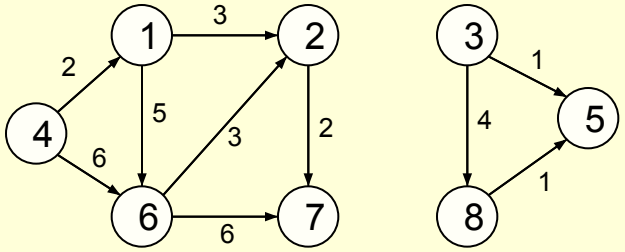
Представление графов. Матрица смежности

Граф:



	1	2	3	4	5	6	7	8
1	∞	3	∞	∞	∞	5	∞	∞
2	∞	∞	∞	∞	∞	∞	2	∞
3	∞	∞	∞	∞	1	∞	∞	4
4	2	∞	∞	∞	∞	6	∞	∞
5	∞	∞	∞	∞	∞	∞	∞	∞
6	∞	3	∞	∞	∞	∞	6	∞
7	∞	∞	∞	∞	∞	∞	∞	∞
8	∞	∞	∞	∞	1	∞	∞	∞

Удобно:

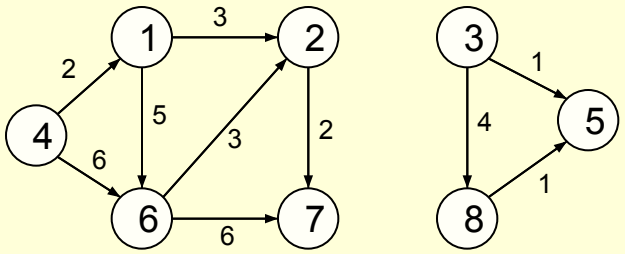
- Добавлять и удалять ребра
- Проверять смежность вершин

Неудобно:

- Добавлять и удалять вершины
- Работать с разреженными графами

Представление графов. Матрица инцидентности

Граф:



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

Удобно:

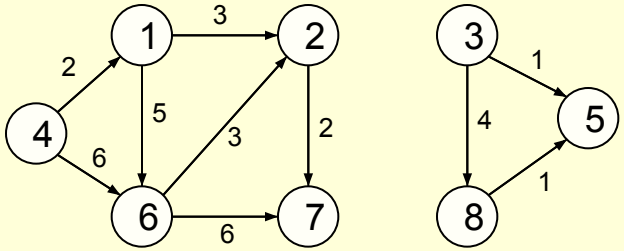
- Менять нагрузку на ребра
- Проверять инцидентность

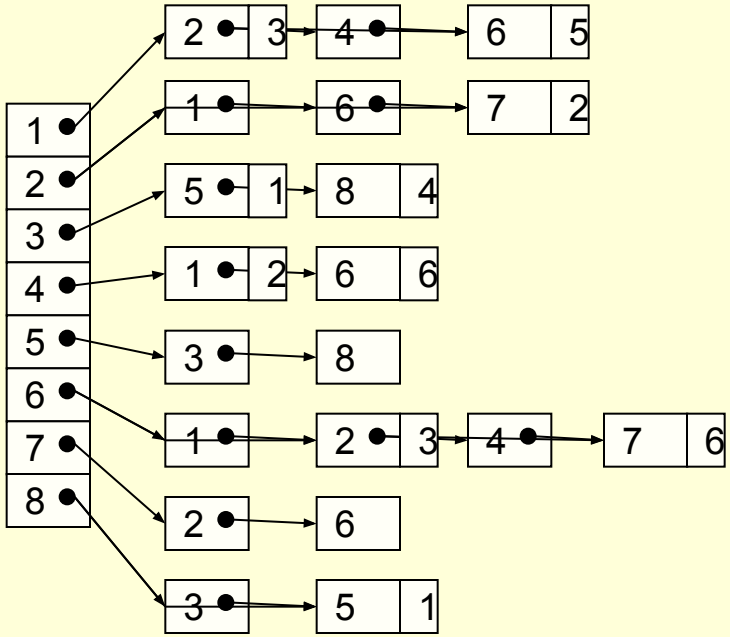
Неудобно:

- Добавлять и удалять вершины
- Работать с разреженными графами

Представление графов. Списки смежности

Граф:





Удобно:

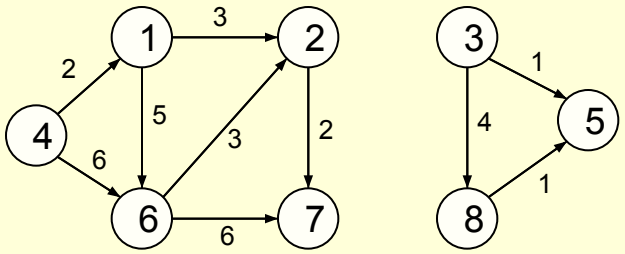
- Искать вершины, смежные с данной
- Добавлять ребра и вершины
- Работать с разреженными графами

Неудобно:

- Проверять наличие ребра
- Удалять ребра и вершины

Представление графов. Список ребер

Граф:

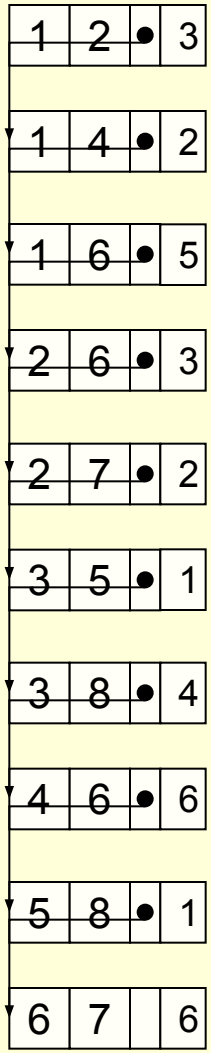


Удобно:

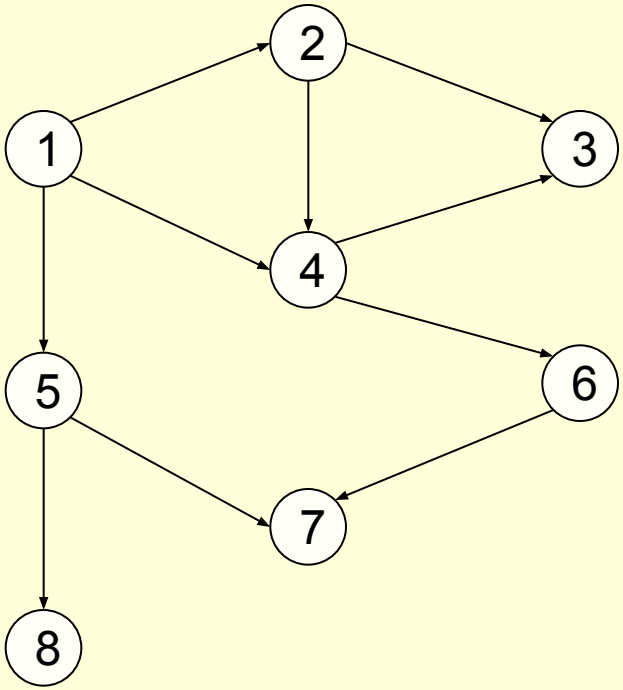
- Добавлять и удалять ребра
- Упорядочивать ребра по возрастанию нагрузки
- Представлять сильно разреженные графы

Неудобно:

