# Определения

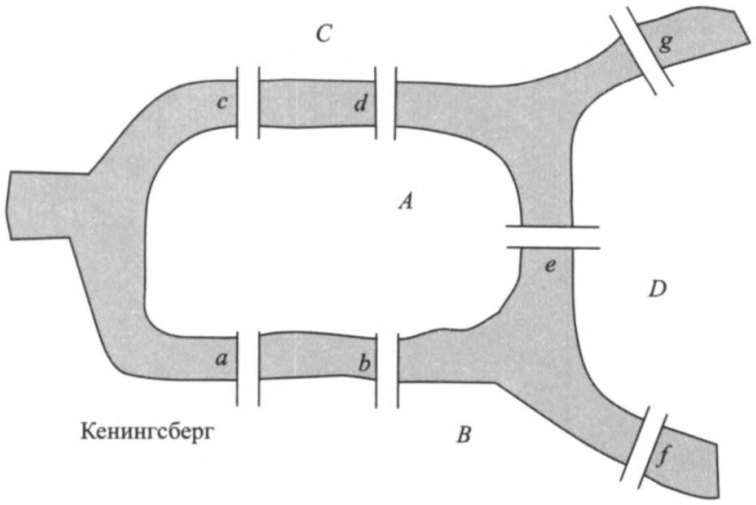
Теория графов – это раздел дискретной математики, изучающий объекты, представимые в виде отдельных элементов (вершин) и связей между ними (дуг, рёбер).

Теория графов берет начало с решения задачи о кенигсбергских мостах в 1736 году знаменитым математиком **Леонардом Эйлером** (1707-1783: родился в Швейцарии, жил и работал в России).

***Задача о кенигсбергских мостах.***


***В прусском городке Кенигсберг на реке Прегал семь мостов. Можно ли найти маршрут прогулки, который проходит ровно 1 раз по каждому из мостов и начинается и заканчивается в одном месте?***

Модель задачи — это *граф*, состоящий из множества *вершин* и множества *ребер*, соединяющих вершины. Вершины $A$, $B$, $C$ и $D$ символизируют берега реки и острова, а ребра $a$, $b$, $c$, $d$, $e$, $f$ и $g$ обозначают семь мостов (см. рис. 7.2). Искомый маршрут (если он существует) соответствует обходу ребер графа таким образом, что каждое из них проходится только один раз. Проход ребра, очевидно, соответствует пересечению реки по мосту.



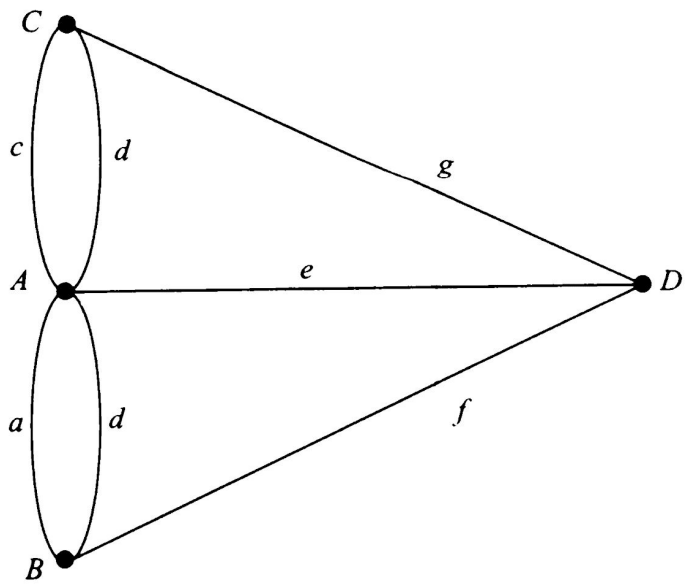
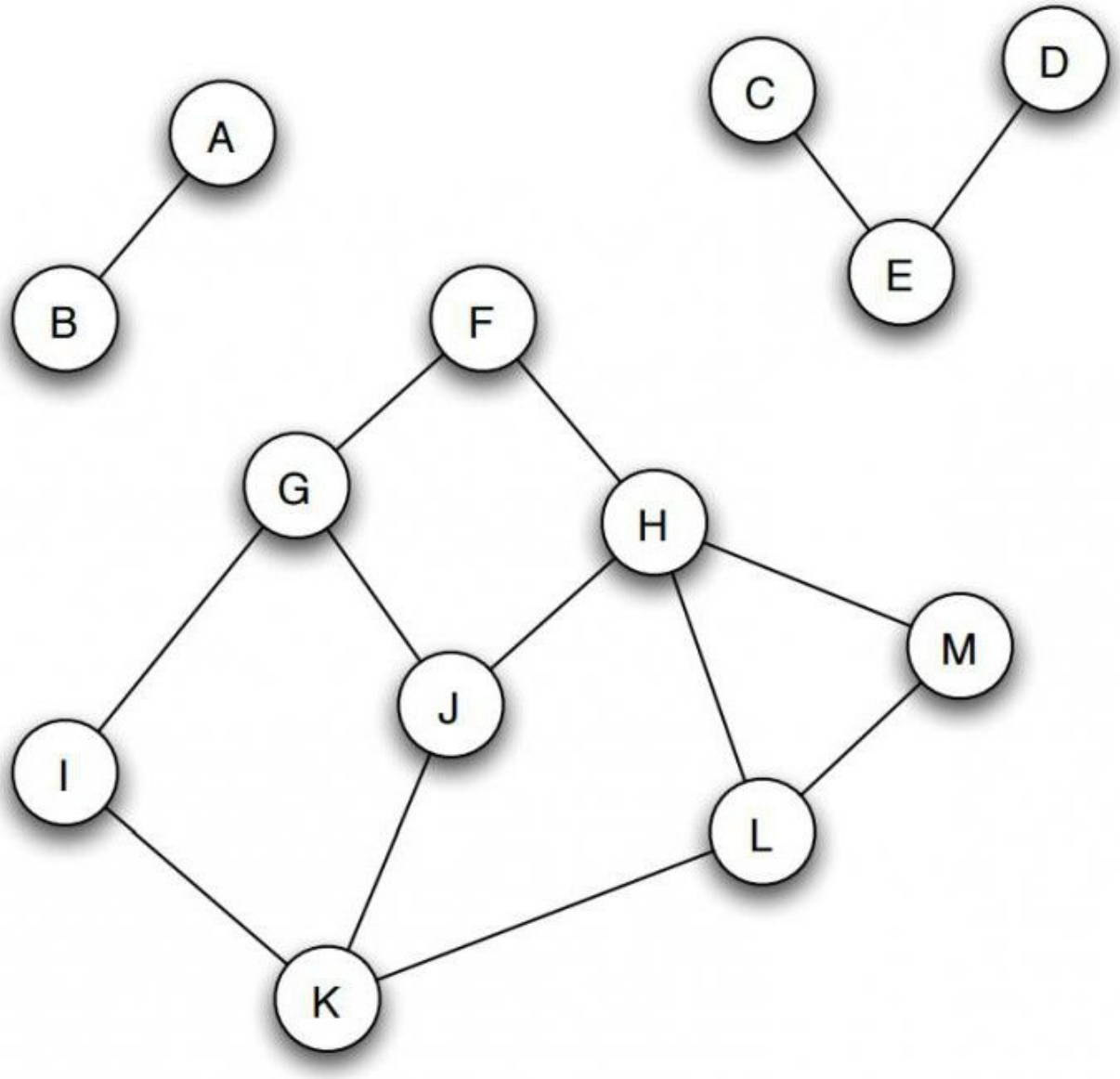
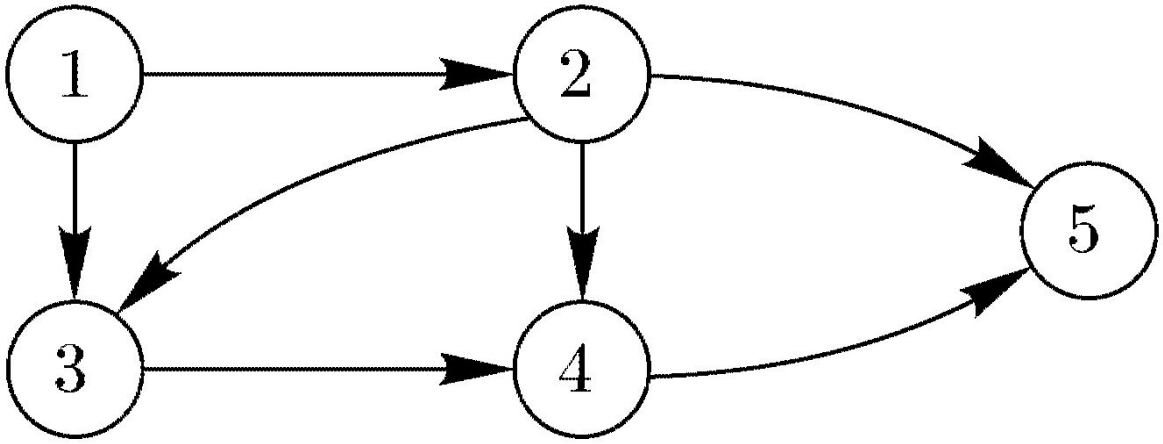



