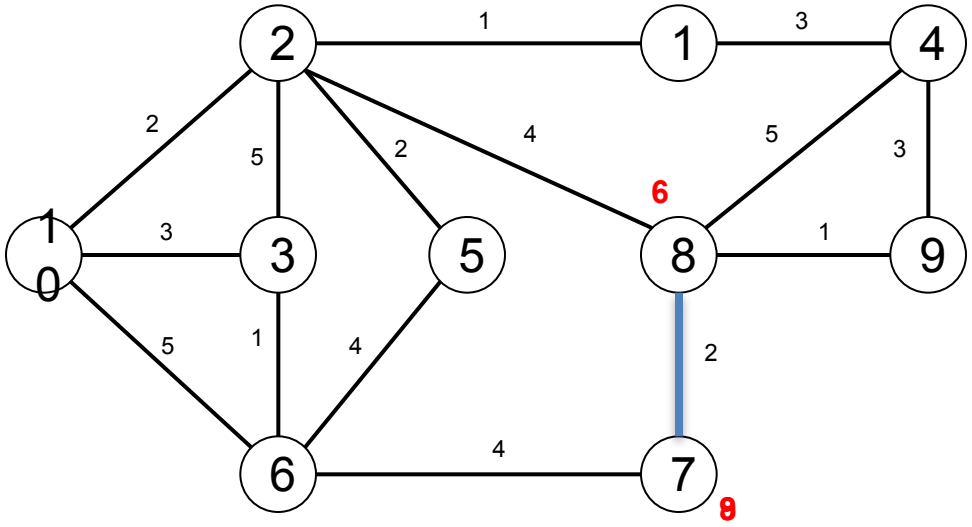
```
Initialize (G,s){
      for (для ∀v ∈ V) {
          d[v] ← ∞
             П[v] ← NULL;
     }
     d[s] ← 0;
}
```

Релаксация ребра (*u*,*v*):

значение *d*[*v*] уменьшается до *d*[*v+w(u,v)*]
(если второе значение меньше первого)

```
Relax (u, v, w) {
  If (d[v] >  d[u] + w(u,v)){
   d[v] = d[ u] + w(u,v);
   П[v] ← u;
   }
}
```

Релаксация ребра при поиске кратчайших путей.



Пусть уже найдены оценки кратчайших путей для вершин, соединенных красным ребром.

```
d[8] = 6;
d[7] = 9
```

Релаксация ребра (u, v):

```
if (d[u] + w(u,v) < d[v]) d[v] = d[u] +
w(u,v);
```

Релаксация ребра (7, 8):

```
9 + 2 > 6
```

Релаксация ребра (8, 7):

```
6 + 2 < 9   ⇒   d[7] = 8
```

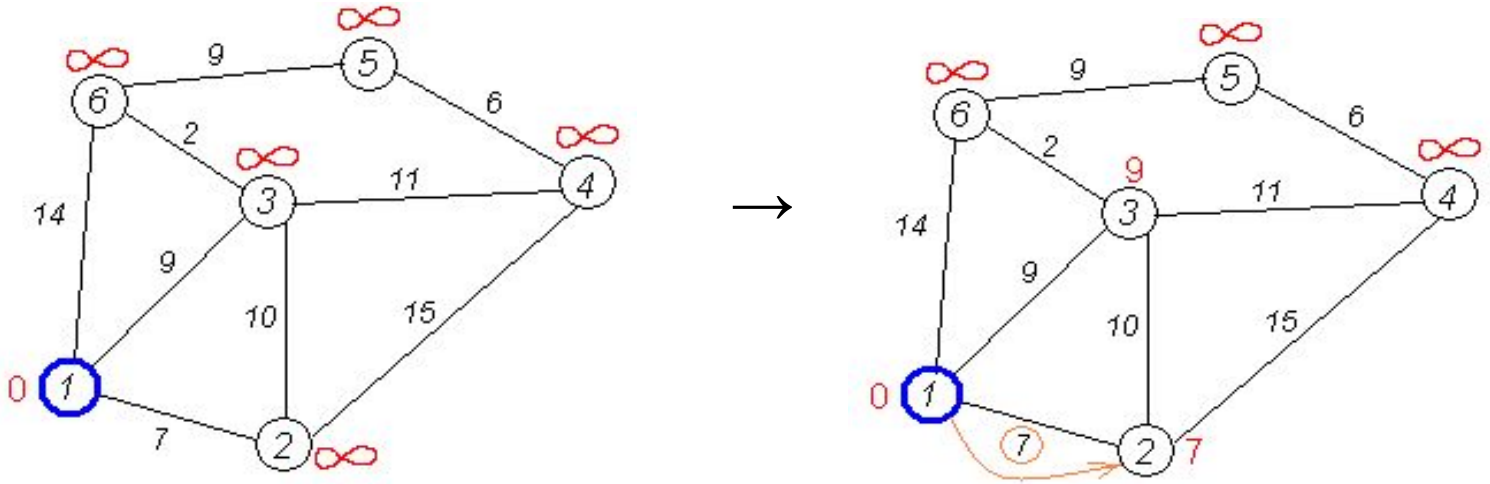
В ходе работы алгоритма поддерживается множество S, состоящее из вершин, для которых $\delta(s,v)$ уже найдено (т.е. $d[s,v] = \delta(s,v)$).