- Определять смежность вершин и ребер
- Осуществлять перебор инцидентных заданной вершине ребер



Обходы графов. Рекурсивная процедура



Обход вершин и дуг графа, достижимых из заданной вершины (v):

1. Пометить вершину (v) как пройденную вперед
2. Цикл по дугам (e), инцидентным вершине (v)
   1) Пометить дугу (e) как пройденную вперед
   2) Если конец дуги (u) не отмечен,
      - то обойти достижимые из (u) вершины и дуги (рекурсивный вызов)
   3) Пометить дугу (e) как пройденную назад
3. Пометить вершину (v) как пройденную назад

Задачи, которые можно решить с помощью этой процедуры:

1. Определить вершины, достижимые из заданной;
2. Обойти все вершины и дуги графа в определенной последовательности («в глубину»);
3. Определить некоторые характеристики связности графа;

… и многие другие

Обходы графов. Рекурсивная процедура

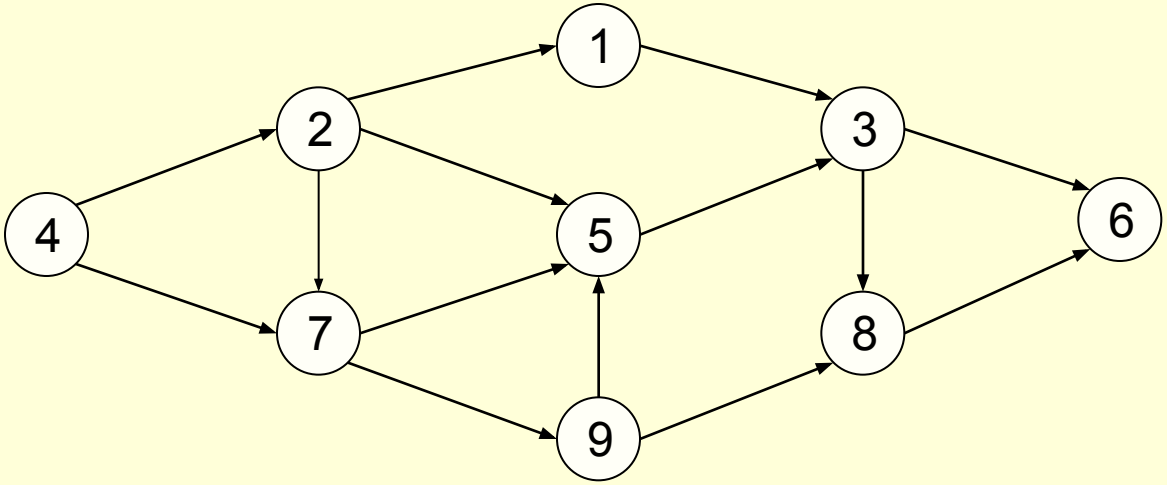
```
public class Graph {
    public static class Arc {
        public Arc(int to,
                   double weight,
                   Arc next) {
            this.to = to;
            this.weight = weight;
            this.next = next;
        }
        private int to;
        private double weight;
        private Arc next;
    }

    private int nVert;
    private Arc[] list;

    public Graph(int nVert) {
        this.nVert = nVert;
        list = new Arc[nVert];
    }
}
```

```
private static void traverseDepthComponent
        (int i, Graph g, GraphVisitor visitor,
         boolean[] visited) {
    visitor.visitVertexIn(i);
    visited[i] = true;
    for (Iterator arcs = g.adjacentArcs(i);
                arcs.hasNext(); ) {
        Graph.Arc arc = (Graph.Arc)arcs.next();
        visitor.visitArcForward(i, arc,
                                visited[arc.getTo()]);
        if (!visited[arc.getTo()]) {
            traverseDepthComponent(arc.getTo(), g,
                                   visitor, visited);
        }
        visitor.visitArcBackward(i, arc);
    }
    visitor.visitVertexOut(i);
}
```

```
public abstract class GraphVisitor {
    public void visitArcForward(int from, Graph.Arc arc, boolean retArc) {}
    public void visitArcBackward(int from, Graph.Arc arc) {}
    public void visitVertexIn(int v) {}
    public void visitVertexOut(int v) {}
    public void visitComponentStart(int start) {}
}
```

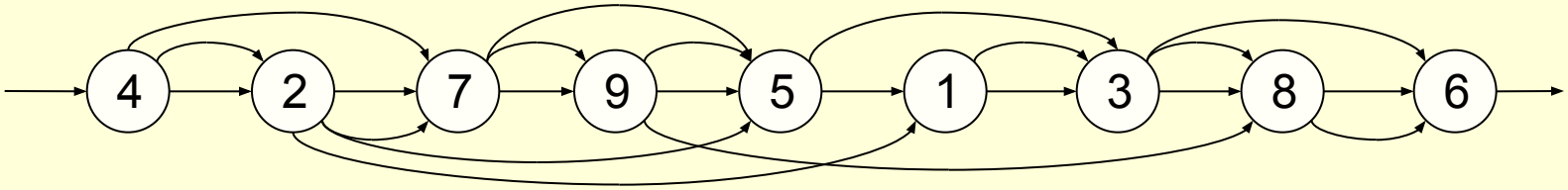
Топологическая сортировка вершин ориентированного графа без циклов.

DAG – Directed Acyclic Graph – ориентированный граф без циклов

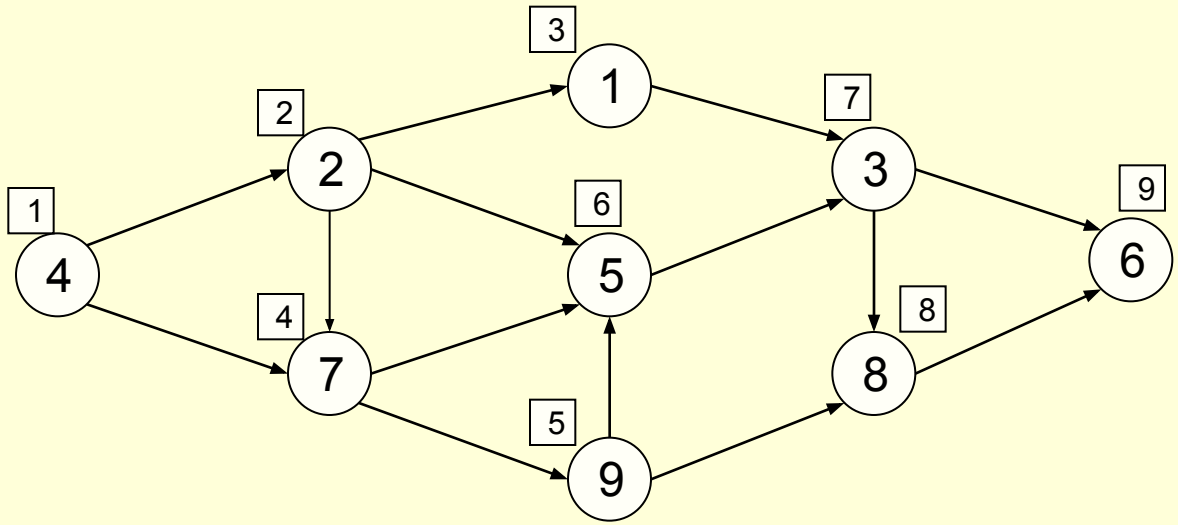


Интерпретация: вершины – элементарные работы;
дуги – зависимость работ друг от друга.

