

Лекция 15. Графы: обход в глубину (DFS), поиск циклов, топологическая сортировка.

1. Введение в графы

Граф – это структура данных, состоящая из **вершин** (узлов) и **ребер** (связей). Графы бывают:

- **Ориентированные** (ребра имеют направление, стрелки).
- **Неориентированные** (ребра без направления).

Примеры применения графов:

- Социальные сети (друзья – неориентированный граф).
- Дорожная сеть (дороги – ребра, перекрёстки – вершины).
- Зависимости между задачами (ориентированный граф).

Основные характеристики графа:

- n – количество вершин.
- m – количество ребер.

Способы представления графа в памяти:

1. **Матрица смежности** – квадратная матрица $n \times n$, где $\text{matrix}[u][v] = 1$, если есть ребро из u в v .

Память: $O(n^2)$, быстрая проверка наличия ребра.

2. **Список смежности** – для каждой вершины хранится список смежных вершин.

Память: $O(n + m)$, эффективен для обходов.

Пример реализации списка смежности:

```
from collections import defaultdict
from typing import List

class Graph:
    def __init__(self, n: int):
        self.n = n # Количество вершин в графе
        # Словарь для хранения списка смежности
        # Ключ - номер вершины, значение - список смежных вершин
        self.graph = defaultdict(list)

    def add_edge(self, u: int, v: int, directed: bool = False):
        # Добавляем ребро из вершины u в вершину v
        self.graph[u].append(v)
        # Если граф неориентированный, добавляем обратное ребро
        if not directed:
            self.graph[v].append(u)
```

2. Обход в глубину (DFS)

DFS – алгоритм обхода графа, который идёт «вглубь», пока это возможно, прежде чем возвращаться.

Принцип работы:

1. Пометить текущую вершину как посещённую.
2. Рекурсивно посетить всех непосещённых соседей.

Пример DFS для неориентированного графа (компоненты связности):

```
def dfs_components(graph: Graph) -> List[int]:  
    # Массив для отслеживания посещенных вершин  
    visited = [False] * graph.n  
    # Массив для хранения идентификатора компоненты связности для каждой  
    # вершины  
    components = [-1] * graph.n  
    comp_id = 0 # Текущий идентификатор компоненты  
  
    def dfs(u: int):  
        # Помечаем текущую вершину как посещенную  
        visited[u] = True  
        # Присваиваем вершине идентификатор текущей компоненты  
        components[u] = comp_id  
        # Рекурсивно обходим всех непосещенных соседей  
        for v in graph.graph[u]:  
            if not visited[v]:  
                dfs(v)  
  
    # Обходим все вершины графа  
    for u in range(graph.n):  
        # Если вершина не посещена, начинаем новую компоненту связности  
        if not visited[u]:  
            dfs(u)  
            comp_id += 1 # Переходим к следующей компоненте  
  
    return components
```

Сложность: $O(n + m)$.

3. Поиск циклов в графах

3.1. В неориентированном графе

Цикл существует, если при DFS мы находим ребро в уже посещённую вершину, которая не является непосредственным родителем.

```
def has_cycle_undirected(graph: Graph) -> bool:  
    visited = [False] * graph.n  
  
    def dfs(u: int, parent: int) -> bool:  
        # Помечаем текущую вершину как посещенную  
        visited[u] = True
```

```

# Обходим всех соседей
for v in graph.graph[u]:
    if not visited[v]:
        # Если сосед не посещен, рекурсивно проверяем его
        if dfs(v, u):
            return True
    # Если сосед посещен и не является родителем - нашли цикл
    elif v != parent:
        return True
return False

# Проверяем все вершины (граф может быть несвязным)
for u in range(graph.n):
    if not visited[u]:
        if dfs(u, -1): # -1 означает отсутствие родителя у стартовой
вершины
            return True
return False

```

3.2. В ориентированном графе

Используем три состояния посещения:

- 0 – не посещена.
- 1 – в процессе обработки (в стеке рекурсии).
- 2 – полностью обработана.

```

def has_cycle_directed(graph: Graph) -> bool:
    # Три состояния: 0 - не посещена, 1 - в обработке, 2 - обработана
    state = [0] * graph.n

    def dfs(u: int) -> bool:
        state[u] = 1 # Начинаем обработку вершины
        for v in graph.graph[u]:
            if state[v] == 0:
                # Если вершина не посещена, рекурсивно проверяем
                if dfs(v):
                    return True
            elif state[v] == 1:
                # Нашли ребро к вершине, которая в процессе обработки - это
цикл
                return True
        state[u] = 2 # Завершили обработку вершины
    return False

    # Проверяем все вершины
    for u in range(graph.n):
        if state[u] == 0:
            if dfs(u):
                return True
    return False

```

4. Топологическая сортировка

Топологическая сортировка – упорядочивание вершин ориентированного ациклического графа (DAG), при котором все рёбра идут слева направо.