1. Выбираем вершину $u \in V \setminus S$ с наименьшим $d[u]$;
2. Добавляем вершину u к множеству S;
3. Выполняем релаксацию для всех инцидентных u ребер;
4. Пока в S не добавили все вершины, повторять шаги 1-3.

Алгоритм Дейкстры

```
Dijkstra(G,w,s){
  Initialize(G,s);
  S ← ø;
  Q ← V;                    //очередь с приоритетами
  While (Q ≠ ø){
    u ← get(Q); //выбрать ближайшую
    S ←  S U {u};
   for (для ∀v ∈ Adj[u])
      Relax ( u, v, w);
  }
}
```

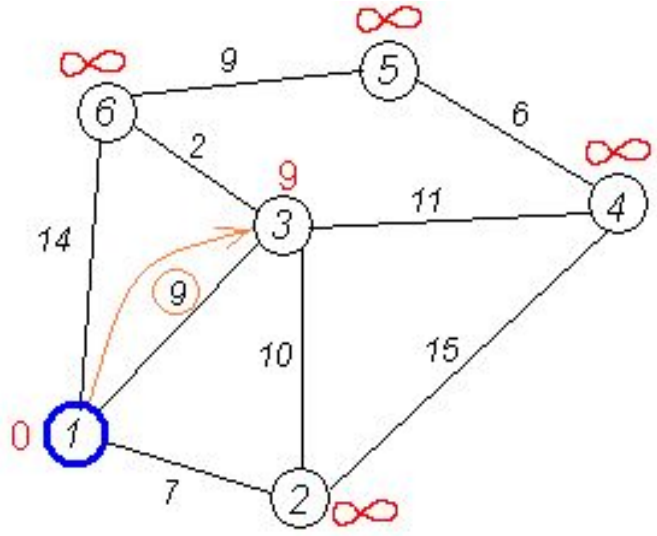
Пример. Каждой вершине из *V* сопоставили метку — минимальное известное расстояние от этой вершины до 1. На каждом шаге посещаем одну вершину и пытаемся уменьшать расстояние.

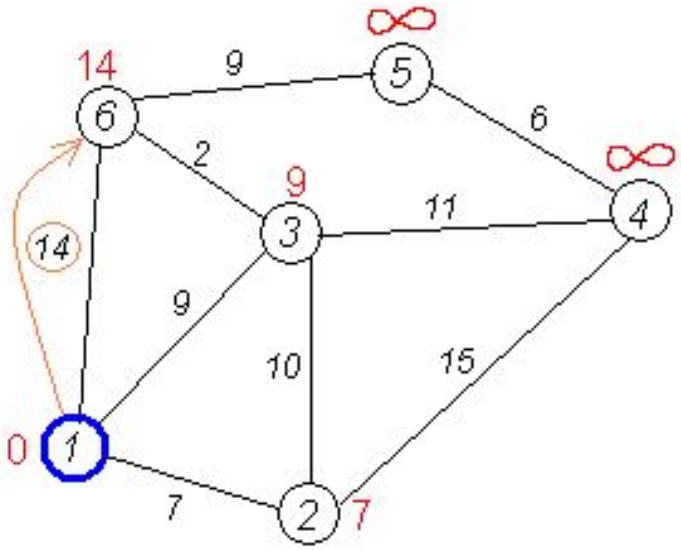


Первый по очереди сосед вершины 1 — вершина 2, потому что длина пути до неё
минимальна. Длинапути в неё через вершину 1 равна кратчайшему расстоянию
до вершины 1 + длина ребра, идущего из 1 в 2, то есть 0 + 7 = 7.