Рисунок 7.2. Модель задачи о мостах Кенигсберга

***Граф, в котором найдется маршрут, начинающийся и заканчивающийся в одной вершине, и проходящий по всем ребрам графа ровно один раз, называется Эйлеровым графом.***
***Последовательность вершин (может быть с повторением), через которые проходит искомый маршрут, как и сам маршрут, называется Эйлеровым циклом.***

***<u>Определение</u>*** ***Графом*** *G=G(V, E)* называется совокупность двух конечных множеств: V – называемого **множеством вершин** и множества E пар элементов из V, т.е. EÍV´V, называемого **множеством рёбер**, если пары не упорядочены, или **множеством дуг**, если пары упорядочены.
В первом случае граф *G(V,E)* называется **неориентированным**, во втором – **ориентированным.**

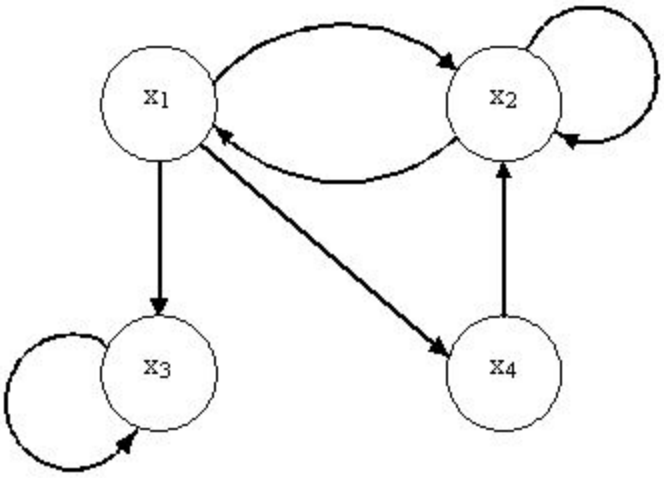

A
B
C
D
E
F
G
H
I
J
K
L
M



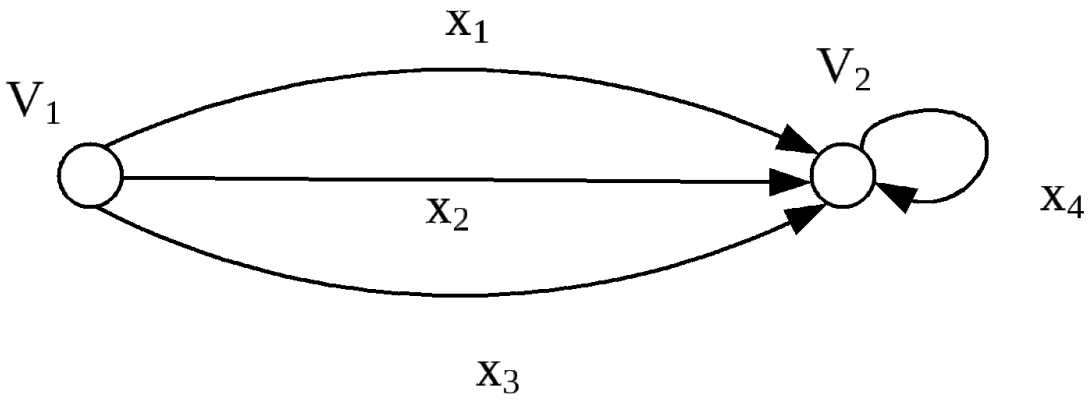

Две вершины $v_1, v_2$ называются **смежными**, если существует соединяющее их ребро. В этой ситуации каждая из вершин называется **инцидентной** соответствующему ребру.

