Сириус
IT-Колледж

Лекция 6.
Графы

Способы хранения: матрица смежности, списки смежности, матрица инцидентности. Поиск в глубину в неориентированных графах, выделение компонент связности. Поиск в глубину в ориентированных графах: ориентированные ациклические графы, топологическая сортировка вершин.

Сириус IT-Колледж

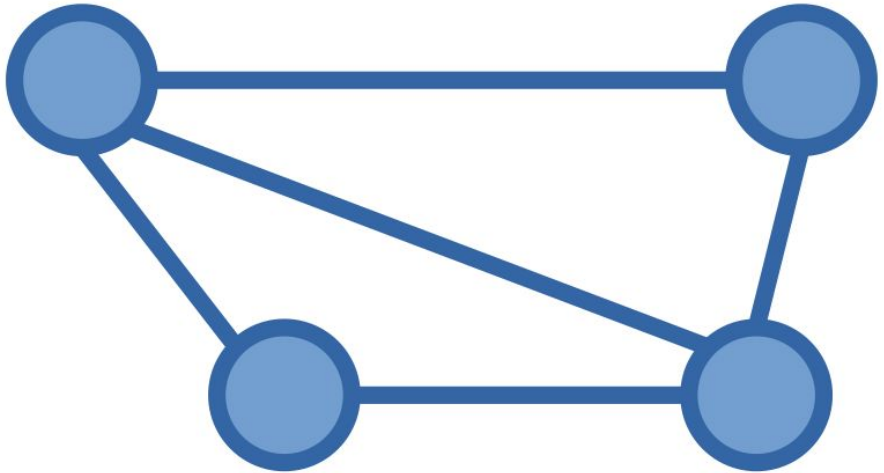
Графы

Граф — математическая абстракция реальной системы любой природы, объекты которой обладают парными связями. Граф как математический объект есть совокупность двух множеств — множества самих объектов, называемого множеством вершин, и множества их парных связей, называемого множеством рёбер. Элемент множества рёбер есть пара элементов множества вершин.

Сириус IT-Колледж

Графы

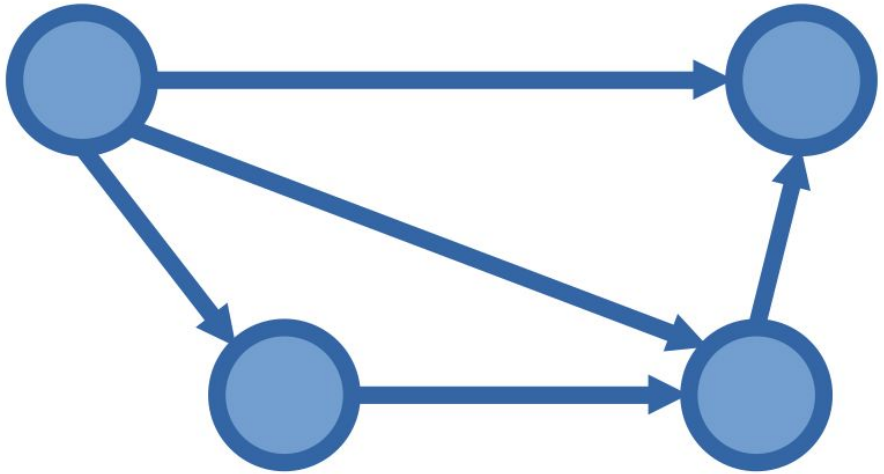
Простой (неориентированный) граф ***G(V,E)*** есть совокупность двух множеств – непустого множества ***V*** и множества ***E*** неупорядоченных пар различных элементов множества ***V***. Множество ***V*** называется множеством вершин, множество ***E*** называется множеством рёбер.