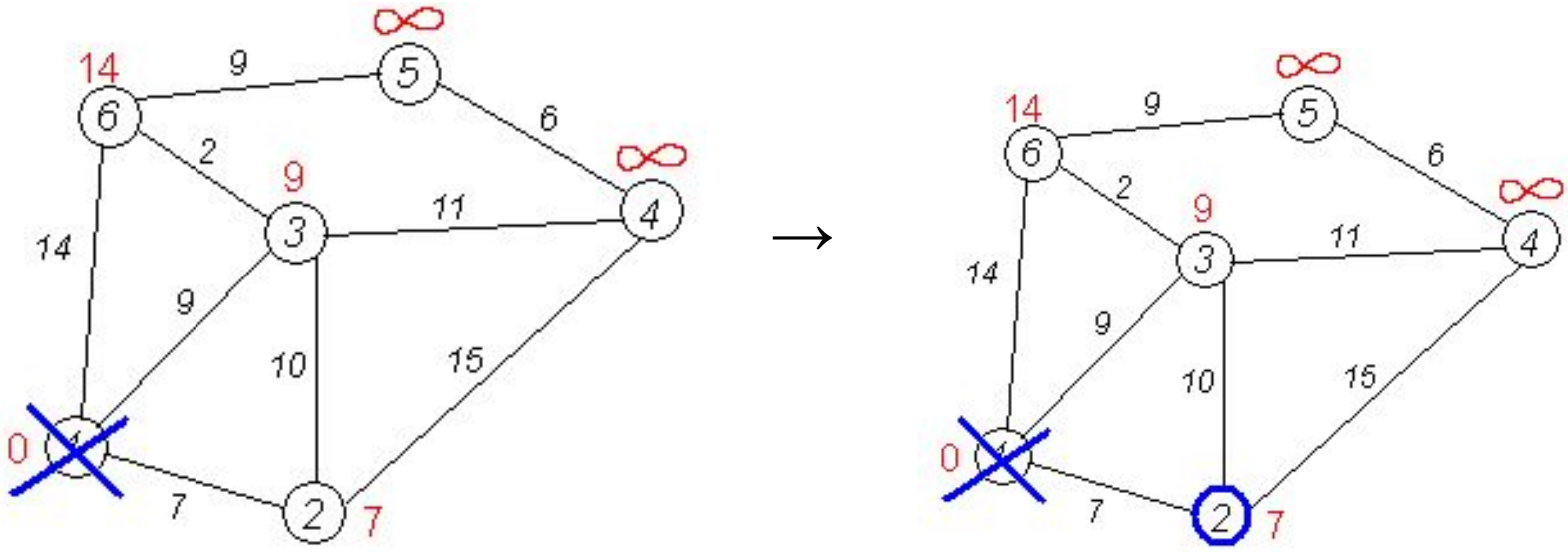
Аналогичную операцию проделываем с двумя другими соседями 1-й вершины —

3-й и 6-й.





Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит. Вычеркнем её, чтобы отметить, что эта вершина посещена.



Снова находим «ближайшую» из непосещенных вершин. Это вершина 2.

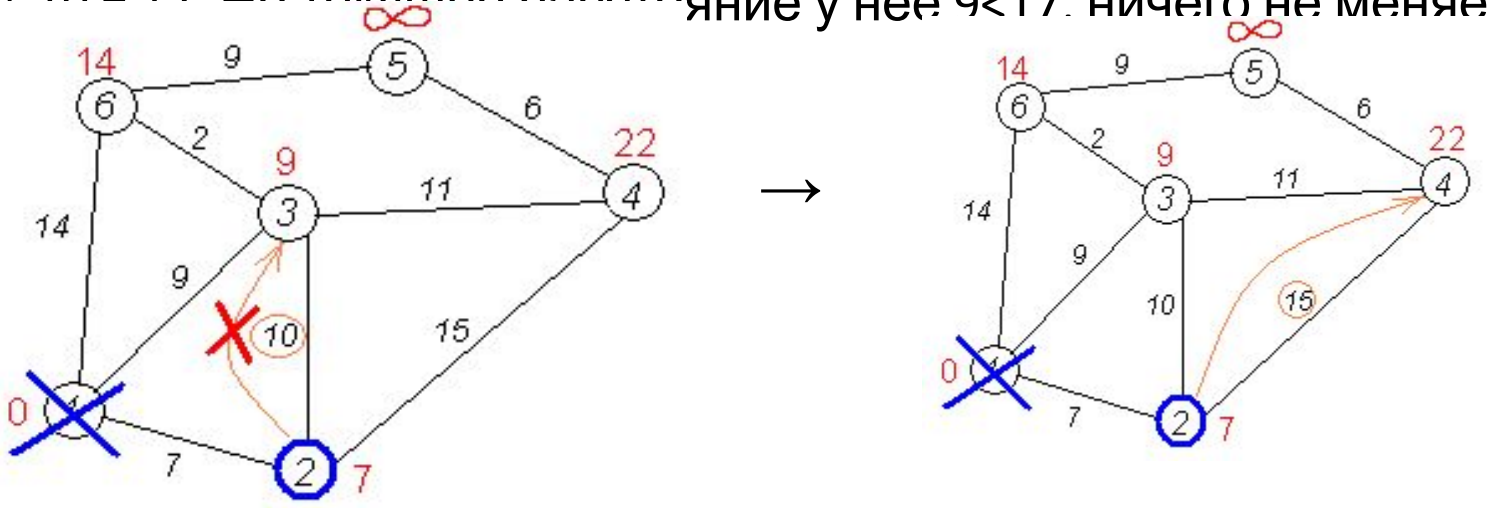
Снова пытаемся уменьшить расстояния у смежных вершин, пытаясь

пройти в них через 2-ю. Смежные вершины к 2 являются 1, 3, 4.

Вершина 1 уже посещалась, поэтому с 1-й вершиной ничего не делаем.

Вершина 3: если идти в неё через 2, то длина такого пути будет

7 + 10 = 17. Но текущее расстояние у нее 9<17, ничего не меняем.

