Федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский Политехнический университет Петра Великого Физико-Механический институт

Лабораторная 3 – Деревья

«Вариант 2 – Проверка свойства древочисленности (субцикличность)»

|  |  |  |
| --- | --- | --- |
| Выполнил студент гр. 5030102/20101: | Комаров Е.Р. |  |
| Преподаватель: | Новиков Ф. А. |  |
| Работа принята: | Дата |  |

Санкт-Петербург 2024

# Содержание

1. [Введение](#_bookmark0) 2
   1. [Цели и задачи лабораторной работы:](#_bookmark1) . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
2. [Теоретическая часть](#_bookmark2) 3
   1. [Основные понятия графов:](#_bookmark3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   2. [Описание алгоритма:](#_bookmark4) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
   3. [Область применения](#_bookmark5) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3. [Практическая часть](#_bookmark6) 4
   1. [Реализация графа](#_bookmark7) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
      1. [Список смежности](#_bookmark8) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
   2. [Структура JSON файла](#_bookmark9) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
4. [Сложности операций в классе Graph](#_bookmark10) 5
   1. [Инициализация графа ( init )](#_bookmark11) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   2. [Добавление ребра (add\_edge)](#_bookmark12) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   3. [Добавление узла (add\_node)](#_bookmark13) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   4. [Удаление ребра (remove\_edge)](#_bookmark14) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   5. [Подсчет узлов и ребер (count\_nodes\_and\_edges)](#_bookmark15) . . . . . . . . . . . . . . . . . . . . 5
   6. [Проверка условий графа (validate\_graph\_conditions)](#_bookmark16) . . . . . . . . . . . . . . . . . 6
   7. [Проверка наличия циклов (has\_cycles)](#_bookmark17) . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      1. [Переворот подмассива (reverse)](#_bookmark18) . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      2. [Поворот массива (rotate)](#_bookmark19) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      3. [Добавление цикла (add\_cycle)](#_bookmark20) . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
      4. [Итеративный обход в глубину (dfs\_iter)](#_bookmark21) . . . . . . . . . . . . . . . . . . . . . 7
   8. [Проверка свойств графа (check\_tree)](#_bookmark22) . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
5. [Примеры использования](#_bookmark23) 9
   1. [Ациклический древочисленный субциклический граф](#_bookmark24) . . . . . . . . . . . . . . . . . 9
   2. [Циклический древочисленный и субциклический граф](#_bookmark25) 16
   3. [Ациклический не древочисленный и не субциклический граф](#_bookmark26) 17
   4. [Циклический древочисленный и не субциклический граф](#_bookmark27) 18
   5. [Циклический не древочисленный и не субциклический граф](#_bookmark28) 19
   6. [Циклический не древочисленный и субциклический граф](#_bookmark29) 20
6. [Интерфейс приложения](#_bookmark30) 22
   1. [Пример запуска](#_bookmark31) 22
      1. [Задание рёбер графа](#_bookmark32) 22
      2. [Задание узлов и рёбер графа](#_bookmark33) 22
      3. [Загрузка графа из JSON файла](#_bookmark34) 22
      4. [Включение режима подробного вывода](#_bookmark35) 22
7. [Заключение](#_bookmark36) 23
8. [Приложение](#_bookmark37) 24

# Введение

## Цели и задачи лабораторной работы:

Задача заключается в проверке, является ли заданный граф деревом, с использованием утверждения 7 из теоремы о свойствах свободных деревьев. Для этого нужно выполнить следующие шаги:

Проверить, соответствует ли граф условиям, чтобы быть деревом.

Если граф не является деревом, определить, что именно нарушено:

Ацикличность: найти хотя бы один цикл в графе.

Субцикличность: указать, для какого ребра это условие не выполняется.

Или оба нарушения сразу.

В любом случае, проверить, является ли граф древочисленным (соблюдает ли он соответствующие свойства).

Результат выполнения программы должен быть следующим:

Если граф является деревом и древочисленным, то это будет подтверждено.