Задача: выстроить работы в последовательности, в которой никакая следующая задача не может зависеть от предыдущей (дуги направлены только вперед)



Топологическая сортировка вершин ориентированного графа без циклов.



«Наивный» алгоритм нумерации вершин:

1. Находим какую-либо вершину, в которую не входят дуги, нумеруем ее.
2. Помечаем дуги, выходящие из помеченной вершины, как «не существующие».
3. Повторяем шаги (1) и (2), пока не будут занумерованы все вершины.

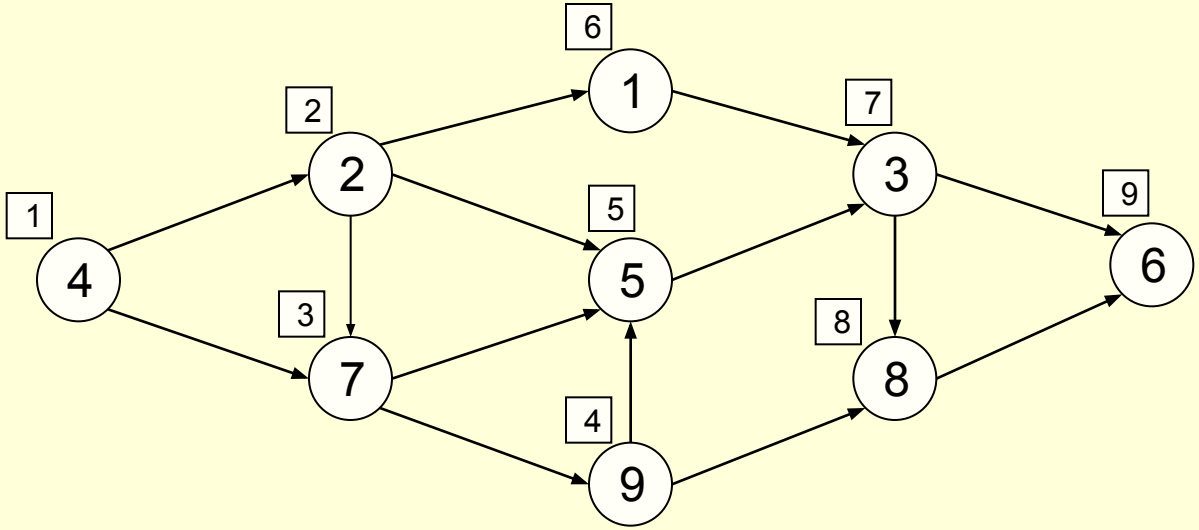
Оценка времени работы алгоритма.

1. Построение очереди с приоритетами из вершин графа	n log n
2. Выборка вершин из очереди	n log n
3. Коррекция очереди при пометке дуг	m log n
Итого:	(m+2n) log n

Проверка ацикличности графа.

Граф содержит цикл, если на шаге (1) не удается найти вершину, в которую не входят дуги.

Топологическая сортировка вершин ориентированного графа без циклов.



«Эффективный» алгоритм нумерации вершин:

1. Производим обход графа с помощью рекурсивной процедуры обхода, начиная с произвольной вершины.
2. Нумеруем каждую вершину при «прохождении ее назад» максимальным из номеров (то есть нумерация происходит в порядке убывания номеров).
3. Повторяем шаги (1) и (2), пока не останется непройденных вершин.

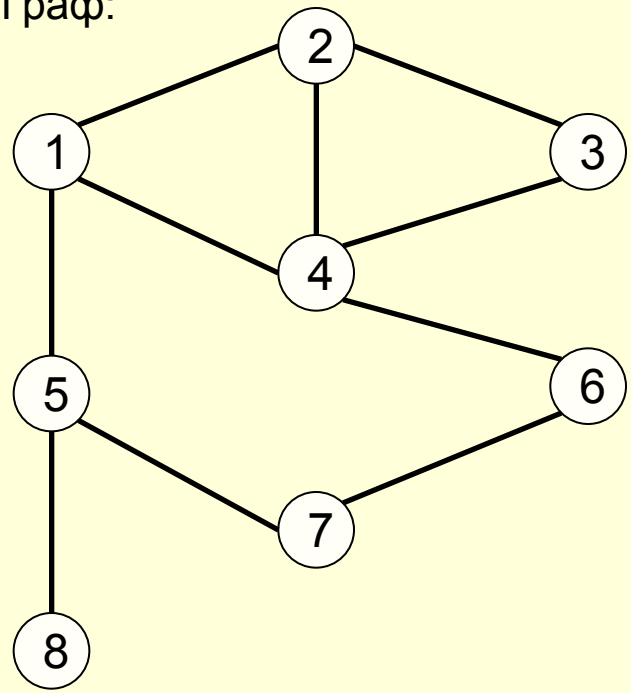
Оценка времени работы алгоритма = время обхода = (m+n).

Проверка ацикличности графа.

Граф содержит цикл, если при проходе по «обратной дуге» попадаем в еще непомеченную («синюю») вершину.

Обходы графов. Общая процедура с использованием структуры хранения вершин

Граф:



1. Начинаем с произвольной выбранной вершины *v*.
2. Помещаем вершину *v* в контейнер:
```
Collection cont = new Collection();
cont.add(v);
```
3. Цикл
```
while (!cont.empty())
```
   1) Выбрать вершину и пометить ее:
```
mark(current = cont.get());
```
   2) Просмотреть все инцидентные ребра
      i. Если ребро ведет в непомеченную вершину *u* (прямая дуга на ребре),
```
cont.add(v);
```
      ii. Иначе, если ребро идет в вершину из контейнера (обратная дуга на ребре) или в помеченную вершину (встречная дуга), то ничего не делаем.

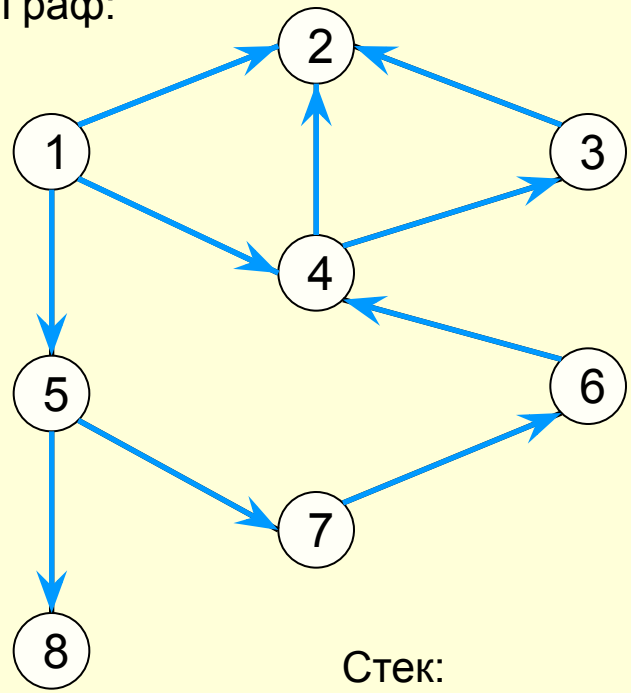
Используя различные контейнеры, с помощью этой процедуры можно решать разнообразные задачи:

Контейнер: 2 3 6 8

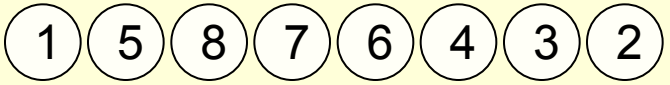
- Обходы вершин и дуг графа в различном порядке (итерация)
- Поиск маршрутов, в том числе, минимальных
- и многие другие…

Использование стека для обхода графа.