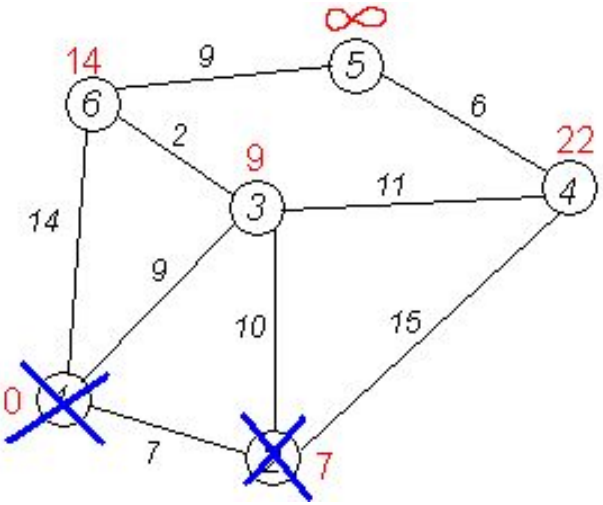


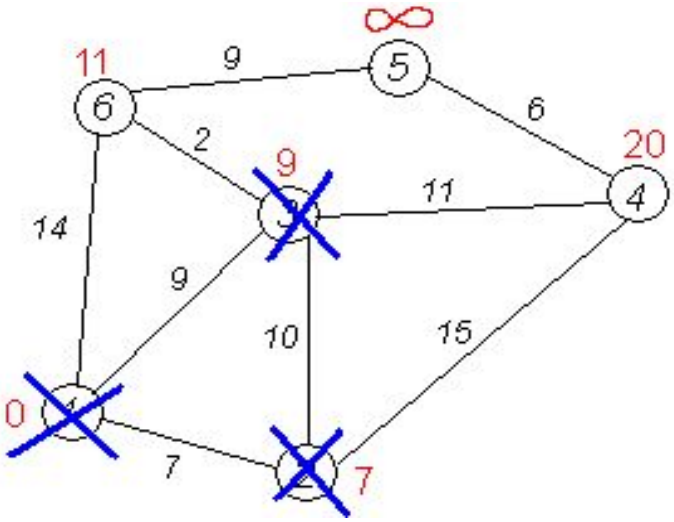
Вершина 4: если идти в неё через 2-ю, то длина такого пути будет = кратчайшее

расстояние до 2 + расстояние между вершинами 2 и 4 = 7 + 15 = 22.

Все смежные вершины с вершиной 2 просмотрены, замораживаем расстояние

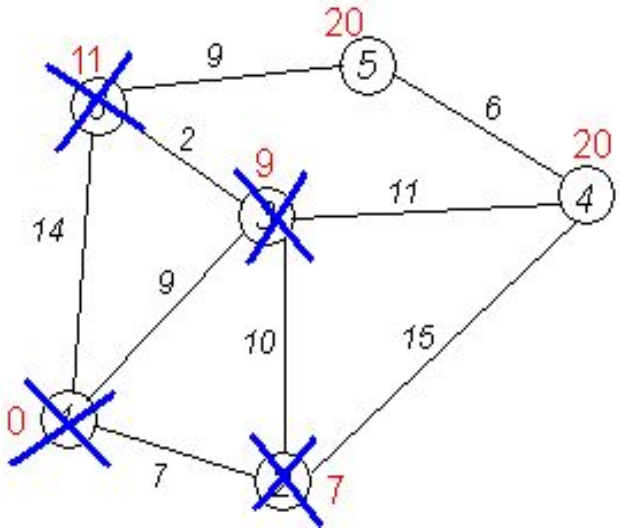
до неё и помечаем её как посещенную.

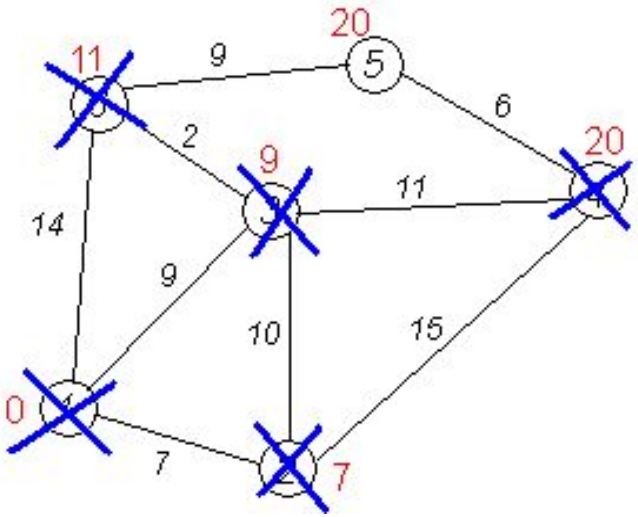


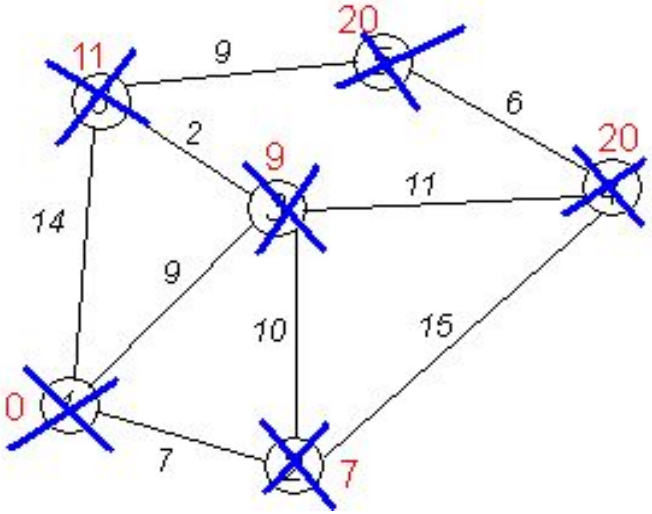


Повторяем шаг алгоритма, выбрав вершину 3. После её обработки получим

Повторяем шаг алгоритма для оставшихся вершин 6, 4 и 5.