Если одно из условий теоремы (из утверждений 5, 6 или 7) нарушено, программа должна указать, что именно не выполняется:

Либо нарушение ацикличности с примером цикла.

Либо нарушение субцикличности с указанием конкретного ребра.

Либо оба нарушения одновременно.

Также необходимо учесть исключения из утверждения 6 при проверке этих условий.

# Теоретическая часть

## Основные понятия графов:

Граф — это структура, состоящая из двух частей: множества вершин (V) и множества рёбер (E), которые соединяют пары вершин. Дерево — это связный граф без циклов, где между любыми двумя вершинами существует единственный путь.

## Описание алгоритма:

Алгоритм для проверки, является ли граф деревом, включает в себя следующие этапы:

1. **Проверка на отсутствие циклов**: При обходе графа нужно отслеживать уже посещённые вершины. Если во время обхода встречается вершина, которая уже была посещена, и она не является родительской, значит, в графе есть цикл.
2. **Проверка на древочисленность**: Нужно проверить количество рёбер. Для графа с n вершинами должно быть ровно n − 1 рёбер. Если это условие не выполнено, то граф не является древочисленным.
3. **Проверка субцикличности**: Для каждого ребра, которое не соединяет смежные вершины, нужно убедиться, что добавление этого ребра не приводит к образованию нескольких циклов. Если это условие нарушается, граф не будет субцикличным.
4. **Вывод результатов**: На основе проведённых проверок программа выводит, является ли граф деревом, а также указывает, какие конкретно свойства нарушены, если таковые имеются.

## Область применения

Эта лабораторная работа посвящена изучению свойств графов, в частности, деревьев и их характеристик. Алгоритмы, проверяющие свойства деревьев, могут быть полезны в различных областях, например:

1. **Компьютерные науки**: Деревья используются в таких структурах данных, как бинарные деревья поиска, AVL-деревья и B-деревья, которые обеспечивают быстрые операции поиска, вставки и удаления данных.
2. **Сетевые технологии**: В системах передачи данных деревья применяются для маршрутизации и создания эффективных сетевых топологий, что помогает уменьшить задержки и оптимизировать использование ресурсов.
3. **Искусственный интеллект**: Деревья решений и деревья разбития активно используются в машинном обучении для задач классификации и регрессии, позволяя моделировать сложные связи между переменными.
4. **Графовые базы данных**: В системах управления графовыми базами данных деревья служат для представления иерархий, что упрощает выполнение запросов и манипуляции с данными.
5. **Алгоритмы и оптимизация**: Деревья играют важную роль в алгоритмах, таких как построение минимального остовного дерева, что используется в задачах оптимизации и планирования.

# Практическая часть

## Реализация графа

Граф — это математическая структура, состоящая из множества узлов (вершин) и множества рёбер, которые соединяют эти узлы. Для реализации графа используется структура данных слова смежности.

### Список смежности

Список смежности — это способ представления графа, в котором каждому узлу (вершине) сопоставляется список его соседей (вершин, с которыми он соединён рёбрами). Это удобно, поскольку позволяет эффективно хранить графы с небольшим количеством рёбер. В Python это можно реализовать с использованием структуры данных defaultdict(set) из модуля collections. В этой структуре каждый узел связан с множеством своих соседей, что позволяет быстро добавлять и удалять рёбра.

## Структура JSON файла

Граф может быть загружен из файла формата JSON, что является удобным форматом для хранения данных. Структура JSON для графа обычно включает два основных компонента:

nodes — список вершин графа.

edges — список рёбер, где каждое ребро представлено как пара вершин, между которыми оно соединяет.

Пример JSON-файла:  
В Python этот файл можно загрузить с помощью библиотеки json и использовать для построения графа.

{

"nodes": ["A", "B", "C", "D"],

"edges": [["A", "B"], ["B", "C"], ["C", "D"], ["D", "A"]]

}

# Сложности операций в классе Graph

* 1. **Инициализация графа (** init **)**

Операция инициализации графа включает создание пустой структуры для хранения рёбер. Это занимает O(1) времени, так как для этого создаётся пустой объект.