Граф:



Если в качестве промежуточной структуры хранения при обходе использовать стек, то получается обход в глубину.



Можно также получить дерево обхода в глубину, если отмечать каждую прямую или обратную дугу.

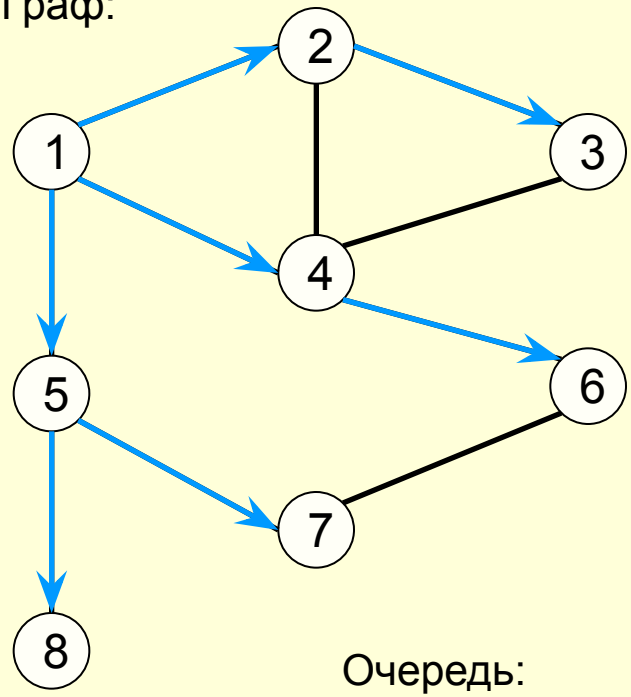
1	2	3	4	5	6	7	8
	3	4	6	1	7	5	5

Стек:

2
4
7
8

Использование очереди для обхода графа.

Граф:



Очередь:

2
4
7
8

Если в качестве промежуточной структуры хранения при обходе использовать очередь, то получается обход в ширину.

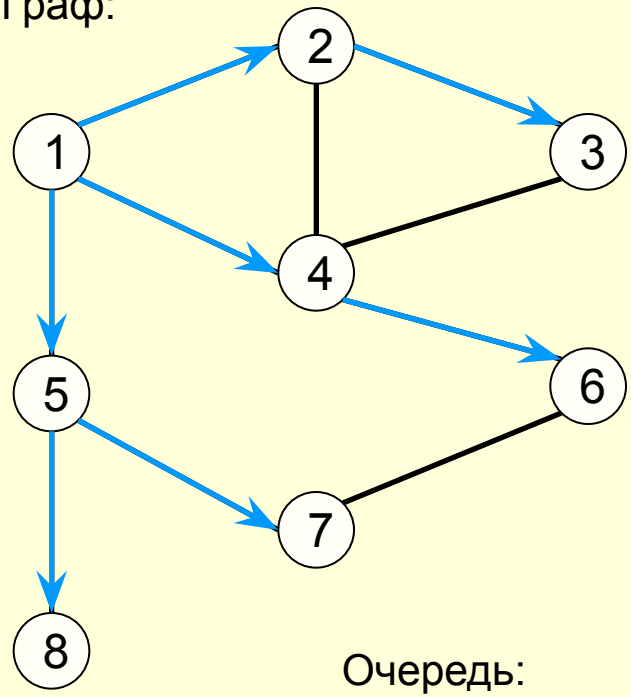
1 2 4 5 3 6 7 8

Можно также получить дерево обхода в ширину, если отмечать каждую прямую дугу.

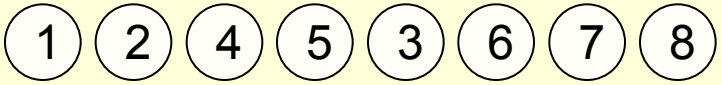
1	2	3	4	5	6	7	8
	1	2	1	1	4	5	5

Поиск кратчайших путей в ненагруженном графе.

Граф:

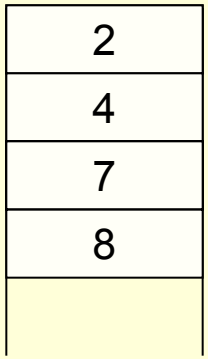


Алгоритм обхода графа с помощью очереди позволяет построить дерево кратчайших путей из заданной вершины и вычислить их длины.



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

Очередь:



Программа обхода графа с использованием контейнера.

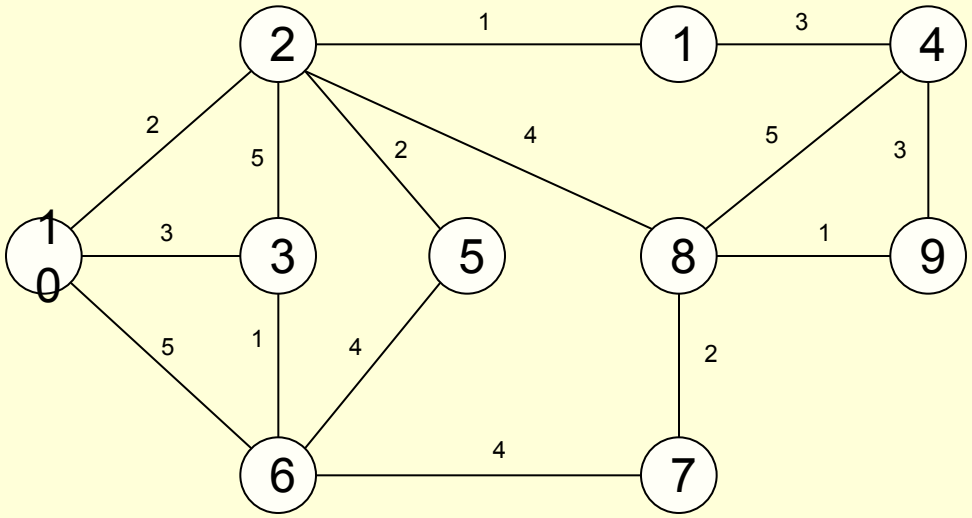
```
public static void traverseWithContainer(Graph g, GraphVisitor visitor, int start,
                ContainerFactory factory) {
    // Инициализация
    int n = g.vertices();                                  // Число вершин графа
    Container container = factory.createContainer(n);  // Создание контейнера
    container.push(start);                                 // Инициализация контейнера
    boolean[] inQueue = new boolean[n];
    boolean[] passed = new boolean[n];
    inQueue[start] = true;
    visitor.visitVertexIn(start);

    while (!container.isEmpty()) {
        int current = container.pull();
        passed[current] = true;
        visitor.visitVertexOut(current);
        for (Iterator arcs = g.adjacentArcs(current); arcs.hasNext(); ) {
            Graph.Arc arc = (Graph.Arc)arcs.next();
            int end = arc.getTo();
            if (passed[end])
                visitor.visitArcBackward(current, arc);
            else {
                visitor.visitArcForward(current, arc, inQueue[end]);
                if (!inQueue[end]) {
                    container.push(end);
                    inQueue[end] = true;
                    visitor.visitVertexIn(end);
                }
            }
        }
    }
}
```

Программа поиска кратчайших путей в ненагруженном графе.

```
public static void findMinPaths(Graph g,           // Исходный граф
                                int start,         // Стартовая вершина
                                final int[] tree,  // Результирующее дерево
                                final int[] dist)  // Массив расстояний
{
    dist[start] = 0;
    tree[start] = 0;
    traverseWithContainer(g, new GraphVisitor() {
            // Коррекция пути производится при первом проходе по дуге вперед
            public void visitArcForward(int from, Graph.Arc arc, boolean retArc) {
                if (!retArc) {
                    int end = arc.getTo();
                    dist[end] = dist[from] + 1;
                    tree[end] = from;
                }
            }
        },
        start, new QueueFactory());
}
```

Алгоритмы поиска кратчайших путей в нагруженном графе.