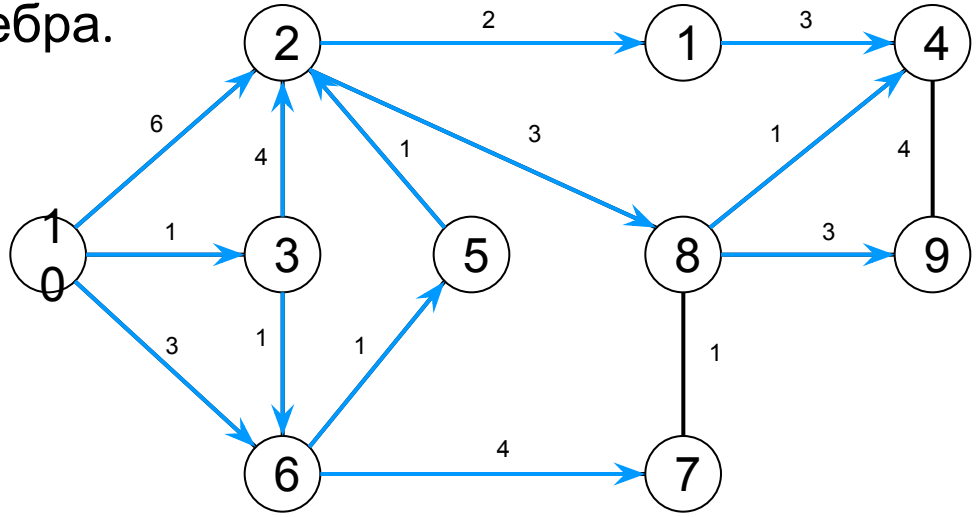




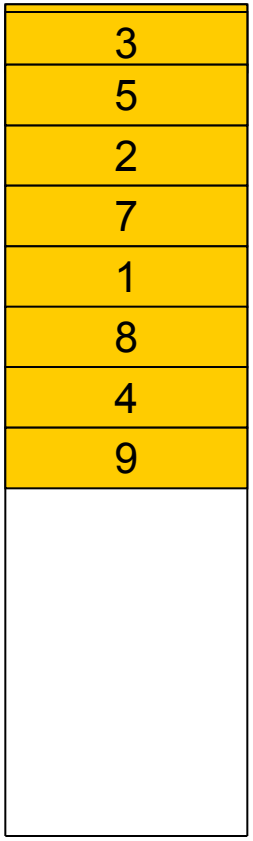


Пример

Очередь с приоритетами. Приоритет – текущая величина найденного расстояния
от начальной вершины. Релаксации подвергаются прямые и обратные ребра.



Очередь



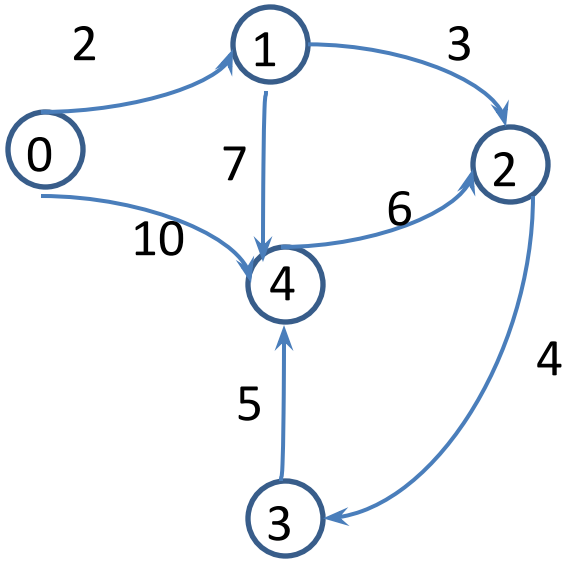
n	1	2	3	4	5	6	7	8	9	10
π	2	5	10	8	6	3	6	2	8	
d	6	4	1	8	3	2	6	7	10	0

Реализация с дополнительным массивом - $O(n^2)$

Массив $D[v]$ содержит стоимость текущего кратчайшего пути из s в v.

Dijkstra

```
{
    S ← {s};
    D[s] ← 0;
    для всех v ∈V \ {v0} выполнить:  D[v] = w (s, v);
    пока S ≠ V выполнять:
    {
     выбрать узел u ∈ V \ S, для которого
          D[u] принимает наименьшее значение;
     добавить  w к S;
     для всех v ∈V \ S выполнить
          D[v] = min (D[v],  D[u] + w(u, v));
    }
}
```



Пример

№	S	u	D[u]	D[1]	D[2]	D[3]	D[4]
0	{0}	-	-	2	+∞	+∞	10
1	{0, 1}	1	2	2	5	+∞	9
2	{0, 1, 2}	2	5	2	5	9	9
3	{0, 1, 2, 3}	3	9	2	5	9	9
4	{0, 1, 2, 3, 4}	4	9	2	5	9	9

Сложность алгоритма Дейкстры зависит от способа нахождения вершины v, а также способа хранения множества непосещенных вершин и способа обновления расстояний.

