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: Бугайцев М.В. _____
Преподаватель: Новиков Ф. А. _____
Работа принята: Дата _____

Санкт-Петербург
2024

Содержание

1 Введение

1.1 Цели и задачи лабораторной работы:

Проверка свойства древочисленности (субцикличность)

На вход программе подаётся граф. Проверить, является ли граф деревом, используя утверждение 7 из теоремы параграфа “Основные свойства свободных деревьев”. Если нет, то в выходной файл написать, что нарушено: ацикличность (в этом случае предоставить хотя бы один найденный цикл), субцикличность (в этом случае указать для какого ребра это неверно) или и то, и другое. В любом случае проверить, является ли граф древочисленным. По итогу нужно запустить программу на различных графах и увидеть, что либо граф является деревом и древочисленный, либо то, что если ровно одно из условий из утверждения 7 не выполнено, то граф не будет древочисленным, либо то, что если оба условия не выполнены, то граф как может, так и не может быть древочисленным (всё перечисленное должно быть верно за исключениями из утверждения 6).

Здесь используются утверждения 5, 6 и 7 из теоремы параграфа “Основные свойства свободных деревьев”.

2 Теоретическая часть

2.1 Основные понятия графов:

Графом называется пара $G = (V, E)$, где V — множество вершин, а E — множество рёбер, соединяющих пары вершин. Дерево — это связный ацикличный граф, в котором между любыми двумя вершинами существует ровно один путь.

2.2 Описание алгоритма:

Алгоритм проверки, является ли граф деревом, включает следующие шаги:

1. **Проверка ацикличности:** Во время обхода необходимо отслеживать посещённые вершины. Если во время обхода обнаруживается вершина, которая уже была посещена и не является родительской, то граф содержит цикл.

2. **Проверка Древочисленности:** Проверяется количество рёбер: для графа с n вершинами должно быть ровно $n - 1$ рёбер. Если количество рёбер больше или меньше, чем $n - 1$, то граф не является древочисленным.

3. **Проверка субцикличности:** Для каждого несмежного ребра проверяется, не образуется больше или меньше, чем один цикл при добавлении в граф. Если такое ребро найдено, то граф не является субцикличным.

4. **Вывод результатов:** На основе проведённых проверок выводится информация о том, является ли граф деревом.

2.3 Область применения

Данная лабораторная работа направлена на изучение свойств графов, в частности, деревьев и их древочисленности. Применение алгоритмов проверки свойств деревьев имеет широкий спектр в различных областях, таких как:

1. **Компьютерные науки**: Деревья используются в структурах данных, таких как бинарные деревья поиска, AVL-деревья и B-деревья, которые обеспечивают эффективный поиск, вставку и удаление данных.

2. **Сетевые технологии**: В системах передачи данных деревья применяются для маршрутизации и организации сетевых топологий, что способствует снижению задержек и оптимизации использования ресурсов.

3. **Искусственный интеллект**: Деревья решений и деревья разбития используются в машинном обучении для задач классификации и регрессии, что позволяет моделировать сложные взаимосвязи между переменными.

4. **Графовые базы данных**: В графовых системах управления базами данных деревья служат для представления иерархических структур данных, что упрощает выполнение запросов и манипуляции с данными.

5. **Алгоритмы и оптимизация**: Деревья играют важную роль в различных алгоритмах, таких как алгоритмы минимального остовного дерева, которые применяются в задачах оптимизации и планирования.

3 Практическая часть

3.1 Реализация графа

В данной секции рассматривается реализация не ориентированного графа, основанная на концепции списка смежности. Список смежности представляет собой эффективный способ хранения графа, который позволяет организовать данные о вершинах и их связях в удобной и компактной форме.

3.1.1 Список смежности

Список смежности - это структура данных, в которой каждый узел графа содержит информацию о своих соседях, то есть о других узлах, соединенных с ним ребрами.

В отличие от матрицы смежности, список смежности более экономичен, так как хранит только существующие связи.

Существует также концепция списка рёбер, которая описывает граф в виде набора рёбер, где каждое ребро представлено парой вершин (или тройкой, если граф взвешенный). Список смежности ,более удобен для алгоритмов обхода графа, таких как поиск в глубину (DFS) и поиск в ширину (BFS), так как позволяет быстро находить всех соседей конкретного узла.