Два различных ребра называются **смежными**, если они имеют общую вершину. В этой ситуации каждое из ребер называется **инцидентным** соответствующей вершине.

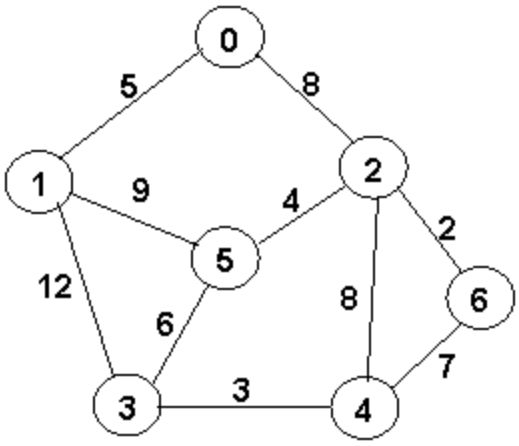
***Определение*** Если в ребре $e=(v_1,v_2)$ имеет место $v_1=v_2$ , то ребро e называется **петлёй**. Если в графе допускается наличие петель, то он называется ***графом с петлями*** или ***псевдографом***.




Если в графе допускается наличие более одного ребра между двумя вершинами, то он называется ***мультиграфом***.

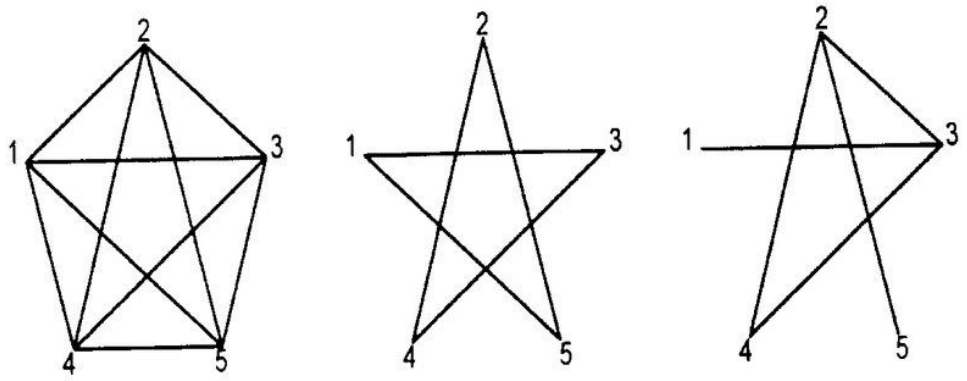



Приписывание каждому ребру графа некоего свойства, имеющего одинаковый для всех ребер семантический смысл, называется **взвешиванием** графа. Такой граф называется ***помеченным*** (или ***нагруженным***).




***Определение*** **Подграф** $G_1 = (V_1, E_1)$ **графа** $G = (V, E)$ **– граф, у которого все вершины и ребра удовлетворяют следующим соотношениям** $V_1 \subseteq V, E_1 \subseteq E$.

Примеры подграфов




24

# Структуры данных

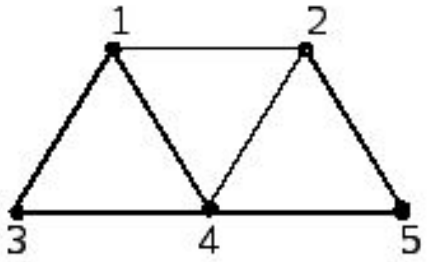
1. ***Перечисление множеств V*** и ***E.*** Применяется очень редко.

2. ***Матрица смежности*** - представляет граф в виде квадратной матрицы

**int A**[N, N]

отражающей смежность вершин

**A[i,j]** = *0*, если вершины **i** и **j** не смежны;
**A[i,j]** =*1*, если вершины **i** и **j** смежны;




| A | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 1 |
| 3 | 1 | 0 | 0 | 1 | 0 |
| 4 | 1 | 1 | 1 | 0 | 1 |
| 5 | 0 | 1 | 0 | 1 | 0 |

Рис. . Матрица смежности

*Свойства:*

а) матрица **A** симметрична, если ***G*** неориентированный;

б) если ***G*** взвешенный, вместо «**1**» можно проставить веса ребер;

в) степень вершины равна числу «**1**» в строке или в соответствующем столбце;

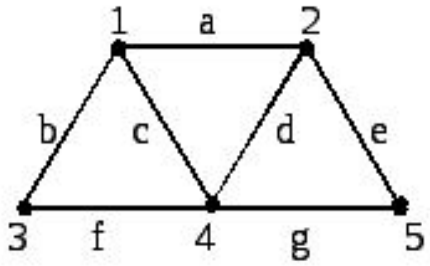
г) пространственная сложность этого способа $\boldsymbol{R(n,m) = O(n^2)}$, временные сложности сведены в таблицу:

| *Операция* | *Временная сложность* |
|---|---|
| Проверка смежности вершин x и y | O(1) |
| Перечисление всех вершин смежных с x | O(N) |
| Определение веса ребра (x, y) | O(1) |
| Определение веса вершины x | O(1) |
| Перечисление всех ребер (x, y) | $O(N^2)$ |
| Перечисление ребер, инцидентных вершине x | Номера ребер не хранятся |
| Перечисление вершин, инцидентных ребру s | Номера ребер не хранятся |

3. ***Матрица инциденций*** – представление с помощью матрицы

**Int X**[N, M]

(***n –*** число вершин графа, ***m*** – число ребер), отражающей инцидентность вершин и ребер, где для неориентированного графа (рис




| C | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 5 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |

Рис. . Граф и его матрица инциденций

**X [ i, j ]** = 1, если вершина $\mathbf{v}_i$ инцидентна ребру $\boldsymbol{e}_j$;
**X [ i, j ]** = 0, в противном случае;
*Свойства:*
а) такое представление полезно для задач, касающихся циклов;
б) обычно требует больше памяти, чем матрица смежности;
в) каждый столбец содержит ровно 2 единицы;
г) никакие два столбца не идентичны;
д) число единиц в ***i*** – строке равно степени вершины $\boldsymbol{v}_i$.
е) пространственная сложность этого способа ***R*(*n,m*) = *O*(*n*m*)**. Временные сложности сведены в таблицу:

| *Операция* | *Временная сложность* |
|---|---|
| Проверка смежности вершин x и y | O(M*N) |
| Перечисление всех вершин смежных с x | O(M*N) |
| Определение веса ребра (x, y) | O(M*N) |
| Определение веса вершины x | Вес вершины не хранится |

| | |
|---|---|
| Перечисление всех ребер (x, y) | O(M) |
| Перечисление ребер, инцидентных вершине x | O(M) |
| Перечисление вершин, инцидентных ребру s | O(N) |

Матрица инцидентности лучше всего подходит для операции *«перечисление ребер, инцидентных вершине x».*

4. ***Перечень ребер***. Это одномерный массив размером ***m***, содержащий список вершин смежных с данной:

```
class Edge
{
        V v1; V v2, // пара вершин, которые связывают ребро
        int w; // вес ребра
}
```

**Edge graph**[M]

Пространственная сложность этого способа ***O***(***m***). Временные сложности сведены в таблицу:

| *Операция* | *Временная сложность* |
|---|---|
| Проверка смежности вершин x и y | O(M) |
| Перечисление всех вершин смежных с x | O(M) |
| Определение веса ребра (x, y) | O(M) |
| Определение веса вершины x | Вес вершины не хранится |
| Перечисление всех ребер (x, y) | O(M) |
| Перечисление ребер инцидентных вершине x | O(M) |

| | |
|---|---|
| Перечисление вершин инцидентных ребру s | O(1) |

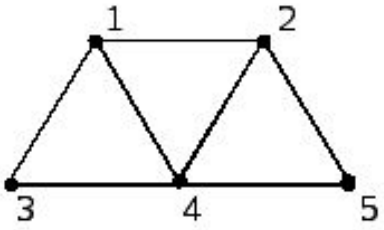
Как видно из таблицы, этот способ хранения графа особенно удобен, если главная операция, которой мы чаще всего будем пользоваться, это перечисление ребер или нахождение вершин и ребер, находящихся в отношениях инцидентности.

5. ***Векторы смежности.*** Форма представления – матрица, в ***i*** – строке которой содержится вектор, компоненты которого указывают на вершины, смежные с $v_i$. (рис. ).

*Свойства:*

а) размер матрицы (***n*** x $s_{max}$), где $s_{max}$ – максимальная степень вершины в ***G***;

б) удобно представлять граф, когда задача может быть решена за небольшое число просмотров каждого ребра в ***G.***



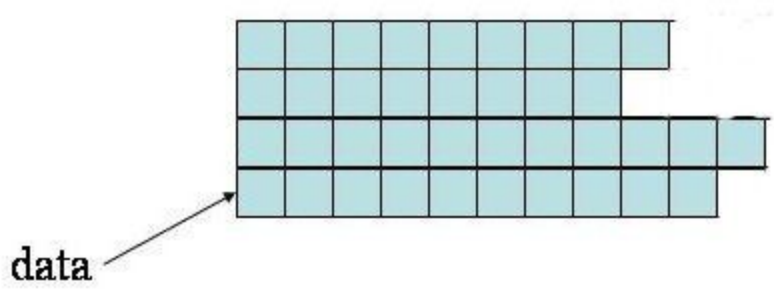

$$C = \begin{bmatrix} 1 & 2 & 3 & 4 & 0 \\ 2 & 1 & 4 & 5 & 0 \\ 3 & 1 & 4 & 0 & 0 \\ 4 & 1 & 2 & 3 & 5 \\ 5 & 2 & 4 & 0 & 0 \end{bmatrix}$$

Рис. . Представление графа в виде векторов смежности

Можно отметить, что данный способ по сравнению с другими матричными способами содержит меньше нулевых элементов и более компактно отображается в памяти.

6. ***Массивы смежности****.* Логично упаковать хвостовые нули и хранить не матрицу, а массив массивов

**int [][]A;**




7. ***Структура с оглавлением.*** Перечни узлов хранятся в общем массиве, а границы перечней указаны в оглавлении.

Элемент ***O*[*j*]** оглавления указывает, где в массиве начинается перечень узлов, смежных с ***j***-м узлом, одновременно являясь границей предыдущего перечня. Массив содержит 2***m*** элементов, оглавление — ***n*** + 1 элемент. Для изолированного узла начало перечня совпадает с началом следующего перечня, т. е. его перечень — пуст.

Перечни упорядочены и вот почему: проверка, существует ли ребро {***i,j***} является типичным действием; в неупорядоченном перечне ***i***-го узла эта проверка выполняется перебором, т. е. медленно, а в упорядоченном перечне возможен дихотомический (быстрый) поиск.


*Рис. .* Структура с оглавлением

Структура с оглавлением не проигрывает (по расходу памяти) матрице смежности — двумерному массиву даже в случае полного графа.

8. ***Списки смежности***. Представление графа с помощью списочной структуры, отражающей смежность вершин и состоящей из массива указателей на списки смежных вершин

```
Class Vertex {
        Info info; // информация об узле
        List<Vertex> vertices;
        }

List<Vertex> graph;
```

Список смежности содержит для каждой вершины ***v***, принадлежащей ***V***, список смежных ей вершин.

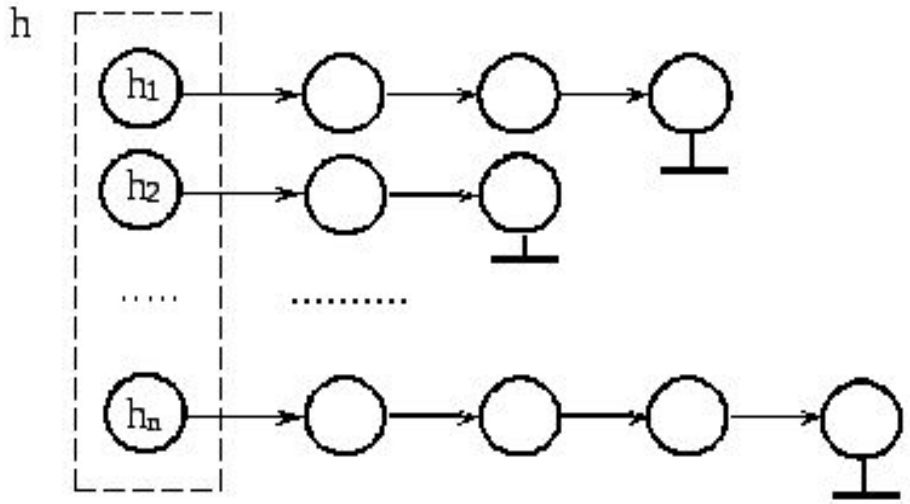



Рис. . Списки смежности

*Свойства:*

а) обладает преимуществами динамической структуры;