1. В простейшем случае, когда для поиска вершины с минимальным $d[v]$ просматривается все множество вершин, а для хранения величин d — массив, время работы алгоритма есть $O(n^2 + m)$. Основной цикл выполняется порядка n раз, в каждом из них на нахождение минимума тратится порядка n операций, плюс количество релаксаций, которое не превосходит количества ребер в исходном графе.
2. * Для разреженных графов непосещенные вершины можно хранить в двоичной куче, а в качестве ключа использовать значения $d[i]$, тогда время извлечения вершины станет $\log n$, при том, что время модификации $d[i]$ возрастет до $\log n$. Так как цикл выполняется порядка n раз, а количество релаксаций не больше m, скорость работы такой реализации $O(n\log n + m\log n)$.

3. * Если для хранения непосещенных вершин использовать фибоначчиеву кучу, для которой удаление происходит в среднем за $O(\log n)$, а уменьшение значения в среднем за $O(1)$, то время работы алгоритма составит $O(n \log n + m)$.

Алгоритм Беллмана — Форда

За время $O(n \times m)$ алгоритм находит кратчайшие пути от одной вершины графа до всех остальных, допускает рёбра с отрицательным весом. Предложен независимо Ричардом Беллманом (Bellman) и Лестером Фордом (Ford).

Алгоритм маршрутизации RIP был впервые разработан в 1969 году, как основной для сети ARPANET.

В 1969 году Агентство передовых исследовательских проектов (ARPA) предложило разработать компьютерную сеть.

Компьютерная сеть была названа ARPANET, все работы финансировались
за счёт Министерства обороны США. Затем сеть ARPANET начала активно
расти и развиваться, её начали использовать учёные из разных областей
науки. В 1973 году к сети были подключены первые иностранные
организации из Великобритании и Норвегии, сеть стала международной.
Стоимость пересылки электронного письма по сети ARPANET составляла
50 центов.
В 1984 году у сети ARPANET появился серьёзный соперник,
Национальный фонд науки США (NSF) основал обширную
межуниверситетскую сеть NSFNet, которая имела гораздо бóльшую
пропускную способность (56 кбит/с), нежели ARPANET.
В 1990 году сеть ARPANET прекратила своё существование, полностью
проиграв конкуренцию NSFNet.

Идея алгоритма

Алгоритм позволяет очень просто определить, существует ли

в графе G отрицательный цикл, достижимый из вершины s.

Для проверки нужно произвести внешнюю итерацию цикла

$|V|$ раз. Если при исполнении последней итерации длина

кратчайшего пути до какой-либо вершины строго

уменьшилась, то в графе есть отрицательный цикл,

достижимый из s.

На основе этого можно предложить следующую

оптимизацию. Можно отслеживать изменения в графе и,

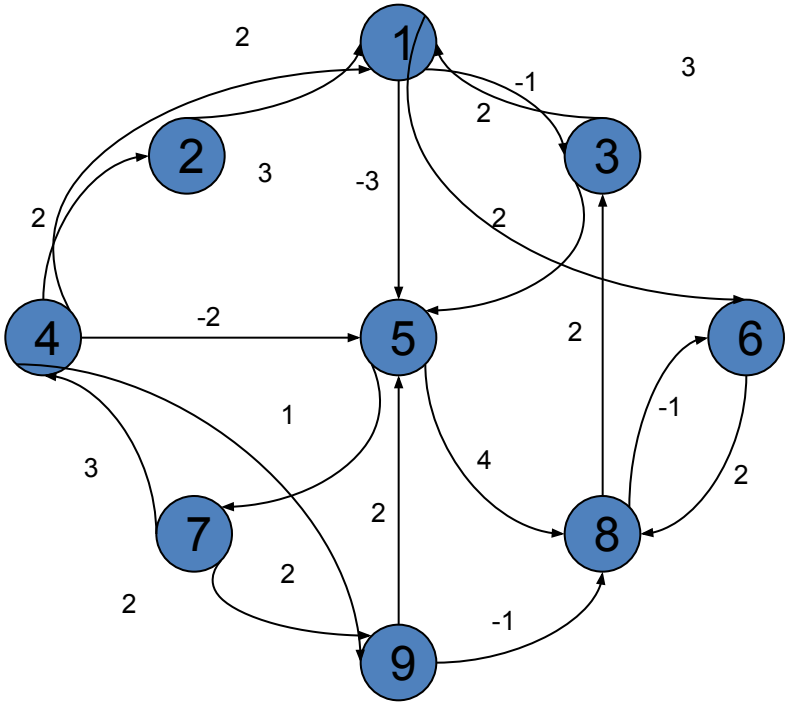
как только они закончатся, дальнейшие итерации будут

```
Bellman-Ford(G,w,s) {
  Initialize(G,s);
  for(i=1; i<n; i++)
   for(∀ (u,v) ∈ E )
      Relax(u,v,w);
  for(∀ (u,v) ∈ E )
   if (d[v]>d[u]+w(u,v)
      return 0;
  return 1;
}
```

Кратчайшие пути в ориентированном графе

1. Если в ориентированном графе нет дуг с отрицательным весом, то алгоритм Дейкстры работает точно так же, как и в случае неориентированных графов.

