Лекция 17. Мосты. Точки сочленения. Поиск Эйлерова цикла.

1. Введение

Деревья и связность – фундаментальные понятия в теории графов. Однако многие задачи требуют анализа **уязвимых мест** связности:

- **Мост** – ребро, удаление которого увеличивает количество компонент связности графа.
- **Точка сочленения** – вершина, удаление которой (вместе с инцидентными рёбрами) увеличивает количество компонент связности.

Поиск таких элементов критичен для анализа надёжности сетей (компьютерных, транспортных, социальных).

Эйлеров цикл – это цикл в графе, проходящий через каждое ребро **ровно один раз**. Граф, содержащий такой цикл, называется **эйлеровым**. Задача поиска Эйлерова цикла имеет приложения в маршрутизации, проектировании микросхем и решении головоломок.

2. Мосты и точки сочленения

2.1. Основные определения

- **Мост.** Ребро (u, v) является мостом, если его удаление разрывает связность графа, увеличивая количество компонент связности.
- **Точка сочленения.** Вершина v является точкой сочленения, если существует хотя бы две другие вершины u и w, такие что все пути между u и w проходят через v.

2.2. Алгоритм поиска мостов и точек сочленения

Для эффективного поиска используется модификация **обхода в глубину (DFS)**.

Ключевые идеи:

1. Вычисляем **время входа** (tin) в каждую вершину при DFS.
2. Для каждой вершины вычисляем **нижайшее время достижимости** (low). Значение low[u] – это минимальное из:
   - tin[u] (время входа в саму вершину).
   - tin[p] для любого ребра (u, p), ведущего к предку u в дереве DFS (обратное ребро).
   - low[to] для любого ребра (u, to), ведущего к потомку u в дереве DFS (прямое ребро).

Критерии:

- **Ребро (u, v) является мостом**, если low[v] > tin[u]. Это означает, что вершина v и её потомки не имеют обратных рёбер к предкам u, поэтому удаление (u, v) разорвёт связь.

• **Вершина u является точкой сочленения**, если:

1. Для корня DFS: у него **два или более детей** в дереве DFS.

2. Для остальных вершин: существует ребёнок v, такой что low[v] >= tin[u]. Это означает, что v и её потомки не имеют обратных рёбер к строгим предкам u, поэтому удаление u разорвёт связь между v и этими предками.

2.3. Реализация

```
from collections import defaultdict
from typing import List, Tuple, Set


class Graph:
    def __init__(self, n: int):
        self.n = n
        self.graph = defaultdict(list)
        self.time = 0  # Глобальный счётчик времени для tin

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

    def find_bridges_and_articulation_points(self) -> Tuple[List[Tuple[int,
int]], List[int]]:
        """
        Находит все мосты и точки сочленения в графе.
        Возвращает:
            bridges: список кортежей (u, v) - мостов
            articulation_points: список вершин - точек сочленения
        """
        visited = [False] * self.n
        tin = [-1] * self.n  # Время первого входа в вершину
        low = [-1] * self.n  # Минимальное время достижимости вершины
        bridges = []
        articulation_points = []
        parent = [-1] * self.n  # Родитель в дереве DFS

        def dfs(u: int, p: int):
            visited[u] = True
            tin[u] = self.time
            low[u] = self.time
            self.time += 1
            children = 0  # Количество детей в дереве DFS (для корня)
            is_articulation = False

            for v in self.graph[u]:
                if v == p:  # Пропускаем ребро к родителю
                    continue
                if not visited[v]:
                    parent[v] = u
                    children += 1
                    dfs(v, u)
```

```
                    # Обновляем low[u] после возврата из рекурсии
                    low[u] = min(low[u], low[v])

                    # Критерий моста
                    if low[v] > tin[u]:
                        bridges.append((u, v))  # u < v для упорядочивания

                    # Критерий точки сочленения (не для корня)
                    if p != -1 and low[v] >= tin[u]:
                        is_articulation = True

                else:
                    # v уже посещена и не является родителем -> обратное
ребро
                    low[u] = min(low[u], tin[v])

            # Отдельная обработка корня для точки сочленения
            if p == -1 and children > 1:
                is_articulation = True

            if is_articulation:
                articulation_points.append(u)

        # Запускаем DFS для всех компонент связности
        for i in range(self.n):
            if not visited[i]:
                self.time = 0  # Сброс времени для новой компоненты
                dfs(i, -1)

        return bridges, articulation_points


# Пример использования
g = Graph(5)
edges = [(1, 0), (0, 2), (2, 1), (0, 3), (3, 4)]
for u, v in edges:
    g.add_edge(u, v)

bridges, articulation_points = g.find_bridges_and_articulation_points()
print("Мосты:", bridges)  # Ожидаем: [(3, 4), (0, 3)]
print("Точки сочленения:", articulation_points)  # Ожидаем: [3, 0]
```

3. Эйлеров цикл

3.1. Условия существования

• **Неориентированный граф** является эйлеровым **тогда и только тогда, когда**:

1. Граф связный (исключая изолированные вершины).
2. Степень каждой вершины **чётна**.