Кратчайший путь между двумя вершинами – это путь с минимальным суммарным весом составляющих этот путь ребер

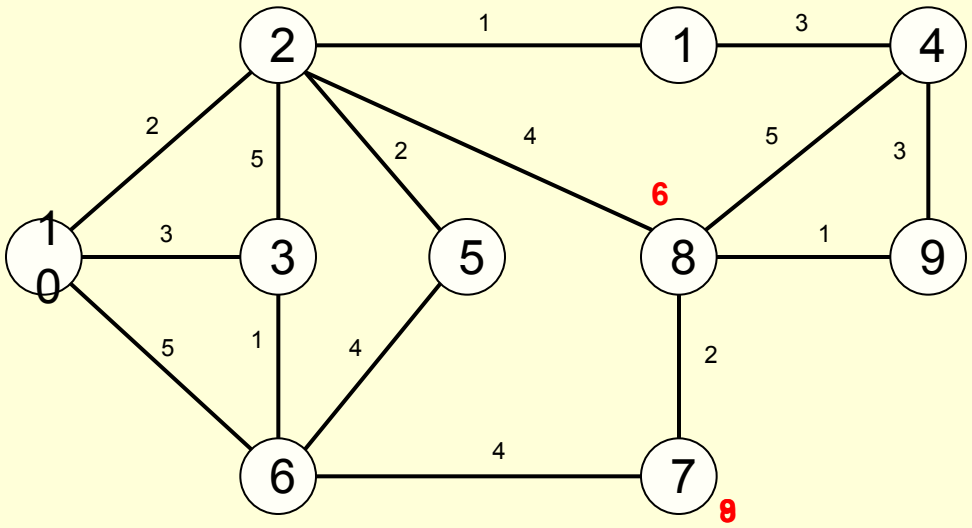


Кратчайшие пути из вершины (10):

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

Кратчайший путь между двумя вершинами существует, если в графе нет цикла с суммарным отрицательным весом.

Алгоритм релаксации ребра при поиске кратчайших путей.



Пусть уже найдены оценки кратчайших путей для вершин, соединенных 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
```

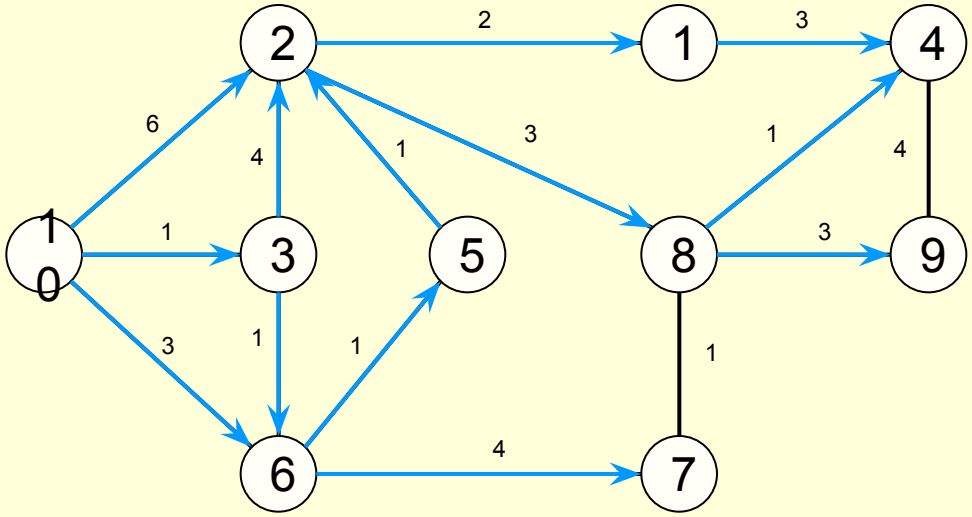
Инициализация:

```
d[start] = 0; d[i] = ∞
```

Последовательная релаксация ребер приведет к нахождению кратчайших путей.
В каком порядке и сколько раз производить релаксацию ребер?

Алгоритм Дейкстры поиска кратчайших путей.

Запустить алгоритм обхода графа, в качестве контейнера выбрать очередь с приоритетами. Приоритет – текущая величина найденного расстояния от начальной вершины. Релаксации подвергаются прямые и обратные ребра.



Очередь

3
5
2
7
1
8
4
9

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

Программа для реализации алгоритма Дейкстры поиска кратчайших путей.

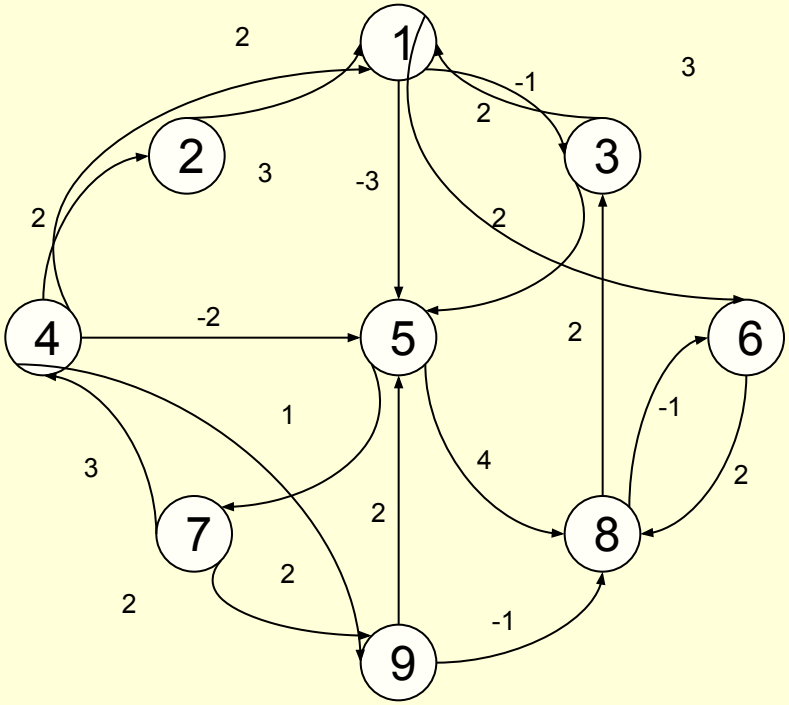
```
public static void DijkstraPath(Graph g, int start,
                                 final int[] tree,
                                 final double[] dist) {
    for (int i = 0; i < g.vertices(); i++) { dist[i] = Double.MAX_VALUE; }
    dist[start] = 0;
    traverseWithContainer(g, new DepthVisitor() {
        public void visitArcForward(int from, Graph.Arc arc, boolean retArc)
        {
            if (dist[from] + arc.getWeight() < dist[arc.getTo()]) {
                dist[arc.getTo()] = dist[from] + arc.getWeight();
                tree[arc.getTo()] = from;
            }
        }
    }, start, new SimplePrioQueueFactory(dist));
}
```

Время работы алгоритма (max): время обхода графа (n + m) плюс
время на организацию работы очереди с приоритетами (n log n)

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.

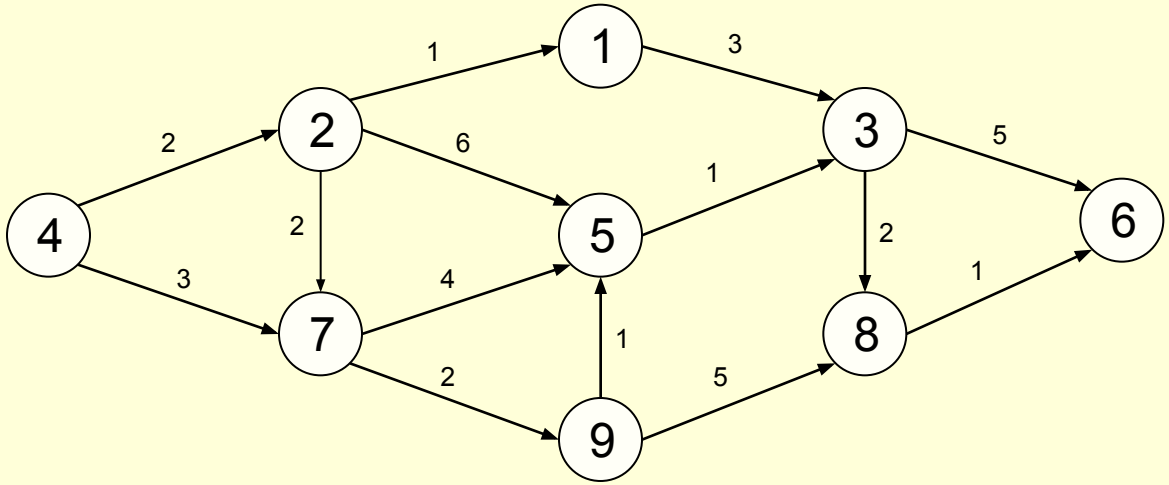


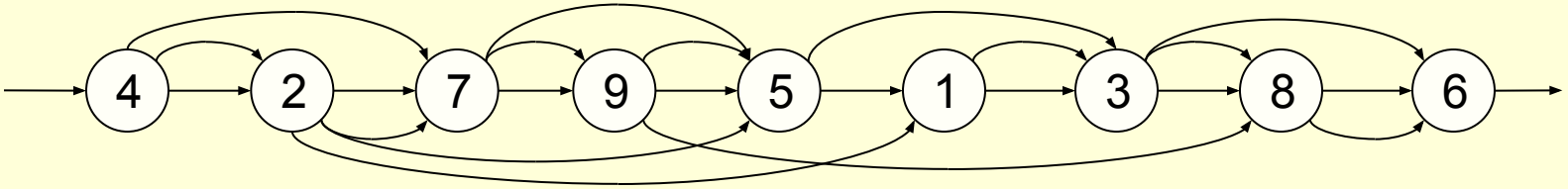
n	1	2	3	4	5	6	7	8	9
π	3	4	8		4	8	5	9	7