2. Если в ориентированном графе нет циклов с отрицательным весом, то можно применить алгоритм Беллмана – Форда.



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

И так далее…
В конце концов получится…

Нахождение кратчайших путей между всеми парами вершин

Строим матрицу стоимостей:

$$M[i, j] = \begin{cases} w(i, j), & \text{если ребро } (i, j) \in E \\ +\infty, & \text{если ребро } (i, j) \notin E \\ 0, & \text{если } i = j \end{cases}$$

Обозначим через d [i, j] матрицу кратчайших

путей между всеми вершинами.

Вершины занумеруем числами от 1 до n.

Алгоритм Флойда-Уоршолла

Обозначим через $d_{ij}^{(k)}$ стоимость кратчайшего пути из вершины с номером i в вершину с номером j с промежуточными вершинами из множества {1, 2, …, k}.

$$d_{ij}^{(k)} = \begin{cases} M[i, j] \text{ , если } k = 0, \\ \min(d_{ij}^{(k-1)} , d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), \text{ если } k \geq 1 \end{cases}$$

$D^{(n)}$ содержит искомое решение

Floyd-Warshall(M, n) {

$D^{(0)} \leftarrow M$;

for k ← 1 to n do

for i ←1 to n do

for j ← 1 to n do

$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$;

return $D^{(n)}$;

}

Транзитивное замыкание графа

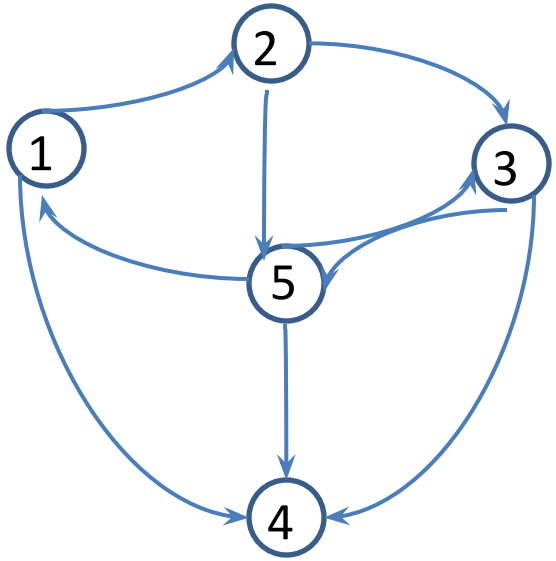
Пусть $G = (V, E)$ ориентированный граф.

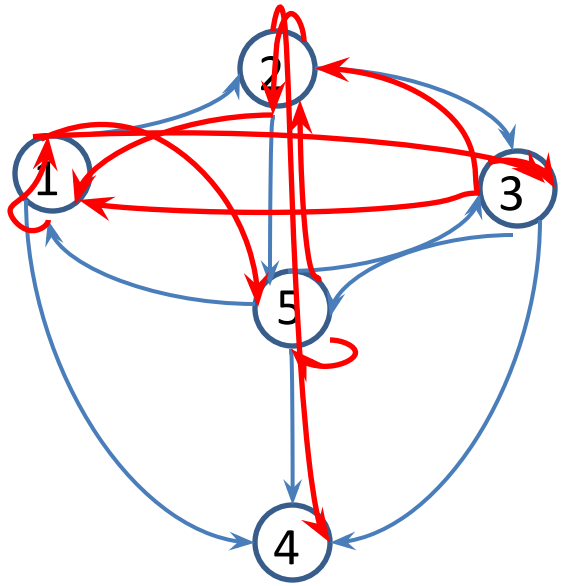
Транзитивным замыканием графа G называется граф $G' = (V, E')$, в котором из вершины v в вершину w идет ребро $\Leftrightarrow$ существует путь (длины 0 или больше) из v в w в графе G.

E':

$(a, b) \in E \;\&\; (b, c) \in E \Rightarrow (a, b) \in E' \;\&\; (b, c) \in E' \;\&\; (a, c) \in E'$

Построение транзитивного замыкания графа. Пример





Обозначим через $t_{ij}^{(k)}$ наличие пути из вершины с номером i в вершину с номером j с промежуточными вершинами из множества {1, 2, …, k}. M – матрица смежностей графа G.

$$t_{ij}^{(k)} = \begin{cases} M[i, j], \text{ если } k = 0, \\ t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}), \text{ если } k \geq 1 \end{cases}$$

$T^{(n)}$ содержит искомое решение.