Сириус IT-Колледж

Графы

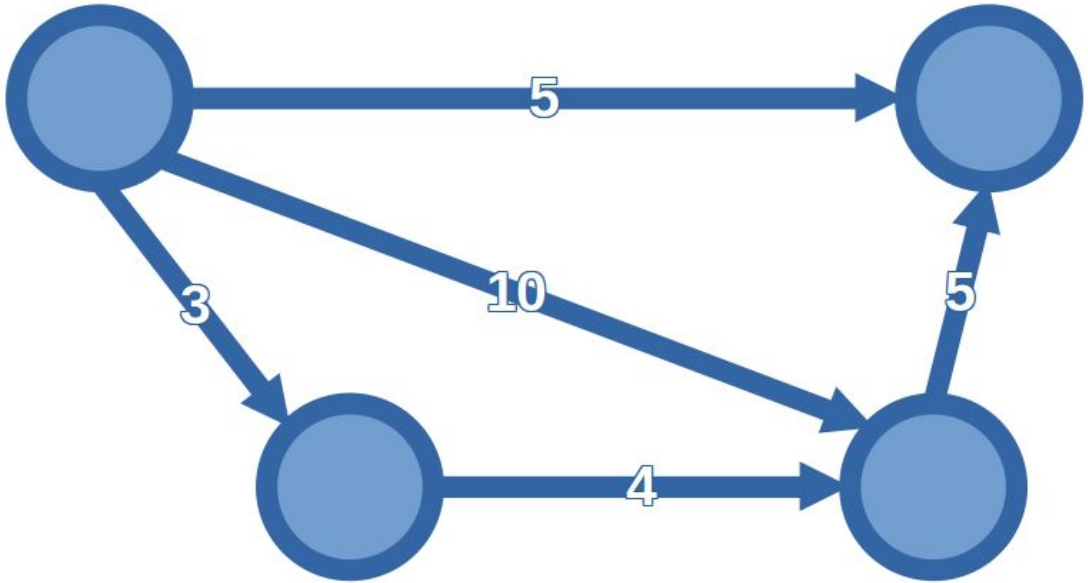
Ориентированный граф (орграф) ***G(V,E)*** есть совокупность двух множеств – непустого множества ***V*** и множества ***E*** дуг или упорядоченных пар различных элементов множества ***V***.



Сириус IT-Колледж

Графы

Взвешенный граф — граф, каждому ребру которого поставлено в соответствие некое значение (вес ребра).



Сириус IT-Колледж

Графы

Путь - любая последовательность вершин, в которой каждые две соседние вершины соединены ребром (A → C → B → G, A → D → B → D → B).

Длина пути - количество рёбер в нём (3 и 4 соответственно для примеров выше).

Цикл - путь, у которого начальная и конечная вершина совпадают (A → C → B → D → A, F → E → F).

Простой путь - путь, в котором нет повторяющихся рёбер.

Простой цикл - цикл, который является простым путём.