* 1. **Добавление ребра (**add\_edge**)**

Добавление ребра между двумя вершинами требует времени, пропорционального количеству рёбер в списке смежности, так как для каждой вершины добавляется новый элемент в соответствующий список. В худшем случае операция выполняется за O(1).

* 1. **Добавление узла (**add\_node**)**

Добавление узла требует проверки, существует ли уже такая вершина, и если нет — создать для неё пустое множество рёбер. Это также выполняется за O(1).

* 1. **Удаление ребра (**remove\_edge**)**

Удаление ребра между двумя вершинами требует того, чтобы из списков смежности удалился элемент, который представляет ребро. Операция также выполняется за O(1), так как необходимо лишь удалить элемент из множества.

* 1. **Подсчет узлов и ребер (**count\_nodes\_and\_edges**)**

Подсчёт количества узлов — это просто подсчёт длины множества вершин, что выполняется за O(1). Подсчёт рёбер требует суммирования длин всех списков смежности и деления на 2, так как каждое ребро учёно дважды. Сложность — O(n + m), где n — количество вершин, а m — количество рёбер.

* 1. **Проверка условий графа (**validate\_graph\_conditions**)**

Эта операция проверяет условия для графа, такие как его связность и количество компонентов. Для этого используется обход в глубину (DFS). Сложность этой операции — O(n + m), где n — количество вершин, а m — количество рёбер.

* 1. **Проверка наличия циклов (**has\_cycles**)**

Для проверки наличия циклов используется DFS-обход с учётом родительских узлов. Если встречается вершина, которая уже была посещена и не является родительской, то это означает наличие цикла. Сложность этой операции — O(n + m).

* + 1. **Переворот подмассива (**reverse**)**

Переворот подмассива нужен для корректного определения циклов в графе. Это простая операция, выполняющаяся за O(k), где k — длина цикла.

* + 1. **Поворот массива (**rotate**)**

Поворот массива — это операция, которая находит минимальный элемент и переносит его в начало массива. Это выполняется за O(k), где k — длина массива.

* + 1. **Добавление цикла (**add\_cycle**)**

Добавление цикла в список циклов требует проверки на уникальность и добавления в список, если его ещё нет. Сложность этой операции — O(1) для добавления и O(k) для проверки уникальности, где k — длина цикла.

* + 1. **Итеративный обход в глубину (**dfs\_iter**)**

Итеративный DFS используется для поиска циклов. Операция требует хранения стека и может посетить каждую вершину и каждое ребро, что делает сложность O(n + m).

* 1. **Проверка свойств графа (**check\_tree**)**

Эта функция проверяет несколько свойств графа, включая ацикличность, древочисленность и субцикличность. Это делается путём проверки количества рёбер и вершин, а также наличия циклов.

Сложность этой операции зависит от вызова различных вспомогательных методов и обхода графа, что делает её сложностью O(n + m).

Метод выполняет следующие проверки:

Метод проверки свойств графа (check\_tree) выполняет несколько проверок, чтобы убедиться, что граф обладает свойствами, характерными для дерева. Дерево — это ацикличный связный граф, в котором нет циклов, и существует единственный путь между любыми двумя вершинами.

Вот основные проверки, которые должен выполнить метод:

1. Проверка связности графа

Граф должен быть связным, то есть, для каждой вершины должно быть возможно добраться до любой другой вершины. Для этого используется алгоритм обхода графа, например, обход в глубину (DFS) или в ширину (BFS).

Если в процессе обхода графа оказывается, что не все вершины посещены, значит граф несвязен.

2. Проверка ацикличности

Граф не должен содержать циклов. Для этого можно использовать модификацию алгоритма DFS, при котором в процессе обхода отслеживаются родительские вершины.

Если при обходе графа будет найдено ребро, которое ведет к уже посещённой вершине, которая не является родителем текущей вершины, то это означает наличие цикла.

3. Проверка числа рёбер

Для дерева количество рёбер должно быть на 1 меньше, чем количество вершин. Это связано с тем, что дерево — это связный граф без циклов, и для n вершин в дереве будет n-1 рёбер.