3.2 Структура JSON файла

Структура JSON файла, используемого для задания графа, включает два основных элемента: nodes и edges.

- nodes: Список узлов графа. Каждый узел представлен уникальным идентификатором. Например, в следующем примере узлы представлены числами от 1 до 5:

  "nodes": [1, 2, 3, 4, 5]

- edges: Список рёбер графа. Каждое ребро представлено в виде массива, содержащего два узла, которые оно соединяет. Например, в следующем примере рёбра соединяют узлы следующим образом:

  "edges": [[1, 2], [1, 3], [2, 4], [1, 5]]

Таким образом, полный пример JSON файла, описывающего граф, выглядит следующим образом:

```
{
   "nodes": [1, 2, 3, 4, 5],
   "edges": [[1, 2], [1, 3], [2, 4], [1, 5]]
}
```

4 Сложности операций в классе Graph

4.1 Инициализация графа (__init__)

Метод __init__ отвечает за создание нового экземпляра класса Graph. При его вызове инициализируется пустой граф, представленный в виде словаря, где ключами являются узлы, а значениями — списки смежных узлов.

- **Временная сложность:** $O(1)$
- **Описание:** Инициализация графа занимает постоянное время, так как создается только пустой словарь.

4.2 Добавление ребра (add_edge)

Метод add_edge отвечает за добавление ребра между двумя узлами графа. При вызове этого метода узлы u и v становятся соседями, и ребро добавляется в обе стороны, так как граф является неориентированным.

- **Временная сложность:** $O(1)$
- **Описание:** Добавление ребра занимает постоянное время, так как операция добавления элемента в список (в данном случае, список смежных узлов) выполняется за $O(1)$. В худшем случае, если узел не существует в графе, он будет создан с пустым списком, что также не влияет на временную сложность.

4.3 Добавление узла (add_node)

Метод add_node отвечает за добавление нового узла в граф. При вызове этого метода, если узел node еще не существует в графе, он добавляется в словарь self.graph с пустым списком смежных узлов.

- **Временная сложность:** $O(1)$
- **Описание:** Добавление узла занимает постоянное время, так как проверка наличия узла в словаре и добавление нового ключа выполняются за $O(1)$. Если узел уже существует, метод просто завершает выполнение без изменений.

4.4 Удаление ребра (remove_edge)

Метод remove_edge отвечает за удаление ребра между двумя узлами графа. При вызове этого метода, если узлы u и v существуют в графе и между ними есть ребро, оно удаляется в обе стороны, так как граф является неориентированным.

- **Временная сложность:** $O(n)$
- **Описание:** Удаление элемента из списка требует линейного времени в худшем случае, так как необходимо пройти по списку смежных узлов, чтобы найти и удалить нужный элемент. Здесь n — это количество смежных узлов для узла u или v. Если узлы не существуют или ребро отсутствует, метод завершает выполнение без изменений.

4.5 Подсчет узлов и ребер (count_nodes_and_edges)

Метод count_nodes_and_edges отвечает за подсчет количества узлов и ребер в графе. При вызове этого метода он возвращает кортеж, содержащий количество узлов и количество ребер.

- **Временная сложность:** $O(n)$
- **Описание:** Подсчет количества узлов выполняется за $O(1)$. Подсчет количества ребер требует прохода по всем узлам, что занимает $O(n)$, где n — это количество узлов в графе. Поскольку каждое ребро учитывается дважды (по одному разу для каждого узла), итоговое количество ребер делится на 2. Таким образом, общая временная сложность метода составляет $O(n)$.

4.6 Проверка условий графа (validate_graph_conditions)

Метод validate_graph_conditions отвечает за проверку условий, которые должны выполняться для графа. Он использует алгоритм обхода в глубину (DFS) для определения компонентов связности графа и подсчета количества узлов и ребер в каждом компоненте.

- **Временная сложность:** $O(n + m)$
- **Описание:** Метод проходит по всем узлам графа и использует стек для реализации DFS, что позволяет обойти все узлы. Временная сложность составляет $O(n)$ для обхода узлов и $O(m)$ для подсчета ребер, где n — количество узлов, а m — количество ребер. Так как в графе может быть много ребер, общая временная сложность метода составляет $O(n + m)$.

4.7 Проверка наличия циклов (has_cycles)

Метод has_cycles проверяет наличие циклов в графе, используя указанный алгоритм обхода. Он собирает все найденные циклы и возвращает их в отсортированном виде.