Алгоритм построения транзитивного замыкания графа

Tranzitive_Closure(M, n)

{

$T^{(0)} \leftarrow M$;

for k ← 1 to n do

for i ←1 to n do

for j ← 1 to n do

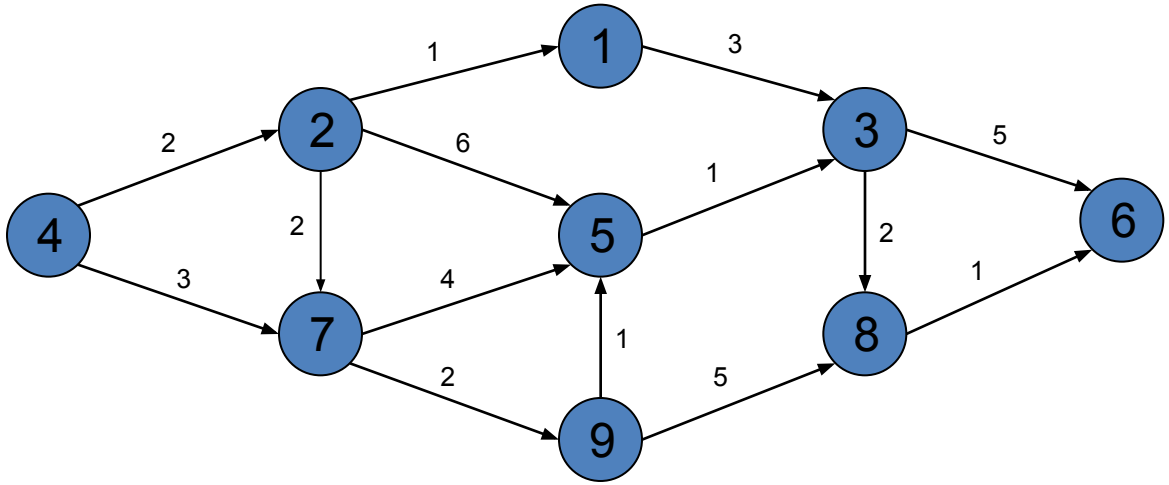
$t_{ij}^{(k)} \leftarrow t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$;

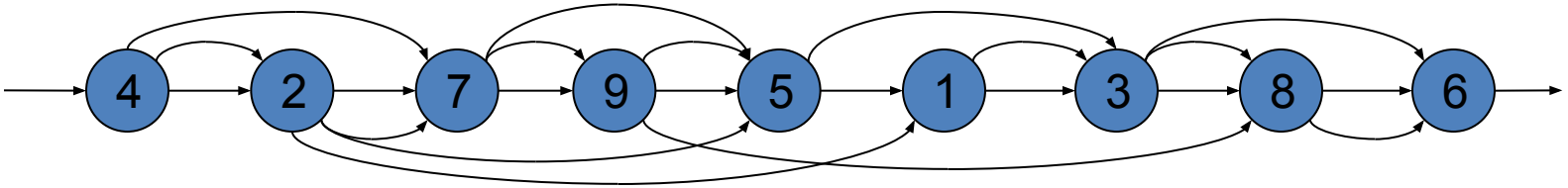
return $T^{(n)}$;

}

Кратчайшие пути в ориентированном графе

3. Если в ориентированном графе нет циклов, то можно провести топологическую сортировку вершин, после чего выполнить релаксацию исходящих дуг в порядке возрастания номеров вершин.

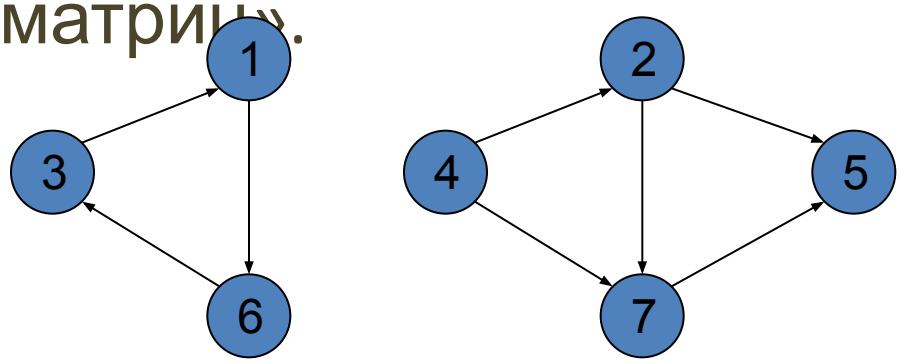




n	1	2	3	4	5	6	7	8	9
π	2	4	1		9	8	4	3	7
d	3	2	6	0	6	9	3	8	5

Один из вариантов применения алгоритма: нахождение критического пути.

Алгоритм «умножения матриц».



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

Пусть матрица $G^{(l)}$ представляет собой граф путей длиной l (то есть в матрице единица находится в ячейке (u,v), если в исходном графе существовал путь из u в v длиной не больше l).

Тогда матрица $G^{(1)}$ – это матрица смежности исходного графа G,
$G^{(n)}$ – матрица смежности его транзитивного замыкания (очевидно, что если в графе существует путь длины, большей *n*, то существует и путь, длины не большей *n*).

Алгоритм нахождения транзитивного замыкания: если удается вычислить $G^{(l+1)}$ по $G^{(l)}$, то можно, начав с матрицы G, за *n* шагов получить матрицу $G^{(n)}$.