d	1	2	2	0	-2	-1	-1	0	1

И так далее…
В конце концов получится…

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.

Транзитивное замыкание графа отношения.

Ориентированный ненагруженный граф представляет отношение на множестве его вершин:

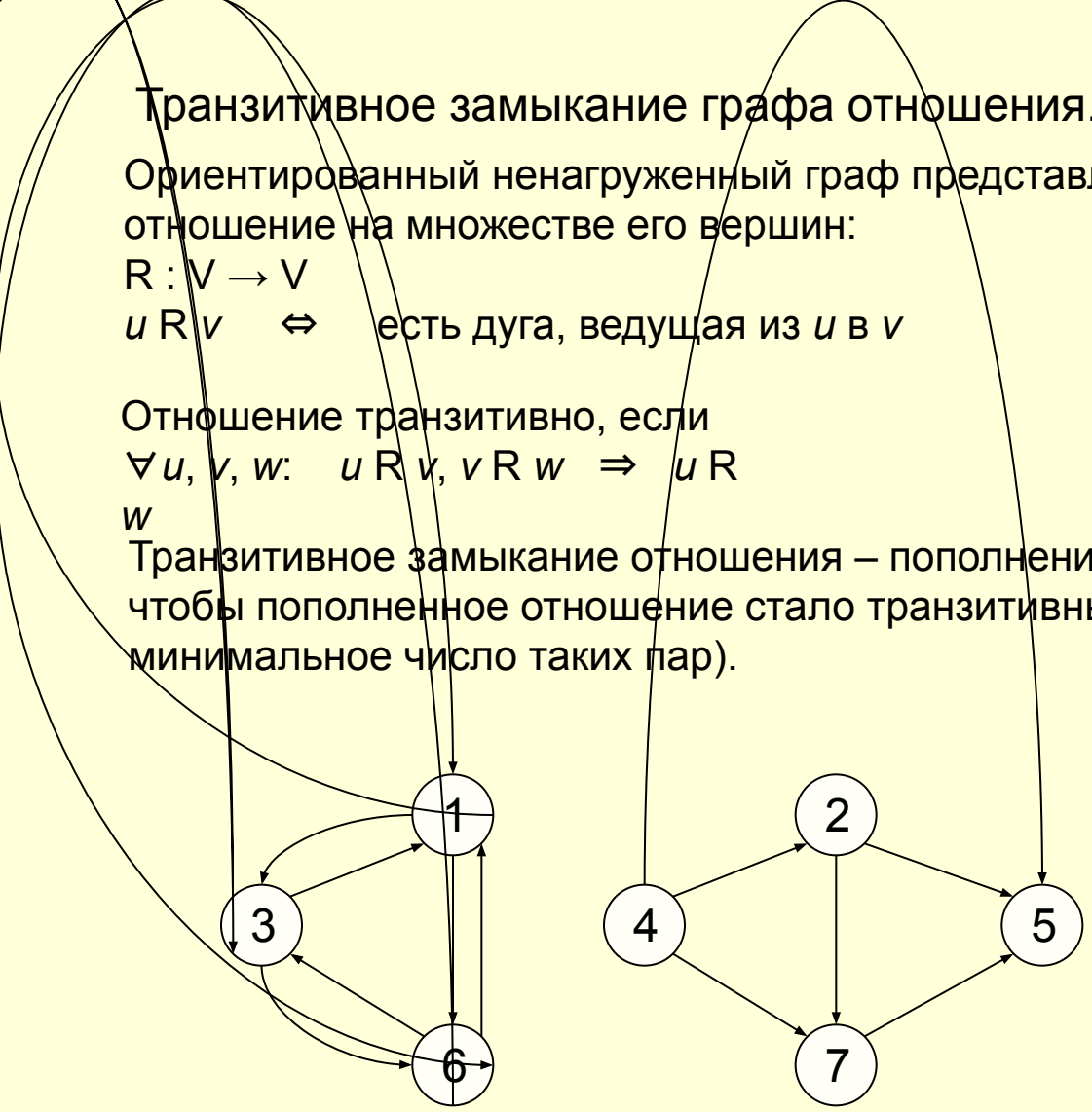
$R : V \to V$

$u \; R \; v \quad \Leftrightarrow$ есть дуга, ведущая из u в v

Отношение транзитивно, если

$\forall u, v, w: \quad u \; R \; v, \; v \; R \; w \quad \Rightarrow \quad u \; R \; w$

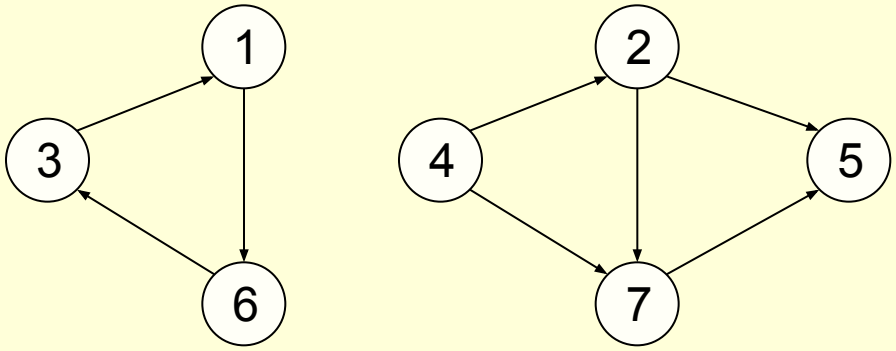
Транзитивное замыкание отношения – пополнение отношения новыми парами так, чтобы пополненное отношение стало транзитивным (необходимо добавить минимальное число таких пар).



Отношение нетранзитивно

Задача нахождения транзитивного замыкания на языке графов:
провести новую дугу из u в v, если в исходном графе существовал путь из u в v.

Транзитивное замыкание графа отношения. 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 $G^{(n)}$.

Транзитивное замыкание графа отношения. Алгоритм «умножения матриц».

Пусть матрица $G^{(l)}$ представляет собой граф путей длиной l (то есть в матрице $G^{(l)}$ единица находится в ячейке (u,v), если в исходном графе существовал путь из u в v длиной не больше l). Тогда что такое матрица $G^{(l+1)}$?

$G^{(l+1)}$ [u,v] = 1, если найдется w такое, что $G^{(l)}$ [u,w] = 1 и G [w,v] = 1.

$$G^{(l+1)}[u,v] = \sum_{w=1}^{n} G^{(l)}[u,w] \times G[w,v]$$

(умножение и сложение понимаются в смысле логических операций «*и*» и «*или*»)

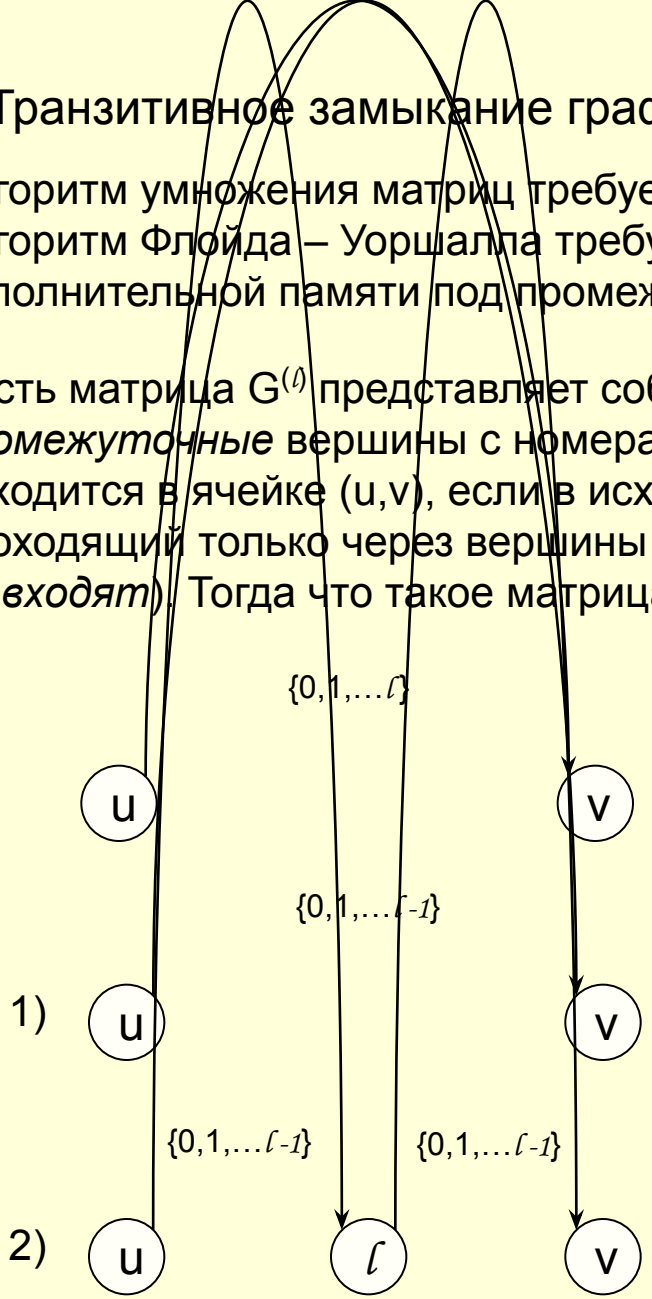
Алгоритм умножения матриц:

```
public static boolean[][] multiply (boolean[][] matr1, boolean[][] matr2) {
    int n = matr1.length;
    boolean[][] matr = new boolean[n][n];
    for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) {
        matr[i][j] = false;
        for (int k = 0; k < n; k++) {
            matr[i][j] ||= matr1[i][k] && matr2[k][j];
    }}}
    return matr;
}

public static boolean[][] transClosure(boolean[][] G) {
    boolean[][] Gl = G;
    for (int l = 1; l < n; l++) { Gl = multiply(Gl, g); }
}
```

Транзитивное замыкание графа отношения. Алгоритм Флойда – Уоршалла

Алгоритм умножения матриц требует порядка n^4 простых логических операций. Алгоритм Флойда – Уоршалла требует лишь n^3 простых операций и не требует дополнительной памяти под промежуточные матрицы.

Пусть матрица $G^{(l)}$ представляет собой граф путей, проходящих через *промежуточные* вершины с номерами от 0 до l-1 (то есть в матрице $G^{(l)}$ единица находится в ячейке (u,v), если в исходном графе существовал путь из u в v, проходящий только через вершины из множества {0,… l-1}, u и v в это множество *не входят*). Тогда что такое матрица $G^{(l+1)}$?



$G^{(l+1)}[u,v] = G^{(l)}[u,v] \;||\; G^{(l)}[u,l] \;\&\&\; G^{(l)}[l,v]$

$G^{(l+1)}[u,v] = 1$, если $G^{(l)}[u,v] == 1$

$G^{(l+1)}[u,v] = 1$, если $G^{(l)}[u,l] == 1 \;\&\&\; G^{(l)}[l,v] == 1$

Транзитивное замыкание графа отношения. Алгоритм Флойда – Уоршалла

$G^{(l+1)}[u,v] = G^{(l)}[u,v] \;||\; G^{(l)}[u,l] \;\&\&\; G^{(l)}[l,v]$

$$G^{(l+1)}[u,v] = \begin{cases} G^{(l)}[u,v] & ,если\ G^{(l)}[u,l] == 0 \\ G^{(l)}[u,v] \;||\; G^{(l)}[l,v] & ,если\ G^{(l)}[u,l] == 1 \end{cases}$$

$$G^{(l+1)}[u] = \begin{cases} G^{(l)}[u] & ,если\ G^{(l)}[u,l] == 0 \\ G^{(l)}[u] \;||\; G^{(l)}[l] & ,если\ G^{(l)}[u,l] == 1 \end{cases}$$

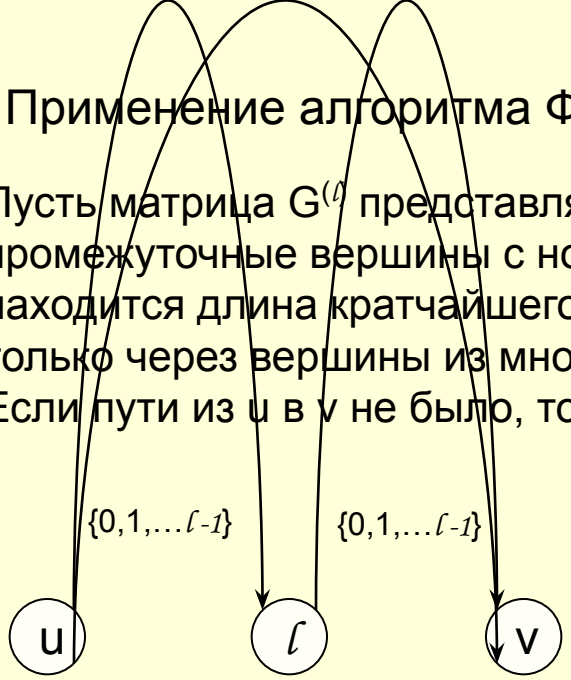
При $u = l$ всегда $G^{(l+1)}[u] = G^{(l)}[u]$

Алгоритм Флойда – Уоршалла нахождения транзитивного замыкания графа отношения.

```
public static boolean[][] transClosure (boolean[][] G) {
    int n = G.length;
    for (int l = 0; l < n; l++) {
        // Формирование матрицы G(l+1):
        for (int u = 0; u < n; u++) {
            if (G[u][l]) {
                for (int v = 0; v < n; v++) {
                    G[u][v] ||= G[l][v];
                }
            }
        }
    }
    return G;
}
```

Применение алгоритма Флойда – Уоршалла для поиска кратчайших путей

Пусть матрица $G^{(l)}$ представляет собой граф *кратчайших* путей, проходящих через промежуточные вершины с номерами от 0 до l-1. То есть в матрице $G^{(l)}$ в ячейке (u,v) находится длина кратчайшего пути из u в v, если он существовал, проходящий только через вершины из множества {0,… l-1}, u и v в это множество не входят. Если пути из u в v не было, то в соответствующей ячейке матрицы будет значение ∞.



$$G^{(l+1)}[u,v] = \min(G^{(l)}[u,l] + G^{(l)}[l,v], G^{(l)}[u,v])$$

{0,1,…l-1}

```
public static double[][] minPathsMatrix (double[][] G) {
    int n = G.length;
    for (int l = 0; l < n; l++) {
        // Формирование матрицы G(l+1):
        for (int u = 0; u < n; u++) {
            if (G[u][l] < Double.MAX_VALUE) {
                for (int v = 0; v < n; v++) {
                    if (G[l][v] < Double.MAX_VALUE)
                        G[u][v] = Math.min(G[u][l] + G[l][v] , G[u][v]);
                }
            }
        }
    }
    return G;
}
```

Применение алгоритма Флойда – Уоршалла для поиска кратчайших путей

Помимо длин путей необходимо найти еще и *матрицу направлений* (аналог дерева предшествования для случая поиска путей из одной вершины).

P [u,v] = p, где p – первая вершина на кратчайшем пути из u в v.

Находим последовательность $P^{(0)}$, $P^{(1)}$,… $P^{(n)}$.

$P^{(0)}$ [u,v] = v, если G [u,v] < ∞ и не определено, если G [u,v] = ∞.

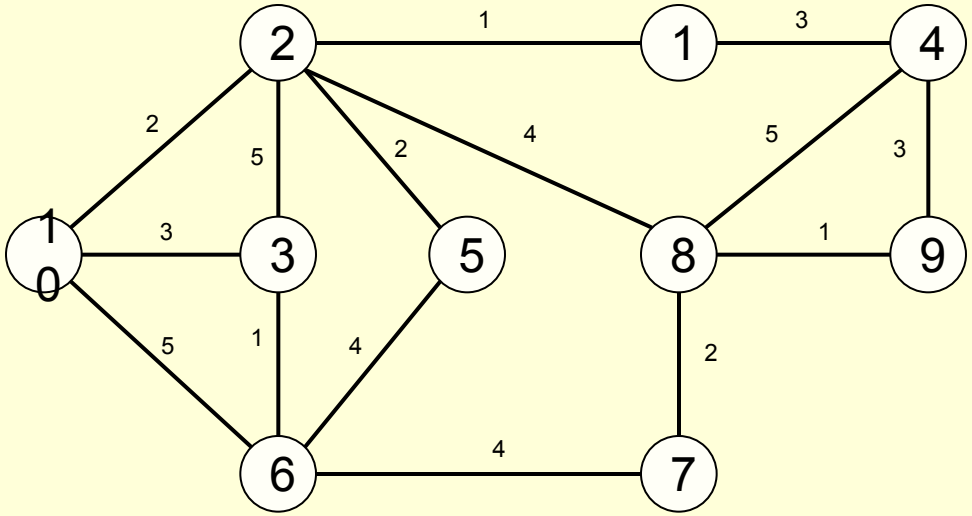
$P^{(\ell+1)}$ [u,v] = $P^{(\ell)}$ [u,v], если не было коррекции кратчайшего пути.

$P^{(\ell+1)}$ [u,v] = $P^{(\ell)}$ [u,ℓ], если была коррекция кратчайшего пути.

```
public static int[][] minPathsMatrix (double[][] G) {
    int n = G.length;
    int P[][] = new int[n][n];
    for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) {
        if (G[i][j] < Double.MAX_VALUE) P[i][j] = j;
    }
    for (int l = 0; l < n; l++) {
        // Формирование матриц G(l+1), P(l+1):
        for (int u = 0; u < n; u++) {
            if (G[u][l] < Double.MAX_VALUE) {
                for (int v = 0; v < n; v++) {
                    if (G[l][v] < Double.MAX_VALUE) {
                        if (G[u][l] + G[l][v] < G[u][v]) {
                            G[u][v] = G[u][l] + G[l][v];
                            P[u][v] = P[u][l];
    }   }   }   }   }   }
    return P;
}
```

Построение минимального скелета нагруженного графа. Алгоритм Прима.

Запускаем алгоритм обхода графа, начиная с произвольной вершины.
В качестве контейнера выбираем очередь с приоритетами. Приоритет – текущая величина найденного расстояния до уже построенной части опорного дерева.
Релаксации (как в Алгоритме Дейкстры) подвергаются прямые и обратные ребра.



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

В результате работы получили список ребер опорного дерева (скелета) вместе с нагрузками на все ребра.

Построение минимального скелета нагруженного графа. Алгоритм Крускала.