- **Временная сложность:** $O((V + E)(C + 1))$, где V— количество вершин, E — количество рёбер, а C — количество найденных циклов.
- **Пространственная сложность:** $O(V + C \cdot V)$, где V — количество вершин, а C — количество найденных циклов.

Метод включает следующие вспомогательные функции:

4.7.1 Переворот подмассива (reverse)

Метод reverse отвечает за переворот части списка arr между индексами start и end. Он изменяет порядок элементов в указанном диапазоне на месте.

- **Временная сложность:** $O(k)$
- **Пространственная сложность:** $O(1)$

Описание:

- Временная сложность $O(k)$, где k — количество элементов между индексами start и end, так как метод проходит по этим элементам один раз.
- Пространственная сложность $O(1)$, так как метод использует фиксированное количество дополнительных переменных для выполнения операции переворота, не требуя дополнительной памяти, пропорциональной размеру входных данных.

4.7.2 Поворот массива (rotate)

Метод rotate отвечает за поворот массива arr так, чтобы минимальный элемент оказался в начале. Если следующий элемент после минимального меньше предыдущего, массив переворачивается. В противном случае массив поворачивается с использованием вспомогательной функции reverse.

- **Временная сложность:** $O(n)$
- **Пространственная сложность:** $O(1)$

Описание:

- Временная сложность $O(n)$, где n — количество элементов в массиве. Метод включает операции поиска минимального элемента и переворота подмассивов, каждая из которых требует линейного времени.
- Пространственная сложность $O(1)$, так как метод использует фиксированное количество дополнительных переменных и не требует дополнительной памяти, пропорциональной размеру входных данных.

4.7.3 Добавление цикла (add_cycle)

Метод add_cycle отвечает за добавление цикла в граф. Он принимает узел neighbor и путь path, из которого извлекается цикл, начиная с указанного узла. Цикл нормализуется с помощью функции rotate, чтобы избежать дублирования.

- **Временная сложность:** $O(k)$
- **Пространственная сложность:** $O(k)$

Описание:

- Временная сложность $O(k)$, где k — длина цикла. Метод включает операции поиска индекса узла и поворота массива, каждая из которых требует линейного времени.
- Пространственная сложность $O(k)$, так как создается новый список для хранения цикла и его нормализованной версии.

4.7.4 Итеративный обход в глубину (dfs_iter)

Метод dfs_iter выполняет итеративный обход в глубину (DFS) графа, начиная с узла start. Он использует стек для хранения узлов и их свойств, таких как родительский узел, глубина и множество посещенных узлов. Метод также отслеживает текущий путь и добавляет циклы, если они обнаружены.

- **Временная сложность:** $O((V + E)(C + 1))$, где V— количество вершин, E — количество рёбер, а C — количество найденных циклов.
- **Пространственная сложность:** $O(V + C \cdot V)$, где V — количество вершин, а C — количество найденных циклов.

Описание:

- Метод использует стек для хранения узлов, которые необходимо посетить, что позволяет избежать рекурсивного вызова и управлять обходом графа итеративно.
- При посещении узла метод проверяет, не был ли он уже посещён, и если да, то проверяет наличие цикла, добавляя его в список найденных циклов.
- Стек также хранит информацию о родительских узлах и глубине, что позволяет отслеживать текущий путь и возвращаться к предыдущим узлам при необходимости.

4.8 Проверка свойств графа (check_tree)

Метод check_tree проверяет различные свойства графа, включая ацикличность, древочисленность и субцикличность. Он выводит информацию о найденных циклах и проверяет, является ли граф деревом.

- **Временная сложность:** $O((V + E)(C) \cdot (V^2)$, где V— количество вершин, E — количество рёбер, а C — количество найденных циклов.
- **Пространственная сложность:** $O(V + C \cdot V)$, где V — количество вершин, а C — количество найденных циклов.

Описание:

- Временная сложность $O((V + E)(C) \cdot (V^2))$ так как наибольшую сложность имеет метод has_cycles который вызывается 1 раз в проверке ацикличности V^2 раз в проверке субцикличности.
- Пространственная сложность $O(V + C \cdot V)$ равна необходимому для работы и хранения результатов метода has_cycles.