Сириус
IT-Колледж

Способы хранения графов

Сириус IT-Колледж

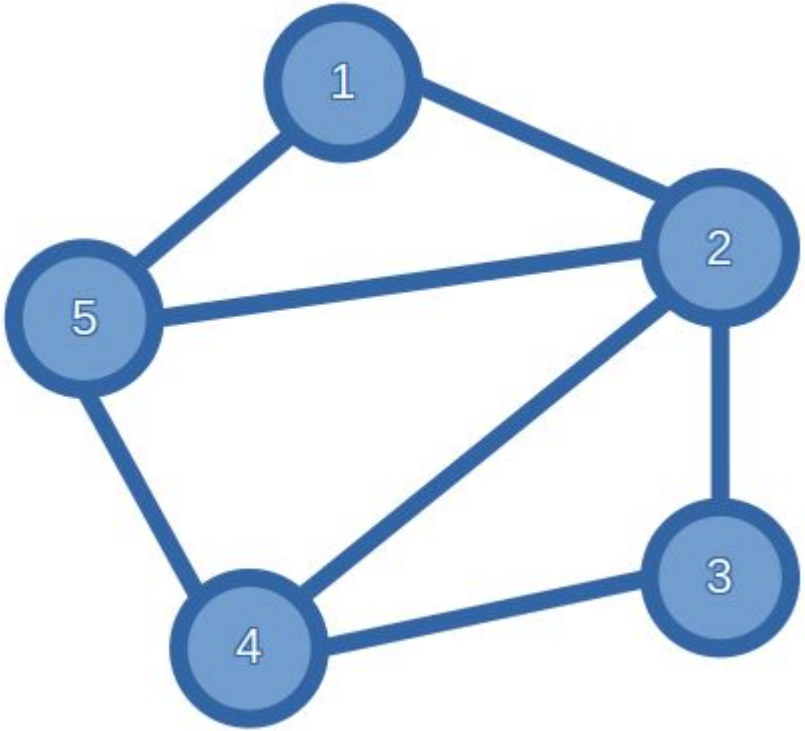
Матрица смежности

Матрицей смежности $A=||\alpha i,j||$ невзвешенного графа $G=(V,E)$ называется матрица $A_{[V\times V]}$, в которой $\alpha_{i,j}$ — количество рёбер, соединяющих вершины v_i и v_j, причём при $i=j$ каждую петлю учитываем дважды, если граф не является ориентированным, и один раз, если граф ориентирован.

Матрицей смежности $A=||\alpha i,j||$ взвешенного графа $G=(V,E)$ называется матрица $A_{[V\times V]}$, в которой $\alpha_{i,j}$ — вес ребра, соединяющего вершины v_i и v_j.

Сириус
IT-Колледж

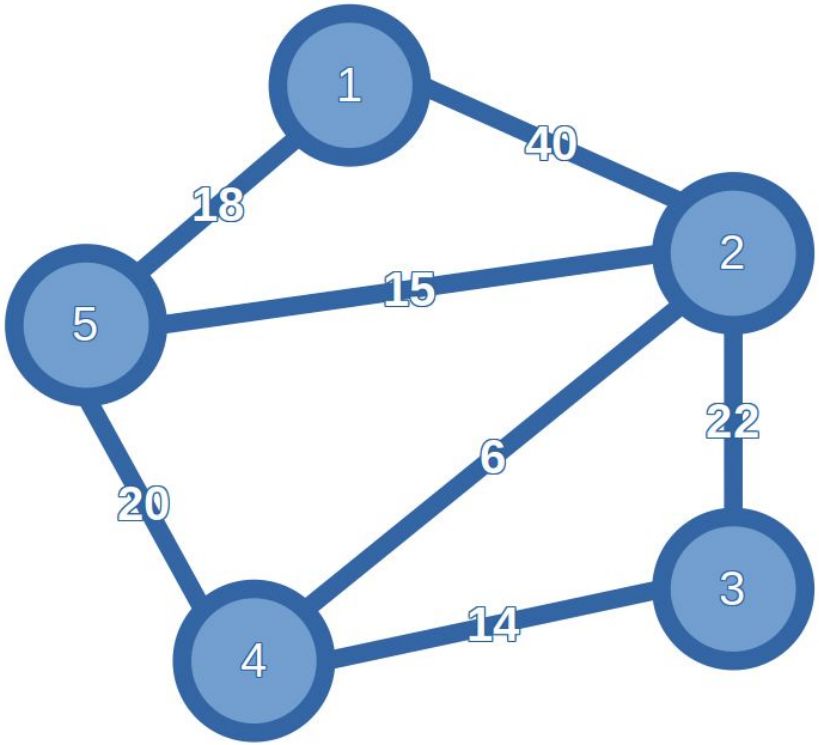
Матрица смежности



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Сириус
IT-Колледж

Матрица смежности



$$\begin{pmatrix} 0 & 40 & \infty & \infty & 18 \\ 40 & 0 & 22 & 6 & 15 \\ \infty & 22 & 0 & 14 & \infty \\ \infty & 6 & 14 & 0 & 20 \\ 18 & 15 & \infty & 20 & 0 \end{pmatrix}$$

Сириус IT-Колледж

Матрица смежности

```
class Graph(object):

  def __init__(self, size):
    self.adjMatrix = [[0] * size for i in
range(size)]
    self.size = size

  def add_edge(self, v1, v2):
    if v1 == v2:
      print(f"Та же вершина {v1} и {v2}")
    self.adjMatrix[v1][v2] = 1
    self.adjMatrix[v2][v1] = 1

  def __len__(self):
    return self.size
```

```
  def remove_edge(self, v1, v2):
    if self.adjMatrix[v1][v2] == 0:
      print(f"Нет ребра между {v1} и
{v2}")
      return
    self.adjMatrix[v1][v2] = 0
    self.adjMatrix[v2][v1] = 0

  def print_matrix(self):
    for row in self.adjMatrix:
      for val in row:
        print(f'{val:4d}')
      print
```