б) списки независимы, и это может в некоторых случаях быть недостатком, если задача заключается в нахождении некоторых путей в графе;

в) для неориентированных графов каждое ребро представлено в списках смежности дважды;
г) число ячеек памяти, необходимое для представления графа с помощью списков смежности, будет иметь порядок ***|V|+|E|.***
д) временные сложности сведены в таблицу:

| *Операция* | *Временная сложность* |
|---|---|
| Проверка смежности вершин x и y | O(N) |
| Перечисление всех вершин смежных с x | O(N) |
| Определение веса ребра (x, y) | O(N) |
| Определение веса вершины x | Вес вершины не хранится |
| Перечисление всех ребер (x, y) | O(M) |
| Перечисление ребер инцидентных вершине x | Номера ребер не хранятся |
| Перечисление вершин инцидентных ребру s | Номера ребер не хранятся |

Этот способ хранения лучше всех других подходит для операции «перечисление всех вершин смежных с x».

9. ***Список вершин и список ребер****.*

```
Class Node {
    Edge[] edges; // массив ребер
    }

Class Edge {
    Node node; // вершина, в которую можно попасть
                // по этому ребру
    double weight; // стоимость перехода
    }
```

**Node[] graph;**

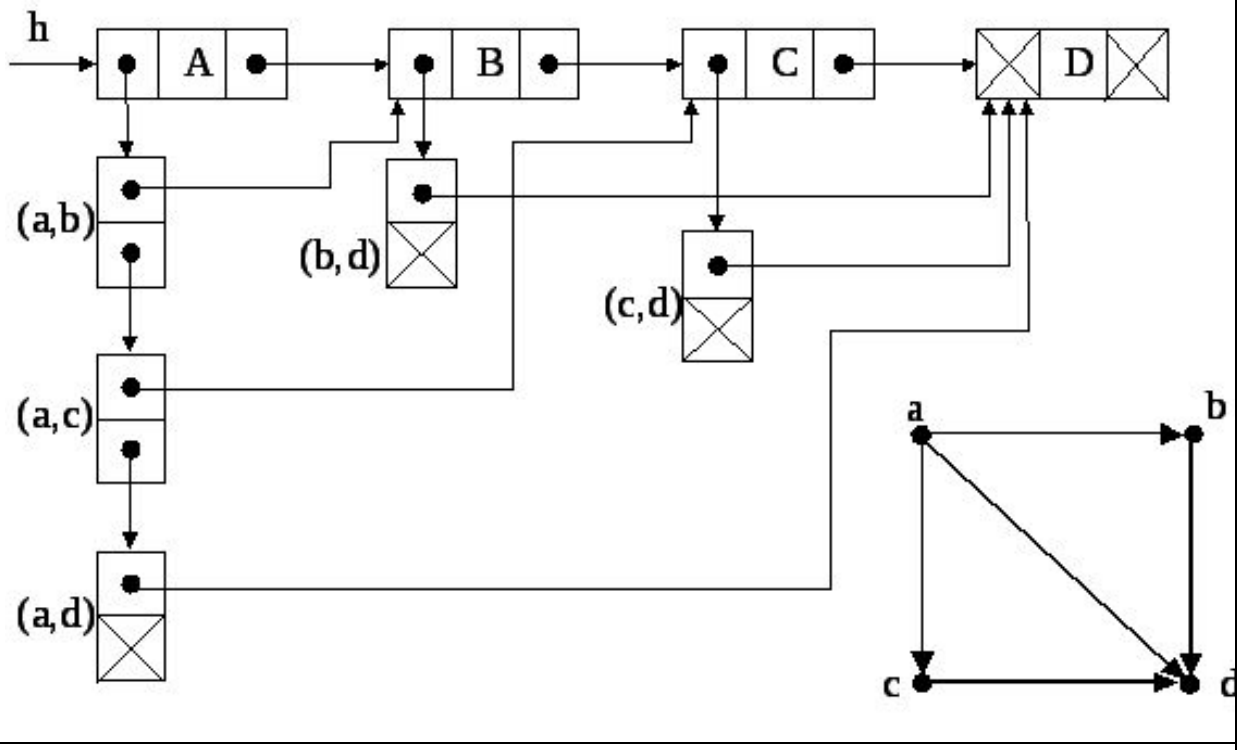



*Рис. .* Представление графа в виде списочной структуры
(**×** − означает ***NULL***)

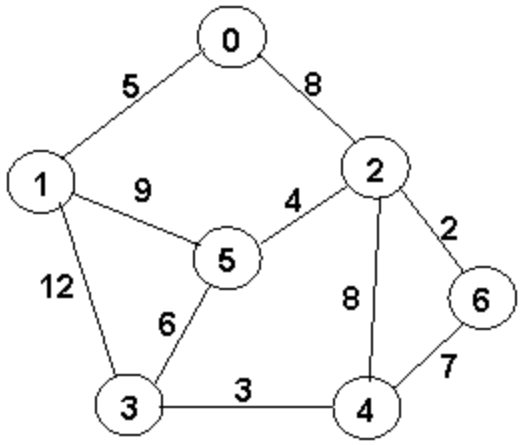
```
Class Node {
        Info info; // информация об узле
        in[] edges; // массив входящих ребер
        out[] edges; // массив исходящих ребер
        }

Class Edge {
        Info info; // информация об узле
        Node out; // вершина, из которой исходит ребро
        Node in; // вершина, в которую можно попасть
                        // по этому ребру
        double weight; // стоимость перехода
        }
```

```
Class Graph {
        Node [] nodes;
        Edge [] edges;
        }
```

Расширенная структура хранения данных о графе.