Алгоритм Кана (использует степени входа):

1. Найти все вершины с нулевой входящей степенью.
2. Добавить их в очередь и последовательно удалять, обновляя степени соседей.

```
from collections import deque

def topological_sort_kahn(graph: Graph) -> List[int]:
    # Массив для хранения входящих степеней вершин
    in_degree = [0] * graph.n
    # Вычисляем входящие степени
    for u in graph.graph:
        for v in graph.graph[u]:
            in_degree[v] += 1

    # Очередь для вершин с нулевой входящей степенью
    queue = deque([u for u in range(graph.n) if in_degree[u] == 0])
    topo_order = [] # Результирующий топологический порядок

    while queue:
        u = queue.popleft()
        topo_order.append(u)
        # Уменьшаем входящие степени всех соседей
        for v in graph.graph[u]:
            in_degree[v] -= 1
            # Если входящая степень стала нулевой, добавляем в очередь
            if in_degree[v] == 0:
                queue.append(v)

    # Проверяем, что все вершины были обработаны
    if len(topo_order) != graph.n:
        raise ValueError("Граф содержит циклы!")
    return topo_order
```

Алгоритм на основе DFS:

- Добавляем вершину в результат после обработки всех её соседей.

```
def topological_sort_dfs(graph: Graph) -> List[int]:
    visited = [False] * graph.n
    stack = [] # Стек для хранения вершин в порядке завершения обработки

    def dfs(u: int):
        visited[u] = True
```

```

# Рекурсивно обрабатываем всех непосещенных соседей
for v in graph.graph[u]:
    if not visited[v]:
        dfs(v)
# После обработки всех соседей добавляем вершину в стек
stack.append(u)

# Запускаем DFS для всех непосещенных вершин
for u in range(graph.n):
    if not visited[u]:
        dfs(u)

# Топологический порядок - обратный порядку добавления в стек
return stack[::-1]

```

5. Полная реализация с примерами

Класс графа с методами DFS, поиска циклов и топологической сортировки:

```

from collections import defaultdict, deque
from typing import List, Tuple


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

    def add_edge(self, u: int, v: int, directed: bool = False):
        self.graph[u].append(v)
        if not directed:
            self.graph[v].append(u)

    def dfs(self, start: int) -> List[int]:
        visited = [False] * self.n
        order = [] # Порядок обхода вершин

        def _dfs(u: int):
            visited[u] = True
            order.append(u)
            # Рекурсивно посещаем всех непосещенных соседей
            for v in self.graph[u]:
                if not visited[v]:
                    _dfs(v)

        _dfs(start)
        return order

    def has_cycle_directed(self) -> bool:
        state = [0] * self.n

```

```

def dfs(u: int) -> bool:
    state[u] = 1 # Вершина в обработке
    for v in self.graph[u]:
        if state[v] == 0:
            if dfs(v):
                return True
        elif state[v] == 1:
            return True # Найден цикл
    state[u] = 2 # Обработка завершена
    return False

for u in range(self.n):
    if state[u] == 0:
        if dfs(u):
            return True
    return False

def topological_sort(self) -> List[int]:
    # Сначала проверяем на наличие циклов
    if self.has_cycle_directed():
        raise ValueError("Граф содержит циклы!")

    visited = [False] * self.n
    stack = []

    def dfs(u: int):
        visited[u] = True
        for v in self.graph[u]:
            if not visited[v]:
                dfs(v)
        stack.append(u)

    for u in range(self.n):
        if not visited[u]:
            dfs(u)

    return stack[::-1]

# Создаем ориентированный граф без циклов
g = Graph(6)
edges = [(5, 2), (5, 0), (4, 0), (4, 1), (2, 3), (3, 1)]
for u, v in edges:
    g.add_edge(u, v, directed=True)

print("Топологическая сортировка:", g.topological_sort())
print("Есть цикл?", g.has_cycle_directed())

# Добавляем цикл (1 -> 5)
g.add_edge(1, 5, directed=True)
print("Есть цикл после добавления ребра (1, 5)?", g.has_cycle_directed())

```

6. Сравнение алгоритмов

Алгоритм	Сложность	Применение
DFS (обход)	$O(n + m)$	Компоненты связности, поиск путей
Поиск циклов (неор.)	$O(n + m)$	Проверка на наличие циклов
Поиск циклов (ор.)	$O(n + m)$	Проверка на ацикличность
Топологическая сортировка	$O(n + m)$	Упорядочивание задач, зависимостей