Сириус IT-Колледж

Матрица смежности

Плюсы:

- Добавление ребра, удаление ребра, проверка наличия ребра между вершинами *i* и *j* за ***O(n)***
- Лучший выбор для плотных графов. В случае разреженного графа и матрицы смежности можно использовать структуры данных для разреженных матриц
- Возможность выполнения операция на GPU

Минусы:

- Требуется VxV памяти для хранения. Чаще всего графы не имеют большого количества связей и лучшим выбором будут списки смежности.
- Выполнения операций по нахождению внешних и внутренних ребер требует большего времени

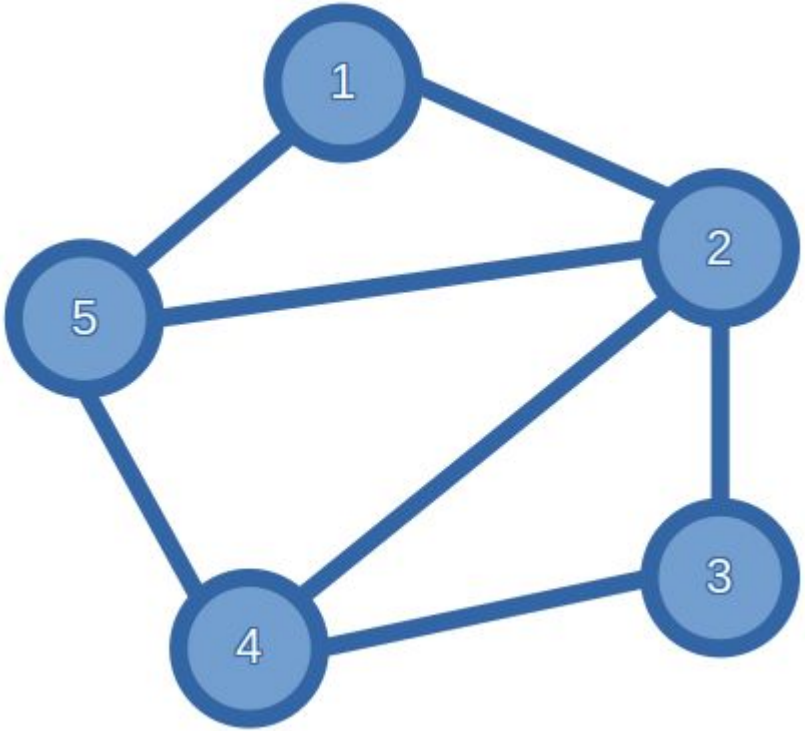
Сириус IT-Колледж

Списки смежности

Список смежности — один из способов представления графа в виде коллекции списков вершин. Каждой вершине графа соответствует список, состоящий из «соседей» этой вершины.

Сириус IT-Колледж

Списки смежности



1	смежно к	2, 5
2	смежно к	1, 3, 4, 5
3	смежно к	2, 4
4	смежно к	2, 3, 5
5	смежно к	1, 2, 4

Сириус IT-Колледж

Списки смежности

```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
```

Сириус IT-Колледж

Списки смежности

```
class AdjNode:
  def __init__(self, value):
    self.vertex = value
    self.next = None

class Graph:
  def __init__(self, num):
    self.V = num
    self.graph = [None] * self.V

  def add_edge(self, s, d):
    node = AdjNode(d)
    node.next = self.graph[s]
    self.graph[s] = node
    node = AdjNode(s)
    node.next = self.graph[d]
    self.graph[d] = node
```

```
  def print_agraph(self):
    for i in range(self.V):
      print("Вершина " + str(i) + ":",
end="")
      temp = self.graph[i]
      while temp:
        print(f" -> {temp.vertex}",
end="")
        temp = temp.next
      print(" \n")
```

Сириус IT-Колледж

Списки смежности

Плюсы:

- Эффективны в плане потребления памяти, так как хранится только информация о ребрах. Для больших разреженных графов могут сберечь большой объем памяти
- Быстрый поиск смежных вершин