Метод выполняет следующие проверки:

1. Ацикличность: Использует метод has_cycles для определения наличия циклов и выводит соответствующую информацию.
2. Древочисленность: Проверяет, соответствует ли количество узлов и рёбер условию $n = m+1$.
3. Субцикличность: Проверяет, можно ли добавить рёбер между несмежными узлами, не создавая более одного цикла.
4. Проверка 5G: Определяет, является ли граф ациклическим и древочисленным.
5. Проверка 6G: Определяет, является ли граф древочисленным и субциклическим (за двумя исключениями).
6. Проверка 7G: Определяет, является ли граф ациклическим и субциклическим.

Метод выводит результаты всех проверок, предоставляя информацию о свойствах графа.

5 Примеры использования

5.1 Ациклический древочисленный субциклический граф

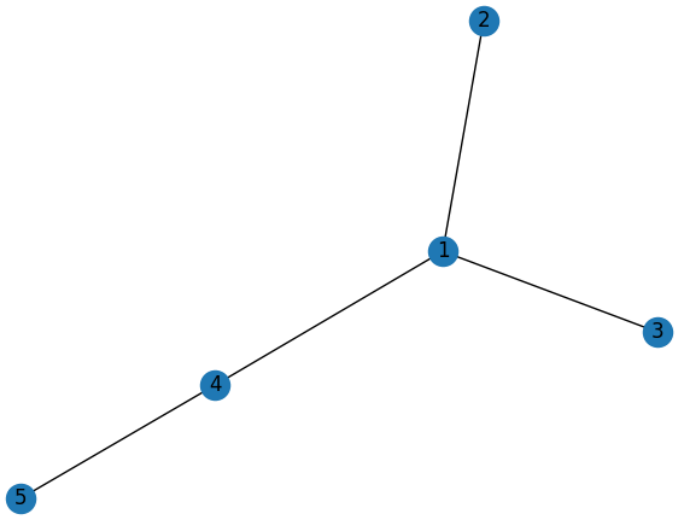

Рис. 1: Пример ациклического древочисленного и субциклического графа

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.
  - Добавляем соседа 4 в стек.
- Посещаем узел: 4, Родитель: 1, Глубина: 1
- Добавляем узел 4 в путь. Текущий путь: [1, 4]
  - Добавляем соседа 5 в стек.
- Посещаем узел: 5, Родитель: 4, Глубина: 2
- Добавляем узел 5 в путь. Текущий путь: [1, 4, 5]
- Посещаем узел: 3, Родитель: 1, Глубина: 1
- Удаляем узел 4 из пути, так как он больше не актуален.
- Удаляем узел 5 из пути, так как он больше не актуален.
- Добавляем узел 3 в путь. Текущий путь: [1, 3]
- Посещаем узел: 2, Родитель: 1, Глубина: 1
- Удаляем узел 3 из пути, так как он больше не актуален.
- Добавляем узел 2 в путь. Текущий путь: [1, 2]
- Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5

Граф не содержит цикл

Граф ацикличен

Проверка древочисленности: Количество узлов: 5, Количество рёбер: 4
Граф древочисленный

Проверка субцикличности:

- Добавлено ребро: $1 \leftrightarrow 5$.

  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.
    - Добавляем соседа 4 в стек.
    - Добавляем соседа 5 в стек.
  - Посещаем узел: 5, Родитель: 1, Глубина: 1
  - Добавляем узел 5 в путь. Текущий путь: [1, 5]
    - Добавляем соседа 4 в стек.
  - Посещаем узел: 4, Родитель: 5, Глубина: 2
  - Добавляем узел 4 в путь. Текущий путь: [1, 5, 4]
    - Обнаружен цикл через соседа: 1
    - Добавлен цикл в ответ: [1, 4, 5, 1]
  - Посещаем узел: 4, Родитель: 1, Глубина: 1
  - Удаляем узел 5 из пути, так как он больше не актуален.
  - Удаляем узел 4 из пути, так как он больше не актуален.
  - Добавляем узел 4 в путь. Текущий путь: [1, 4]
    - Добавляем соседа 5 в стек.
  - Посещаем узел: 5, Родитель: 4, Глубина: 2
  - Добавляем узел 5 в путь. Текущий путь: [1, 4, 5]
    - Обнаружен цикл через соседа: 1
  - Посещаем узел: 3, Родитель: 1, Глубина: 1
  - Удаляем узел 4 из пути, так как он больше не актуален.
  - Удаляем узел 5 из пути, так как он больше не актуален.
  - Добавляем узел 3 в путь. Текущий путь: [1, 3]
  - Посещаем узел: 2, Родитель: 1, Глубина: 1
  - Удаляем узел 3 из пути, так как он больше не актуален.
  - Добавляем узел 2 в путь. Текущий путь: [1, 2]
  - Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5