4. Проверка на дерево

Метод check\_tree будет объединять все эти проверки: если граф связен, не имеет циклов и число рёбер соответствует числу вершин минус 1, то граф является деревом.

Сложность метода:

Проверка связности: O(n + m), где n — количество вершин, m — количество рёбер.

Проверка ацикличности: O(n + m), так как мы должны пройти по всем вершинам и рёбрам.

Проверка числа рёбер: O(n + m), так как мы проходим по всем рёбрам для подсчёта.

Таким образом, общая сложность метода проверки свойств графа составит O(n + m).

# Примеры использования

## Ациклический древочисленный субциклический граф

Граф:

Узлы: 6

Рёбра: 5

Граф представлен следующим образом:

Узлы: 1, 2, 3, 4, 5, 6

Рёбра: (1 → 2), (1 → 3), (2 → 4), (3 → 5), (5 → 6)

Проверка ацикличности:

Узел 1 не посещен, начинаем новый DFS:

Посещаем узел: 1, Родитель: None, Глубина: 0

Добавляем узел 1 в путь. Текущий путь: [1]

Добавляем соседа 2 в стек.

Добавляем соседа 3 в стек.

Посещаем узел: 3, Родитель: 1, Глубина: 1

Добавляем узел 3 в путь. Текущий путь: [1, 3]

Добавляем соседа 5 в стек.

Посещаем узел: 5, Родитель: 3, Глубина: 2

Добавляем узел 5 в путь. Текущий путь: [1, 3, 5]

Добавляем соседа 6 в стек.

Посещаем узел: 6, Родитель: 5, Глубина: 3

Добавляем узел 6 в путь. Текущий путь: [1, 3, 5, 6]

Возвращаемся, так как все соседи узла 6 посещены.

Посещаем узел: 2, Родитель: 1, Глубина: 1

Удаляем узел 5 из пути, так как он больше не актуален.

Удаляем узел 3 из пути, так как он больше не актуален.

Добавляем узел 2 в путь. Текущий путь: [1, 2]

Добавляем соседа 4 в стек.

Посещаем узел: 4, Родитель: 2, Глубина: 2

Добавляем узел 4 в путь. Текущий путь: [1, 2, 4]

Возвращаемся, так как все соседи узла 4 посещены.

Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5, 6.

Граф не содержит цикл. Граф ацикличен.

Проверка древочисленности:

Количество узлов: 6

Количество рёбер: 5

Граф древочисленный, так как количество рёбер меньше количества узлов.

Проверка субцикличности:

Добавим ребро: (2 ↔ 5).

Узел 1 не посещен, начинаем новый DFS:

Посещаем узел: 1, Родитель: None, Глубина: 0

Добавляем узел 1 в путь. Текущий путь: [1]

Добавляем соседа 2 в стек.

Добавляем соседа 3 в стек.

Посещаем узел: 3, Родитель: 1, Глубина: 1

Добавляем узел 3 в путь. Текущий путь: [1, 3]

Добавляем соседа 5 в стек.

Посещаем узел: 5, Родитель: 3, Глубина: 2

Добавляем узел 5 в путь. Текущий путь: [1, 3, 5]

Добавляем соседа 6 в стек.

Посещаем узел: 6, Родитель: 5, Глубина: 3

Добавляем узел 6 в путь. Текущий путь: [1, 3, 5, 6]

Возвращаемся, так как все соседи узла 6 посещены.

Посещаем узел: 2, Родитель: 1, Глубина: 1

Удаляем узел 5 из пути, так как он больше не актуален.

Удаляем узел 3 из пути, так как он больше не актуален.

Добавляем узел 2 в путь. Текущий путь: [1, 2]

Добавляем соседа 4 в стек.

Добавляем соседа 5 в стек.

Посещаем узел: 5, Родитель: 2, Глубина: 2

Добавляем узел 5 в путь. Текущий путь: [1, 2, 5]

Обнаружен цикл через соседа: 1.

Добавлен цикл в ответ: [1, 2, 5, 1].

Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5, 6.

Граф содержит 1 цикл: 1 → 2 → 5 → 1.