Минусы:

- Построение списка смежности не быстрее построения матрицы смежности, так как необходимо так же проверить и найти все узлы

Сириус IT-Колледж

Списки смежности

Операция	Список смежности	Матрица смежности
Проверка на наличие ребра (x,y)	$O(\|V\|)$	$O(1)$
Определение степени вершины	$O(1)$	$O(\|V\|)$
Использование памяти для разреженных графов	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$
Вставка/удаление грани	$O(1)$	$O(d)$
Обход графа	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$

Сириус IT-Колледж

Матрица инцидентности

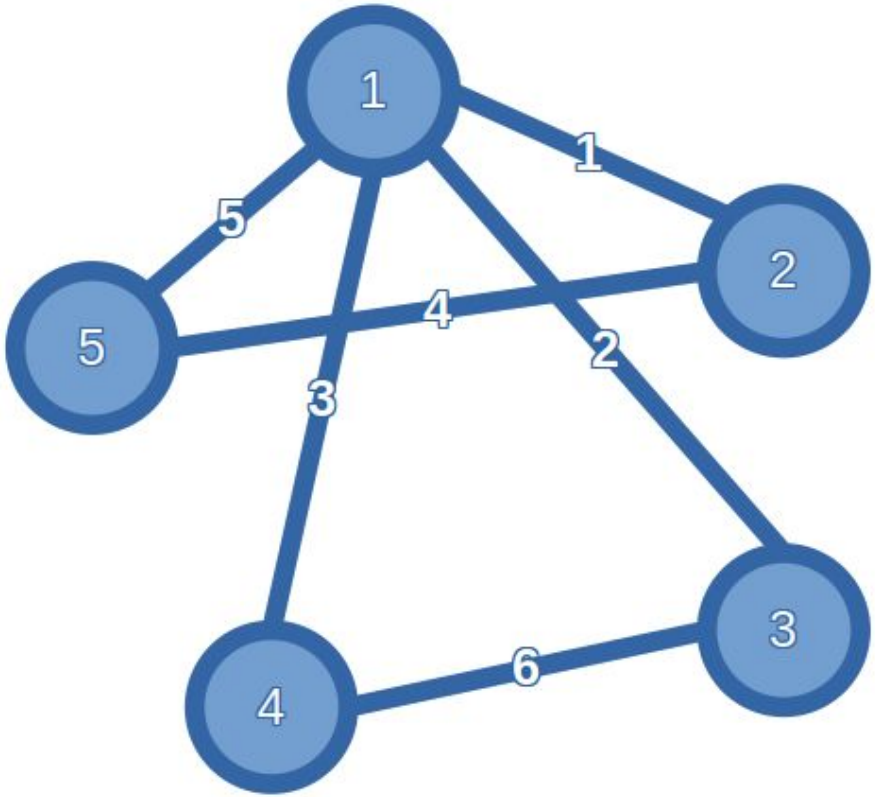
Вершина ***v* инцидентна** ребру ***e***, если $v \in e$; тогда еще говорят, что ***e*** есть ребро при ***v***;

Матрицей инцидентности (инциденций) неориентированного графа называется матрица $I(|V|\times|E|)$, для которой $I_{i,j}=1$, если вершина v_i инцидентна ребру e_j, в противном случае $I_{i,j}=0$.

Матрицей инцидентности (инциденций) ориентированного графа называется матрица $I(|V|\times|E|)$, для которой $I_{i,j}=1$, если вершина v_i является началом дуги e_j, $I_{i,j}=-1$, если v_i является концом дуги e_j, в остальных случаях $I_{i,j}=0$.