# Поиск в глубину в графе

*Depth-first search,* ***DFS -*** поиск в глубину




DSF(v)
пометить v как использованную;
путь.добавить(v);
**для** u **из** V
    **если** цель достигнута **то**
        **вернуть** истина
    **если** u не использовалась **то**
        **если** DFS(u) **то**
            **вернуть** истина
путь.удалить(верх)
вернуть ложь

«Правило левой руки» (идти, ведя левой рукой по стенке) будет поиском в глубину

Избавляемся от рекурсии

```
DFS (v)
    Пометить v как использованную
    Положить на стек(v);.
    пока стек не пуст
        u = верхняя вершина на стеке
        если u цель то
            вернуть истина
        если u не использовалась то
            пометить u как использованную
            для w инцидентной с u
                если w не использована то
                    положить на стек w

    вернуть ложь
```

***Цель обхода?***
***Как пометить как использованную?***

## Применение

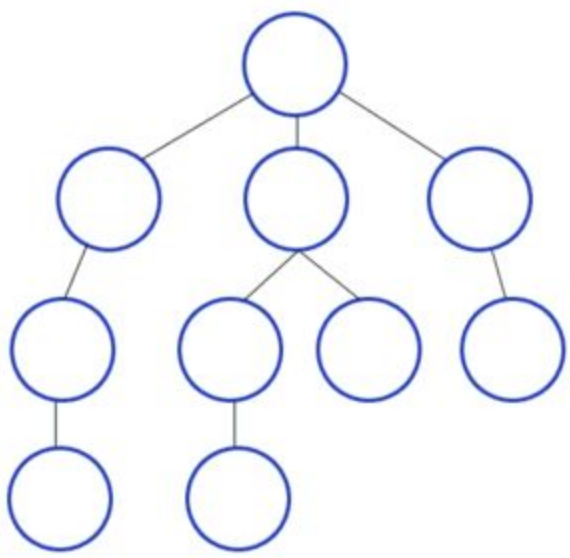
Поиск в глубину ограниченно применяется как собственно *поиск*, чаще всего на древовидных структурах: когда расстояние между точками малó, поиск в глубину может «плутать» где-то далеко.

Зато поиск в глубину — хороший инструмент для исследования топологических свойств графов. Например:

- В алгоритмах, решающих задачу перебором
- В качестве подпрограммы в алгоритмах поиска одно- и двусвязных компонент.
- В топологической сортировке.
- Для поиска точек сочленения, мостов.
- В различных расчётах на графах. Например, как часть алгоритма Диница поиска максимального потока.

# Поиск в ширину в графе

*breadth-first search*, **BFS**



```
BFS(v)
очередь.добавить(v);
пока очередь не пусто
        v = очереди.извлечь();
        v.ПометитьКакИспользованный();
        если v цель то
                вернуть ОК
        иначе
                для u инцендентных v
                        если u.НеИспользован и не очередь.содержит(u) то
                                очередь.добавить(u)
вернуть не найдено
```

```
BFS(v)
очередь.добавить(v);
пока очередь не пусто
    v = очереди.извлечь();
    если v не использован то
        v.ПометитьКакИспользованный()
        если v цель то
            вернуть ОК
        иначе
            для u инцендентных v
                если u.НеИспользован то
                    очередь.добавить(u)
вернуть не найдено
```

## Неинформи́рованный по́иск

**Неинформи́рованный по́иск** (также **слепой поиск**, **метод грубой силы**, англ. *uninformed search, blind search, brute-force search*) — стратегия поиска решений в пространстве состояний, в которой не используется дополнительная информация о состояниях, кроме той, которая представлена в определении задачи. Всё, на что способен метод неинформированного поиска, — вырабатывать преемников и отличать целевое состояние от нецелевого.

## Поиск по критерию стоимости

**Поиск по критерию стоимости** (метод равных цен, uniform-cost search, **UCS**) — обобщение алгоритма поиска в ширину, учитывающее стоимости действий (рёбер графа состояний). Поиск по критерию стоимости развёртывает узлы в порядке возрастания стоимости кратчайшего пути от корневого узла. На каждом шаге алгоритма развёртывается узел с наименьшей стоимостью $g(n)$. Узлы хранятся в отсортированном списке List<Key, Value>.

```
USC(v)
сум_вес = 0
очередь.добавить(сум_вес, v);
пока очередь не пусто
    v, сум_вес = очереди.извлечь();
    если v не использован то
```

```
            v.ПометитьКакИспользованный()
            если v цель то
                вернуть ОК
            иначе
                сум_вес += v.вес
                для u инцендентных v
                    если u.НеИспользован то
                        очередь.добавить(сум_вес+u.вес, u)
вернуть не найдено
```

## Поиск с ограничением глубины

**Поиск с ограничением глубины** (depth-limited search, **DLS**) — вариант поиска в глубину, в котором применяется заранее определённый предел глубины *l*, что позволяет решить проблему бесконечного пути.

Поиск с ограничением глубины не является полным, так как при $l < d$ цель не будет найдена, и не является оптимальным при $l > d$. Его временная сложность равна $O(b^l)$, а пространственная сложность — $O(b \cdot l)$.

Поиск с ограничением глубины применяется в алгоритме поиска с итеративным углублением.

```
DLS(v, предел)
пометить v как использованную;
для u из V
    если цель достигнута то
        вернуть истина
    если уровень >= предел то
        вернуть ложь
    если u не использовалась то
        если DFS(u, уровень+1, предел) то
            вернуть истина
Вернуть ложь
```

## Нерекурсивный вариант

```
DLS (v, предел)
    уровень = 0
```

```
Пометить v как использованную
Положить на стек(v, уровень);.
пока стек не пуст
        u = верхняя вершина на стеке
        уровень = получить уровень
        если u цель то
                вернуть истина
        уровень++
        если уровень > предел то
                вернуть ложь
        если u не использовалась то
                пометить u как использованную (уровень)
                для w инцидентной с u
                        если w не использована то
                                положить на стек w
вернуть ложь
```

## Поиск в глубину с итеративным углублением

**Поиск в глубину с итеративным углублением** (iterative-deepening depth-first search, **IDDFS**, **DFID**) — стратегия, которая позволяет найти наилучший предел глубины поиска DLS. Это достигается путём пошагового увеличения предела *l* до тех пор, пока не будет найдена цель.