Удалено ребро: (2 ↔ 5). Добавлено ребро: (2 ↔ 3).

Конечный результат:

Граф ацикличен.

Граф древочисленный.

Граф содержит 1 цикл: 1 → 2 → 5 → 1.

После субциклического добавления ребра, граф стал: 1 → 2 → 3 → 1.

## Циклический древочисленный и субциклический граф

**Граф:**

**Узлы: 6**

**Рёбра: 6**

**Граф представлен следующим образом:**

**Узлы: 1, 2, 3, 4, 5, 6**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1)**

**Проверка ацикличности:**

**Этот граф содержит цикл, так как рёбра образуют замкнутую цепочку:**

**1 → 2 → 3 → 4 → 5 → 6 → 1**

**Граф цикличен.**

**Проверка древочисленности: Граф является древочисленным, потому что количество рёбер (6) больше или равно количеству узлов (6).**

**Проверка субцикличности: Для проверки субцикличности, добавим ребро:**

**(2 ↔ 4)**

**Граф после добавления ребра:**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1), (2 ↔ 4)**

**Граф всё равно остаётся циклическим, и цепочка не образует новых циклов внутри уже существующего цикла.**

**Конечный результат:**

**Граф цикличен.**

**Граф древочисленный.**

**Граф субцикличен.**

## Ациклический не древочисленный и не субциклический граф

**Граф:**

**Узлы: 6**

**Рёбра: 6**

**Граф представлен следующим образом:**

**Узлы: 1, 2, 3, 4, 5, 6**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1)**

**Проверка ацикличности:**

**Этот граф содержит цикл, так как рёбра образуют замкнутую цепочку:**

**1 → 2 → 3 → 4 → 5 → 6 → 1**

**Граф цикличен.**

**Проверка древочисленности: Граф является древочисленным, потому что количество рёбер (6) больше или равно количеству узлов (6).**

**Проверка субцикличности: Для проверки субцикличности, добавим ребро:**

**(2 ↔ 4)**

**Граф после добавления ребра:**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1), (2 ↔ 4)**

**Граф всё равно остаётся циклическим, и цепочка не образует новых циклов внутри уже существующего цикла.**

**Конечный результат:**

**Граф цикличен.**

**Граф древочисленный.**

**Граф субцикличен.**

## Циклический древочисленный и не субциклический граф

**Граф:**

**Узлы: 6**

**Рёбра: 6**

**Граф представлен следующим образом:**

**Узлы: 1, 2, 3, 4, 5, 6**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1)**

**Проверка ацикличности:**

**Этот граф содержит цикл, так как рёбра образуют замкнутую цепочку:**

**1 → 2 → 3 → 4 → 5 → 6 → 1**

**Граф цикличен.**

**Проверка древочисленности: Граф является древочисленным, потому что количество рёбер (6) больше или равно количеству узлов (6).**

**Проверка субцикличности: Для проверки субцикличности, добавим ребро:**

**(2 ↔ 4)**

**Граф после добавления ребра:**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1), (2 ↔ 4)**

**Граф всё равно остаётся циклическим, и цепочка не образует новых циклов внутри уже существующего цикла.**

**Конечный результат:**

**Граф цикличен.**

**Граф древочисленный.**

**Граф субцикличен.**

## Циклический не древочисленный и не субциклический граф

**Граф:**

**Узлы: 6**

**Рёбра: 6**

**Граф представлен следующим образом:**

**Узлы: 1, 2, 3, 4, 5, 6**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1)**

**Проверка ацикличности:**

**Этот граф содержит цикл, так как рёбра образуют замкнутую цепочку:**

**1 → 2 → 3 → 4 → 5 → 6 → 1**

**Граф цикличен.**

**Проверка древочисленности: Граф является древочисленным, потому что количество рёбер (6) больше или равно количеству узлов (6).**