Сириус
IT-Колледж

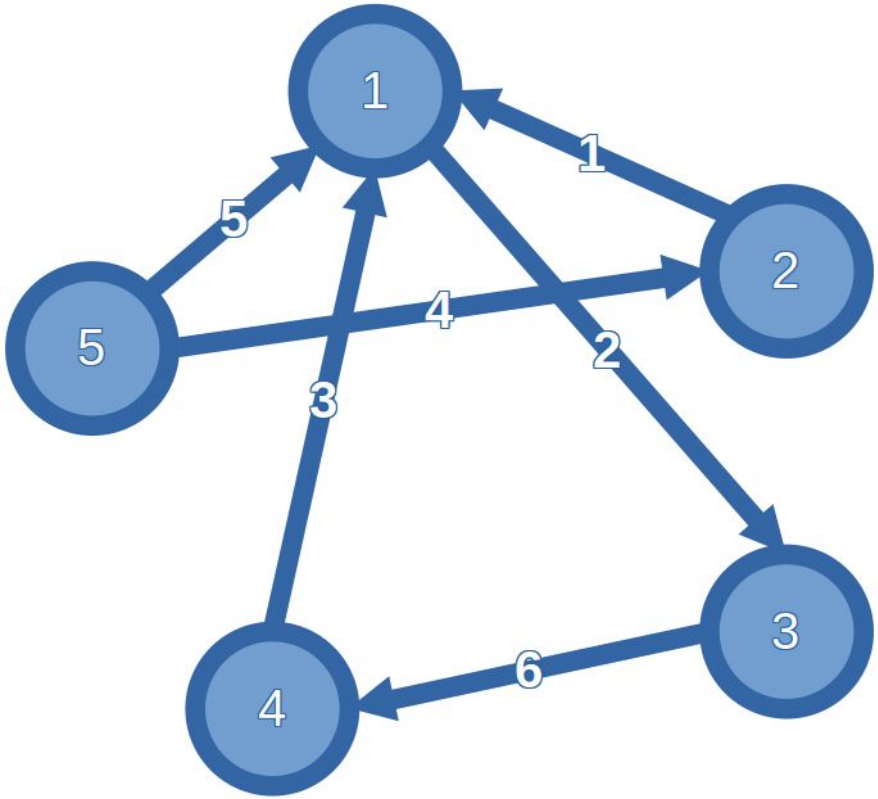
Матрица инцидентности



$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Сириус
IT-Колледж

Матрица инцидентности



$$\begin{pmatrix} -1 & 1 & -1 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Сириус IT-Колледж

Списки смежности

```
class Graph(object):
  def __init__(self, size):
    self.incMatrix = []
    self.size = size

  def add_edge(self, v1, v2):
    if v1 == v2:
      print(f"Та же вершина {v1} и {v2}")
      return
    newEdge = [1 if i == v1 or i == v2
else 0 for i in range(self.size)]
    self.incMatrix.append(newEdge)

  def __len__(self):
    return self.size
```

```
  def remove_edge(self, v1, v2):
    for e in range(len(self.incMatrix)):
      if self.incMatrix[e][v1] and
self.incMatrix[e][v2]:
        self.incMatrix.pop(e)
        return
    print(f"Нет ребра между {v1} и {v2}")
```

Сириус
IT-Колледж

Поиск в глубину в неориентированных графах

Сириус IT-Колледж

Поиск в глубину в неориентированных графах

Обход в глубину (поиск в глубину, Depth-First Search, DFS) — один из основных методов обхода графа, часто используемый для проверки связности, поиска цикла и компонент сильной связности и для топологической сортировки.

Общая идея алгоритма состоит в следующем: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них.

Сириус IT-Колледж

Поиск в глубину в неориентированных графах

Пошаговое представление:

1. Выбираем любую вершину из еще не пройденных, обозначим ее как ***u***.
2. Запускаем процедуру ***dfs(u)***
   a. Помечаем вершину ***u*** как пройденную
   b. Для каждой не пройденной смежной с ***u*** вершиной (назовем ее ***v***) запускаем ***dfs(v)***
3. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

Сириус IT-Колледж

Поиск в глубину в неориентированных графах

```
from collections import defaultdict


class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end=' ')

        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    def DFS(self, v):

        visited = set()

        self.DFSUtil(v, visited)
```