В поиске с итеративным углублением сочетаются преимущества поиска в глубину (пространственная сложность $O(b \cdot l)$) и поиска в ширину (полнота и оптимальность при конечном $b$ и неотрицательных весах рёбер).

Временная сложность алгоритма имеет порядок $O(b^l)$

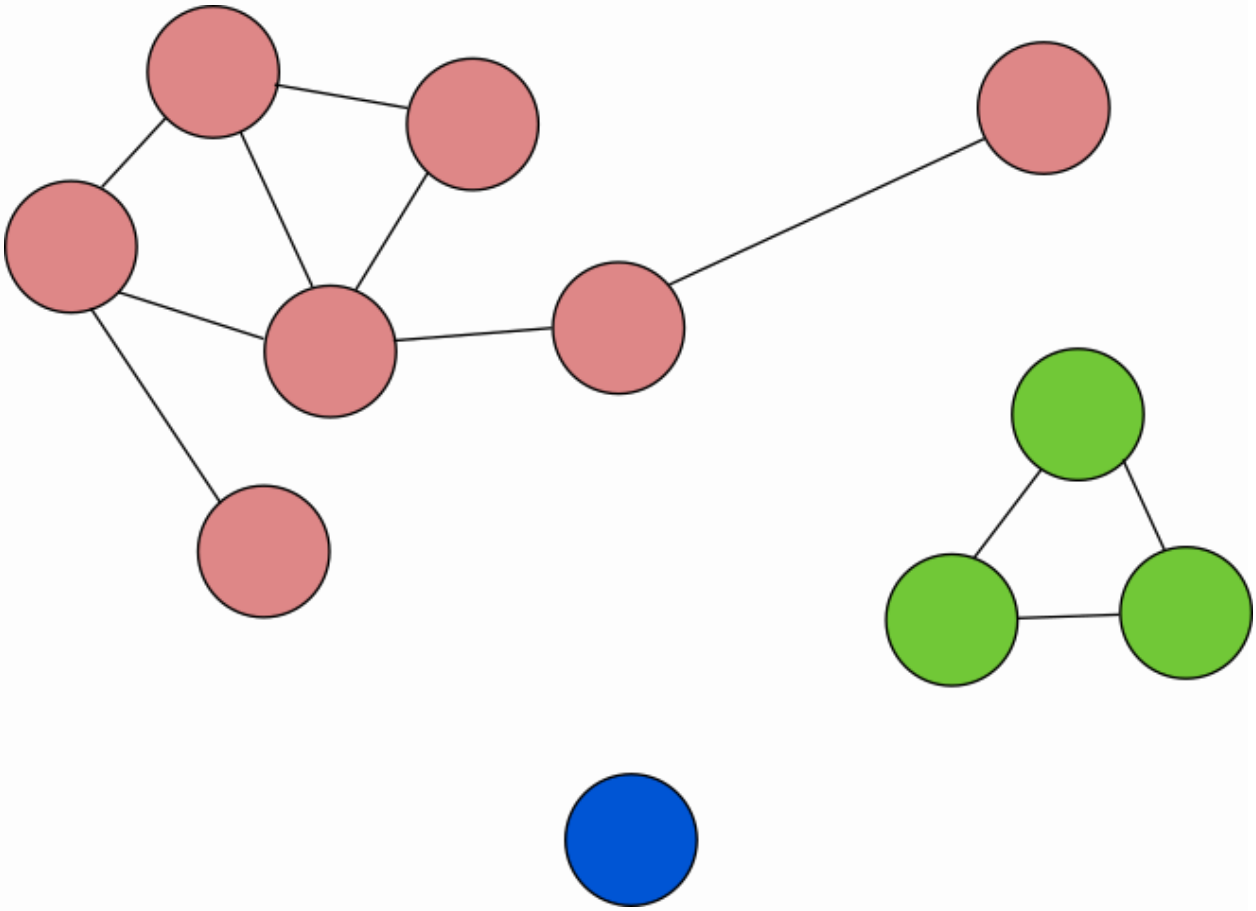
```
IDDFS(v)
предел = 1
пока не DLS(v, предел)
        предел ++
```

**Что не так в этом алгоритме?**
**Как сделать так, чтобы начальные уровни повторно не проходились?**

# Поиск компонент сильной связности

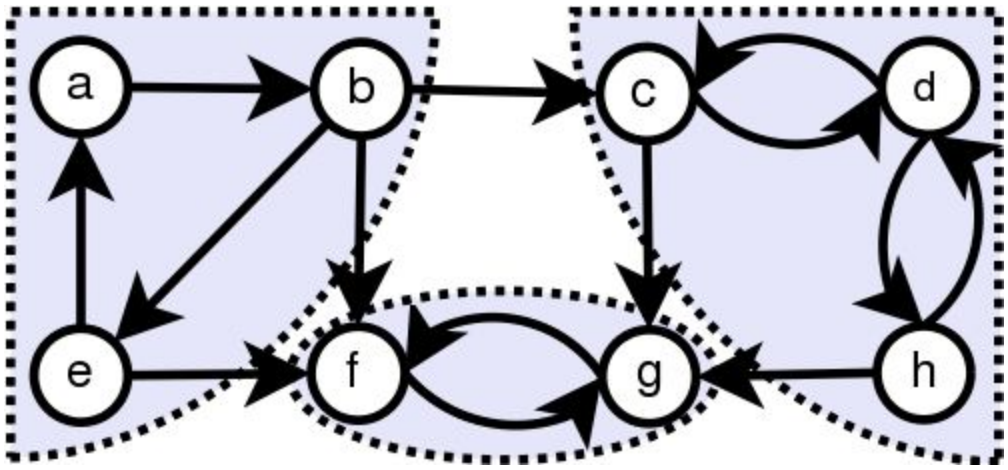
## Поиск компонент сильной связности

Понятие компоненты связности вытекает из понятия связности графа. Попросту говоря, компонента связности - часть графа (подграф), являющаяся связной. Формально, компонента связности - набор вершин графа, между любой парой которых существует путь.



Общее понятие связности распространяется только на неориентированные графы. Для описания ориентированных графов используются понятия *сильной* и *слабой* связности.

Орграф называется ***сильно связным*** (англ. *strongly connected*), если любые две его вершины сильно связаны. Две вершины s и t любого графа сильно связаны, если существует ориентированный путь из s в t и ориентированный путь из t в s. **Компонентами сильной связности** орграфа называются его максимальные по включению сильно связные подграфы. **Областью сильной связности** называется множество вершин компоненты сильной связности.