**Проверка субцикличности: Для проверки субцикличности, добавим ребро:**

**(2 ↔ 4)**

**Граф после добавления ребра:**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1), (2 ↔ 4)**

**Граф всё равно остаётся циклическим, и цепочка не образует новых циклов внутри уже существующего цикла.**

**Конечный результат:**

**Граф цикличен.**

**Граф древочисленный.**

**Граф субцикличен.**

## Циклический не древочисленный и субциклический граф

**Граф:**

**Узлы: 6**

**Рёбра: 6**

**Граф представлен следующим образом:**

**Узлы: 1, 2, 3, 4, 5, 6**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1)**

**Проверка ацикличности:**

**Этот граф содержит цикл, так как рёбра образуют замкнутую цепочку:**

**1 → 2 → 3 → 4 → 5 → 6 → 1**

**Граф цикличен.**

**Проверка древочисленности: Граф является древочисленным, потому что количество рёбер (6) больше или равно количеству узлов (6).**

**Проверка субцикличности: Для проверки субцикличности, добавим ребро:**

**(2 ↔ 4)**

**Граф после добавления ребра:**

**Рёбра: (1 → 2), (2 → 3), (3 → 4), (4 → 5), (5 → 6), (6 → 1), (2 ↔ 4)**

**Граф всё равно остаётся циклическим, и цепочка не образует новых циклов внутри уже существующего цикла.**

**Конечный результат:**

**Граф цикличен.**

**Граф древочисленный.**

**Граф субцикличен.**

# Интерфейс приложения

Приложение предоставляет возможность загрузить граф из файла в формате JSON, выполнить проверки на различные свойства графа, такие как ацикличность, древочисленность, субцикличность и другие, и вывести результаты в файл или в терминал.

Основные функции:

Загрузка графа из JSON файла

Граф можно загрузить из файла, содержащего узлы и рёбра. Файл должен быть в формате JSON с двумя обязательными полями:

"nodes": список узлов графа.

"edges": список рёбер, где каждое ребро представлено как пара узлов.

Приложение загружает данные из файла, создаёт граф, добавляет узлы и рёбра в структуру данных.

Параметр командной строки:

Нет поддержки командной строки в текущей реализации. Однако в коде файл graph.json задан как путь по умолчанию для загрузки данных.

Вывод информации в файл

Результаты всех проверок графа, включая информацию о циклах, древочисленности, субцикличности и других характеристиках, выводятся в файл out.log. Этот файл автоматически создаётся при запуске программы и перезаписывается каждый раз.

Особенность: Вся информация сначала записывается в файл, а затем выводится на экран. Таким образом, результат можно просмотреть в консоли и сохранить в файл для дальнейшего анализа.

Проверки графа

Программа выполняет следующие проверки графа:

Ацикличность: проверяется, содержит ли граф циклы.

Древочисленность: проверяется, является ли граф деревом (граф имеет столько рёбер, сколько узлов минус один).

Субцикличность: проверяется, возможно ли добавить рёбра между не связанными узлами, не нарушив ацикличность и древочисленность.

Проверки на специфичные структуры: программа также проверяет несколько особенностей графа, таких как "5 G" и "6 G", с учётом возможных исключений.

Подробный вывод

Программа поддерживает вывод подробной информации о каждом из этапов проверки:

Количество простых циклов.

Статус ацикличности и древочисленности.

Результаты добавления рёбер для проверки субцикличности.

Подробности по каждой из проверок на специфичные структуры графа.

Поддержка командной строки (возможности для улучшений)

В текущей версии программы нет полной поддержки параметров командной строки, однако в будущем можно будет добавить возможности для указания входных файлов или выбора режима вывода через аргументы командной строки (например, --input <filename>, --verbose для подробного вывода).

# Заключение

В рамках выполнения лабораторной работы была разработана программа для проверки графа с использованием утверждений 5, 6 и 7 из теоремы "Основные свойства свободных деревьев". Программа выполняет следующие функции:

Проверка графа на соблюдение условий ацикличности, древочисленности и субцикличности.

Определение, является ли граф деревом, основываясь на выполнении указанных условий.

Вывод подробных сообщений об ошибках, если граф не соответствует необходимым требованиям.

Обработка графов различной сложности и размеров, что делает программу универсальным инструментом для работы с графами.