Сириус IT-Колледж

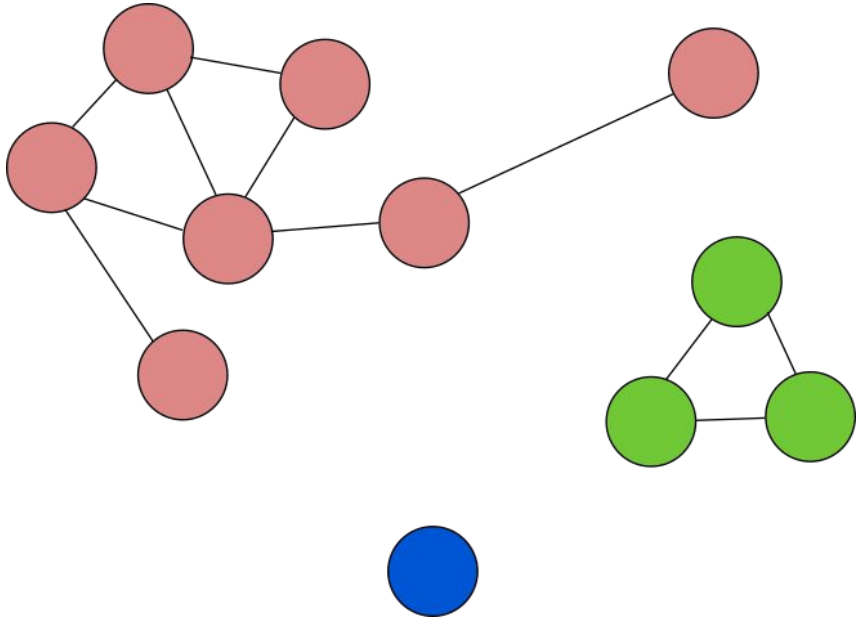
Поиск в глубину в неориентированных графах

Процедура ***dfs*** вызывается от каждой вершины не более одного раза, а внутри процедуры рассматриваются все такие ребра ***{e | begin(e)=u}***. Всего таких ребер для всех вершин в графе ***O(E)***, следовательно, время работы алгоритма оценивается как ***O(V+E)***.

Сириус
IT-Колледж

Выделение компонент связности

Компонента связности - набор вершин графа, между любой парой которых существует путь.



Сириус IT-Колледж

Выделение компонент связности

Для поиска компонент связности используется обычный DFS практически без модификаций. При запуске обхода из одной вершины, он гарантированно посетит все вершины, до которых возможно добраться, то есть, всю компоненту связности, к которой принадлежит начальная вершина. Для нахождения всех компонент просто попытаемся запустить обход из каждой вершины по очереди, если мы ещё не обошли её компоненту ранее.

Простейший вариант: просто заполнить список `comp`, где `comp[i]` - номер компоненты связности, к которой принадлежит вершина `i`.

Сириус IT-Колледж

Выделение компонент связности

```
visited = [False] * n
def dfs(start):
    visited[start] = True
    for v in g[start]:
        if not visited[v]:
            dfs(v)

ncomp = 0
for i in range(n):
    i f not visited[i]:
        ncomp += 1
        dfs(i)
```

Сириус
IT-Колледж

Поиск в глубину в ориентированных графах

Сириус IT-Колледж

Ориентированный ациклический граф

Ориентированный ациклический граф (направленный ациклический граф, DAG, directed acyclic graph) — орграф, в котором отсутствуют направленные циклы, но могут быть «параллельные» пути, выходящие из одного узла и разными путями приходящие в конечный узел. Направленный ациклический граф является обобщением дерева (точнее, их объединения — леса).

Направленные ациклические графы широко используются в приложениях: в компиляторах, в искусственном интеллекте (для представления искусственных нейронных сетей без обратной связи), в статистике и машинном обучении.

Сириус IT-Колледж

Нахождение цикла в орграфе

1. Пометить текущий узел как посещенный и добавить его индекс в стек.
2. Пройти в цикле по вершинам, выполнить рекурсивный вызов функции **dfs** (данный шаг необходим для гарантии, что, если граф является лесом, мы проверим все подграфы):
    a. В каждом рекурсивном вызове найти все смежные вершины для данной вершины:
        i. Если смежная вершина уже добавлена в стек, то граф циклический, возвращаем истину.
        ii. Иначе, вызываем рекурсивную функцию для смежной вершины.
    b. При выходе из рекурсивного вызова, удалить текущий узел из стека, чтобы показать, что он более не является частью проверяемого пути.
3. Если какая либо из функций возвращает истину, остановить дальнейшее выполнение и вернуть истину в качестве результата.