• **Ориентированный граф** является эйлеровым **тогда и только тогда, когда**:

1. Граф слабо связный (связный как неориентированный).
2. Для каждой вершины **входящая степень равна исходящей** (in_degree[v] == out_degree[v]).

3.2. Алгоритм Флёри (Fleury)

Исторический алгоритм, который избегает прохождения по мостам, если есть альтернатива. Медленный на практике.

3.3. Алгоритм на основе DFS (иерархический DFS)

Более эффективный алгоритм, использующий стек и DFS.
Шаги алгоритма:
1. Проверить условия существования цикла.
2. Начать с вершины, имеющей исходящие рёбра (для неориентированного – любой вершины с ненулевой степенью).
3. Рекурсивно обходить граф, удаляя пройденные рёбра.
4. Добавлять вершину в результат, когда у неё не останется исходящих рёбер.
5. Развернуть результат, чтобы получить цикл в правильном порядке.

3.4. Реализация

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


class EulerianGraph:
    def __init__(self, n: int, directed: bool = False):
        self.n = n
        self.directed = directed
        self.graph = defaultdict(list)
        self.in_degree = [0] * n
        self.out_degree = [0] * n

    def add_edge(self, u: int, v: int):
        self.graph[u].append(v)
        self.out_degree[u] += 1
        self.in_degree[v] += 1
        if not self.directed:
            self.graph[v].append(u)
            self.out_degree[v] += 1
            self.in_degree[u] += 1

    def has_eulerian_circuit(self) -> bool:
        """Проверяет, существует ли Эйлеров цикл."""
        if self.directed:
            # Ориентированный: слабая связность и in_degree == out_degree
```

```
            for i in range(self.n):
                if self.in_degree[i] != self.out_degree[i]:
                    return False
            # Проверка слабой связности опущена для краткости, но важна
        else:
            # Неориентированный: связность и все степени чётны
            for i in range(self.n):
                if self.out_degree[i] % 2 != 0:
                    return False
            # Проверка связности опущена для краткости, но важна
        return True

    def find_eulerian_circuit(self) -> List[int]:
        """Находит Эйлеров цикл. Граф должен быть эйлеровым."""
        if not self.has_eulerian_circuit():
            raise ValueError("Граф не является эйлеровым")

        # Создаём локальную копию списка смежности, чтобы не портить исходную
        graph_copy = defaultdict(list)
        for u in self.graph:
            graph_copy[u] = self.graph[u][:]  # Копируем список

        stack = []  # Стек для вершин
        circuit = []  # Результирующий цикл

        # Начинаем с вершины, у которой есть рёбра
        start_vertex = 0
        for i in range(self.n):
            if self.out_degree[i] > 0:
                start_vertex = i
                break

        stack.append(start_vertex)

        while stack:
            u = stack[-1]
            if graph_copy[u]:  # Если остались исходящие рёбра
                # Берём следующее ребро
                v = graph_copy[u].pop()
                # Удаляем обратное ребро для неориентированного графа
                if not self.directed:
                    # Находим и удаляем ребро (v, u) из списка v
                    graph_copy[v].remove(u)
                stack.append(v)
            else:
                # Все рёбра из u пройдены, добавляем u в цикл
                circuit.append(stack.pop())

        # Цикл получается в обратном порядке
        circuit.reverse()
        return circuit


# Пример использования для неориентированного графа
g_undir = EulerianGraph(3, directed=False)
```

```
g_undir.add_edge(0, 1)
g_undir.add_edge(1, 2)
g_undir.add_edge(2, 0)
print("Эйлеров цикл (неориентированный):", g_undir.find_eulerian_circuit())
# Вывод: [0, 2, 1, 0]

# Пример использования для ориентированного графа
g_dir = EulerianGraph(3, directed=True)
g_dir.add_edge(0, 1)
g_dir.add_edge(1, 2)
g_dir.add_edge(2, 0)
print("Эйлеров цикл (ориентированный):", g_dir.find_eulerian_circuit())
# Вывод: [0, 1, 2, 0]
```

4. Сравнение алгоритмов

Алгоритм	**Сложность**	**Применение**
Поиск мостов и точек сочленения	O(n + m)	Анализ уязвимостей сетей, надёжности
Алгоритм Флёри	$O(m^2)$	Учебный, исторический, не рекомендуется на практике
Иерархический DFS (Эйлер)	O(m)	Эффективный поиск Эйлерова цикла, маршрутизация