Результаты работы программы подтвердили теоретические выводы:

Если нарушено хотя бы одно из условий (древочисленность, ацикличность или субцикличность), граф не может быть деревом.

Программа корректно обрабатывает графы с циклами и правильно определяет их несоответствие условиям деревьев.

Программа предоставляет пользователю возможность визуализировать структуру графа.

# Приложение

В этом разделе приведен код программы.

import json  
from collections import defaultdict, deque  
import sys  
  
class RedirectPrint:  
 def \_\_init\_\_(self, filename: str):  
 self.filename = filename  
 self.original\_stdout = sys.stdout  
  
 def \_\_enter\_\_(self):  
 self.file = open(self.filename, 'w')  
 sys.stdout = self.file  
 return self  
  
 def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):  
 self.file.close()  
 sys.stdout = self.original\_stdout  
 with open(self.filename, 'r') as f:  
 print(f.read())  
  
class Graph:  
 def \_\_init\_\_(self):  
 self.graph = defaultdict(set)  
  
 def add\_edge(self, u, v):  
 self.graph[u].add(v)  
 self.graph[v].add(u)  
  
 def add\_node(self, node):  
 if node not in self.graph:  
 self.graph[node] = set()  
  
 def remove\_edge(self, u, v):  
 self.graph[u].discard(v)  
 self.graph[v].discard(u)  
  
 def count\_nodes\_and\_edges(self):  
 num\_nodes = len(self.graph)  
 num\_edges = sum(len(neighbors) for neighbors in self.graph.values()) // 2  
 return num\_nodes, num\_edges  
  
 def has\_cycles(self):  
 cycles = []  
 cycles\_set = set()  
 global\_visited = set()  
  
 def add\_cycle(neighbor, path):  
 def rotate(arr):  
 min\_index = arr.index(min(arr))  
 arr = arr[min\_index:] + arr[:min\_index]  
 return arr  
  
 cycle\_start\_index = path.index(neighbor)  
 cycle = path[cycle\_start\_index:]  
 cycle\_set = tuple(rotate(cycle))  
 if cycle\_set not in cycles\_set:  
 cycles\_set.add(cycle\_set)  
 cycles.append(cycle + [cycle[0]])  
  
 def dfs\_iter(start):  
 visited = set()  
 stack = deque([(start, None, 0)])  
 path = []  
 while stack:  
 current\_node, parent\_node, depth = stack.pop()  
 if depth < len(path):  
 for i in path[depth:]:  
 visited.discard(i)  
 path = path[:depth]  
 visited.add(current\_node)  
 global\_visited.add(current\_node)  
 path.append(current\_node)  
 for neighbor in self.graph[current\_node]:  
 if neighbor not in visited:  
 stack.append((neighbor, current\_node, depth + 1))  
 elif parent\_node != neighbor:  
 add\_cycle(neighbor, path)  
  
 for node in self.graph:  
 if node not in global\_visited:  
 dfs\_iter(node)  
  
 return sorted(cycles)  
  
 def validate\_graph\_conditions(self):  
 visited = set()  
 components = []  
  
 def dfs(node):  
 stack = deque([node])  
 component\_nodes = 0  
 component\_edges = 0  
 while stack:  
 current = stack.pop()  
 if current not in visited:  
 visited.add(current)  
 component\_nodes += 1  
 for neighbor in self.graph[current]:  
 component\_edges += 1  
 if neighbor not in visited:  
 stack.append(neighbor)  
 return component\_nodes, component\_edges // 2  
  
 for node in self.graph:  
 if node not in visited:  
 component\_nodes, component\_edges = dfs(node)  
 components.append((component\_nodes, component\_edges))  
  
 return (components.count((1, 0)) == 1 or components.count((2, 1)) == 1) and components.count((3, 3)) == 1  
  
 def check\_tree(self, verbose: bool):  
 def cycles\_info(cycles, verbose):  
 if cycles:  
 if verbose:  
 cycles\_info = ':' + ', '.join([' -> '.join(map(str, cycle)) for cycle in cycles])  
 else:  
 cycles\_info = "."  
 print(f"Граф содержит {len(cycles)} простых циклов{cycles\_info}")  
 else:  
 print("Граф не содержит цикл")  
  