Сириус IT-Колледж

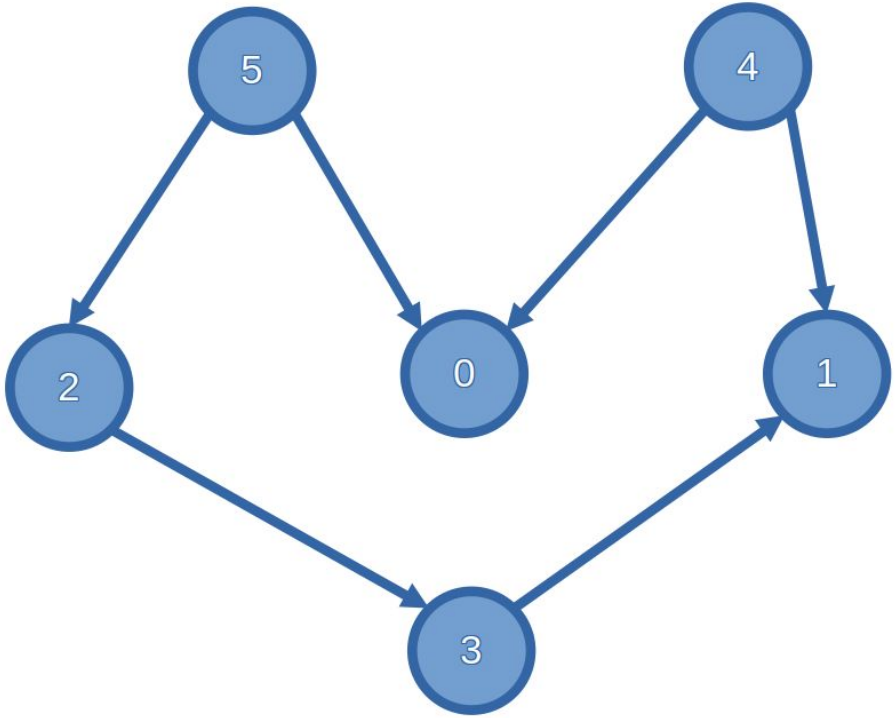
Топологическая сортировка вершин

Топологическая сортировка для ориентированного ациклического графа (Directed Acyclic Graphs, DAG) — это линейное упорядочение вершин, для которого выполняется следующее условие — для каждого направленного ребра ***uv*** вершина ***u*** предшествует вершине ***v*** в упорядочении. Если граф не является DAG, то топологическая сортировка для него невозможна.

Сириус IT-Колледж

Топологическая сортировка вершин

Например, топологическая сортировка приведенного графа — «5 4 2 3 1 0». Для графа может существовать несколько топологических сортировок. Например, другая топологическая сортировка для этого же графа — «4 5 2 3 1 0». Первая вершина в топологической сортировке — это всегда вершина без входящих ребер.



Сириус IT-Колледж

Алгоритм Кана

1. Вычислить количество входящих дуг для каждой вершины в графе и установить начальное значение счетчика посещенных узлов на 0.
2. Выбрать все вершины с 0 входящих дуг и поставить их все в очередь.
3. Добавить одну посещенную вершину к счетчику после удаления вершины из очереди.
4. Для каждой смежной вершины уменьшить число входов на 1.
5. Добавить вершину в очередь, если если число входов любой из смежных вершин уменьшилось до 0.
6. Пока очередь не пуста, повторять с шага 3.
7. Топологическая сортировка невозможна для графа если число посещенных узлов не равно числу вершин графа.

Сириус IT-Колледж

Алгоритм Кана

```
def isCyclic(self):
    inDegree = [0] * self.V
    q = deque()
    visited = 0

    for u in range(self.V):
        for v in self.adj[u]:
            inDegree[v] += 1

        for u in range(self.V):
            if inDegree[u] == 0:
                q.append(u)

        while q:
            u = q.popleft()
            visited += 1

            for v in self.adj[u]:
                inDegree[v] -= 1
                if inDegree[v] == 0:
                    q.append(v)

    return visited != self.V
```