Отношение R(v,u) называется отношением **слабой связности** *(англ. weak connected)*, если вершины u и v связаны в неориентированном графе G′G′, полученном из графа G удалением ориентации с рёбер.

## Алгоритмы

Простейший алгоритм решения задачи о поиске сильно связных компонент в орграфе работает следующим образом:

1. При помощи транзитивного замыкания проверяем, достижима ли t из s, и s из t, для всех пар s и t.
2. Затем определяем такой неориентированный граф, в котором для каждой такой пары содержится ребро.
3. Поиск компонент связности такого неориентированного графа даст нам компоненты сильной связности нашего орграфа.

Очевидно основное время работы данного алгоритма приходится на реализацию транзитивного замыкания.

Также существует три алгоритма, решающих данную задачу за линейное время, то есть в V раз быстрее, чем приведенный выше алгоритм. Это алгоритмы

- Косарайю,
- Габова и Тарьяна.

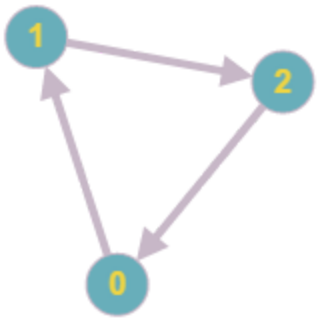
## Алгоритм Косарайю

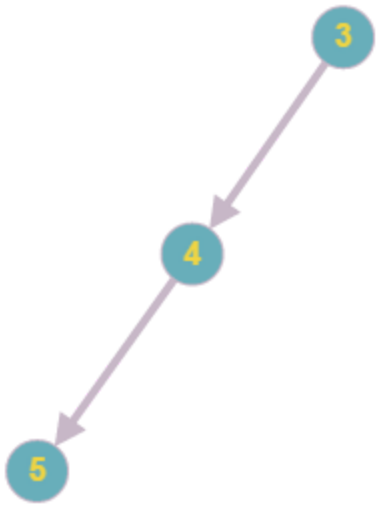
Самбасив Рао Косарайю – американский ученый с индийскими корнями, профессор информатики в Университете Джона Хопкинса. В 1978 написал работу в
которой описал такой метод, как метод эффективного вычисления компонент сильной связности в орграфе. Позднее данный метод стал известен как «Алгоритм
Косарайю», название закрепилось. В 1983 году алгоритм был опубликован в статье
А. Ахо, Д. Хопкрофта «Структуры данных и алгоритмы» ( Data Structures and Algorithms).

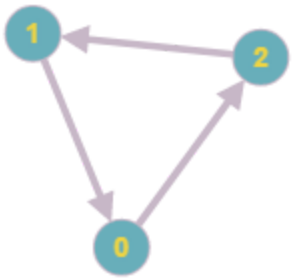
1. Построить граф H с обратными (инвертированными) рёбрами
2. Выполнить в H поиск в глубину и найти f[u] — порядок окончания обработки вершины u
3. Выполнить поиск в глубину в G, перебирая вершины во внешнем цикле в порядке убывания f[u]

Полученные на 3-ем этапе деревья поиска в глубину будут являться компонентами сильной связности графа G.

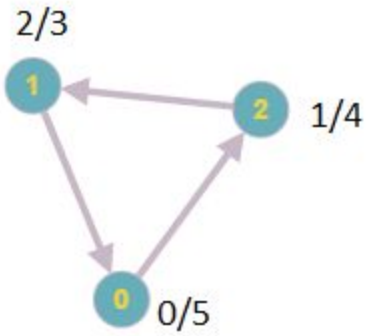
Так как компоненты сильной связности G и H графа совпадают, то первый поиск в глубину для нахождения f[u] можно выполнить на графе G, а второй — на H.

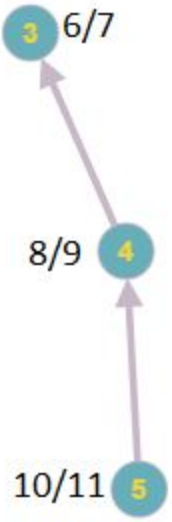

1
2
0


3
4
5

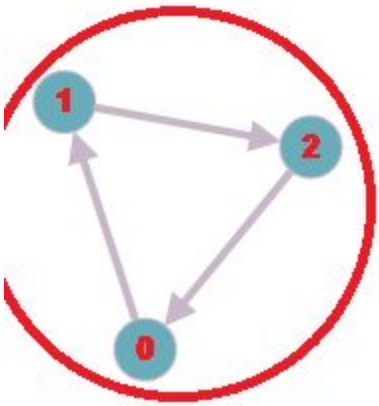

1
2
0

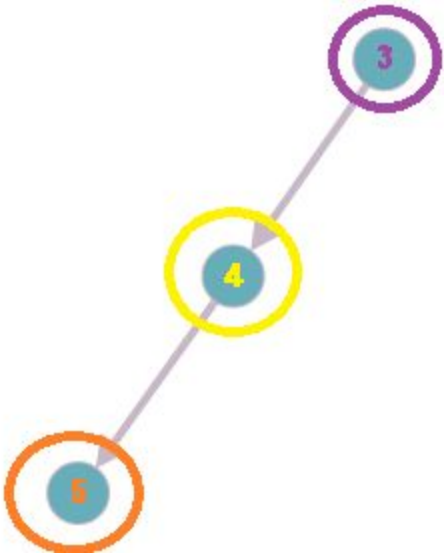

3
4
5




| 1 | 2 | 0 | 3 | 4 | 5 |
|---|---|---|---|---|---|



```
Косарайю():
        //1
        формируем G и H,
        инициализируем очередь, компонент[], посещали[]
        // 2
        для u из H.V
            если не посещали[u] то
                H.DFS1(u)
       // 3
        индекс_компонента = 1
        для u из очередь в обратном порядке
            если компонент[u] < 0 то
                G.DFS2(u)
                индекс_компонента++

DFS1(v):
        посещали[v] = истина
        для u из связности(v)
            если не посещали[u] то
                DFS1(u)
        очередь.добавить(v)

   DFS2(v):
        компонент[v] = индекс_компонента
        для u из связности(v)
            если компонент[u] < 0 то
                DFS2(u)
```