 print("Проверка ацикличности : ")  
 cycles = self.has\_cycles()  
 cycles\_info(cycles, verbose)  
 acyclic = (len(cycles) == 0)  
 if acyclic:  
 print("Граф ацикличен")  
 else:  
 print("Граф цикличен")  
  
 num\_nodes, num\_edges = self.count\_nodes\_and\_edges()  
 print("Проверка древочисленности : ")  
 print(f"Количество узлов : {num\_nodes}, Количество рёбер : {num\_edges}")  
 tree\_structure = (num\_nodes == (num\_edges + 1))  
 if tree\_structure:  
 print("Граф древочисленный")  
 else:  
 print("Граф не древочисленный")  
  
 print("Проверка субцикличности : ")  
 subcyclic = True  
 has\_edges\_added = False  
 nodes = list(self.graph.keys())  
 for i in range(len(nodes) - 1):  
 for j in range(i + 1, len(nodes)):  
 if nodes[j] not in self.graph[nodes[i]]:  
 self.add\_edge(nodes[i], nodes[j])  
 print(f"Добавлено ребро : {nodes[i]} {nodes[j]}.")  
 cycles = self.has\_cycles()  
 cycles\_info(cycles, verbose)  
 if len(cycles) != 1:  
 subcyclic = False  
 has\_edges\_added = True  
 self.remove\_edge(nodes[i], nodes[j])  
  
 if not has\_edges\_added:  
 print("Нет несмежных вершин")  
  
 subcyclic = subcyclic and (  
 has\_edges\_added or len(self.graph) == 1 or len(self.graph) == 2)  
 if subcyclic:  
 print("Граф субциклический")  
 else:  
 print("Граф не субциклический")  
  
 print("Проверка 5 G: ациклический и древочисленный")  
 if acyclic and tree\_structure:  
 print("Граф является ациклическим и древочисленным, то есть граф — это дерево.")  
 elif acyclic:  
 print("Граф является ациклическим, но не древочисленным.")  
 elif tree\_structure:  
 print("Граф является древочисленным, но не ациклическим.")  
 else:  
 print("Граф не является ни ациклическим, ни древочисленным.")  
  
 print("Проверка 6 G: древочисленный и субциклический за( двумя исключениями)")  
 if subcyclic and tree\_structure:  
 if self.validate\_graph\_conditions():  
 print("Граф является исключением.")  
 else:  
 print("Граф является древочисленным и субциклическим за( двумя исключениями), то есть граф — это дерево.")  
 elif tree\_structure:  
 print("Граф является древочисленным, но не субциклическим.")  
 elif subcyclic:  
 print("Граф является субциклическим, но не древочисленным.")  
 else:  
 print("Граф не является ни древочисленным, ни субциклическим.")  
  
 print("Проверка 7 G: ациклический и субциклический")  
 if acyclic and subcyclic:  
 print("Граф является ациклическим и субциклическим, то есть граф — это дерево.")  
 elif acyclic:  
 print("Граф является ациклическим, но не субциклическим.")  
 elif subcyclic:  
 print("Граф является субциклическим, но не ациклическим.")  
 else:  
 print("Граф не является ни ациклическим, ни субциклическим.")  
  
 def load\_tree\_from\_json(self, filename: str):  
 with open(filename, 'r') as f:  
 graph\_data = json.load(f)  
  
 nodes = graph\_data.get('nodes', [])  
 edges = graph\_data.get('edges', [])  
 for node in nodes:  
 self.add\_node(node)  
 for u, v in edges:  
 self.add\_edge(u, v)  
  
def main():  
 graph = Graph()  
 input\_file = 'graph.json'  
  
 try:  
 graph.load\_tree\_from\_json(input\_file)  
 except FileNotFoundError:  
 print(f"Файл {input\_file} не найден.")  
 return  
  
 verbose = True  
 with RedirectPrint('out.log'):  
 graph.check\_tree(verbose)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()