  Граф содержит 1 простых циклов: $1 \to 4 \to 5 \to 1$

- Удалено ребро: $1 \leftrightarrow 5$.
- Добавлено ребро: $2 \leftrightarrow 3$.

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.
  - Добавляем соседа 4 в стек.
- Посещаем узел: 4, Родитель: 1, Глубина: 1
- Добавляем узел 4 в путь. Текущий путь: [1, 4]
  - Добавляем соседа 5 в стек.
- Посещаем узел: 5, Родитель: 4, Глубина: 2
- Добавляем узел 5 в путь. Текущий путь: [1, 4, 5]
- Посещаем узел: 3, Родитель: 1, Глубина: 1
- Удаляем узел 4 из пути, так как он больше не актуален.
- Удаляем узел 5 из пути, так как он больше не актуален.
- Добавляем узел 3 в путь. Текущий путь: [1, 3]
  - Добавляем соседа 2 в стек.
- Посещаем узел: 2, Родитель: 3, Глубина: 2
- Добавляем узел 2 в путь. Текущий путь: [1, 3, 2]
  - Обнаружен цикл через соседа: 1
  - Добавлен цикл в ответ: [1, 2, 3, 1]
- Посещаем узел: 2, Родитель: 1, Глубина: 1
- Удаляем узел 3 из пути, так как он больше не актуален.
- Удаляем узел 2 из пути, так как он больше не актуален.
- Добавляем узел 2 в путь. Текущий путь: [1, 2]
  - Добавляем соседа 3 в стек.
- Посещаем узел: 3, Родитель: 2, Глубина: 2
- Добавляем узел 3 в путь. Текущий путь: [1, 2, 3]
  - Обнаружен цикл через соседа: 1
- Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5

Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$

- Удалено ребро: $2 \leftrightarrow 3$.
- Добавлено ребро: $2 \leftrightarrow 4$.

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.
  - Добавляем соседа 4 в стек.
- Посещаем узел: 4, Родитель: 1, Глубина: 1
- Добавляем узел 4 в путь. Текущий путь: [1, 4]
  - Добавляем соседа 2 в стек.
  - Добавляем соседа 5 в стек.

- Посещаем узел: 5, Родитель: 4, Глубина: 2
- Добавляем узел 5 в путь. Текущий путь: [1, 4, 5]
- Посещаем узел: 2, Родитель: 4, Глубина: 2
- Удаляем узел 5 из пути, так как он больше не актуален.
- Добавляем узел 2 в путь. Текущий путь: [1, 4, 2]
  * Обнаружен цикл через соседа: 1
  * Добавлен цикл в ответ: [1, 2, 4, 1]
- Посещаем узел: 3, Родитель: 1, Глубина: 1
- Удаляем узел 4 из пути, так как он больше не актуален.
- Удаляем узел 2 из пути, так как он больше не актуален.
- Добавляем узел 3 в путь. Текущий путь: [1, 3]
- Посещаем узел: 2, Родитель: 1, Глубина: 1
- 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]
  * Обнаружен цикл через соседа: 1
  * Добавляем соседа 5 в стек.
- Посещаем узел: 5, Родитель: 4, Глубина: 3
- Добавляем узел 5 в путь. Текущий путь: [1, 2, 4, 5]
- Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5

Граф содержит 1 простых циклов: $1 \to 2 \to 4 \to 1$

- Удалено ребро: $2 \leftrightarrow 4$.
- Добавлено ребро: $2 \leftrightarrow 5$.

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.
  * Добавляем соседа 4 в стек.
- Посещаем узел: 4, Родитель: 1, Глубина: 1
- Добавляем узел 4 в путь. Текущий путь: [1, 4]
  * Добавляем соседа 5 в стек.
- Посещаем узел: 5, Родитель: 4, Глубина: 2
- Добавляем узел 5 в путь. Текущий путь: [1, 4, 5]
  * Добавляем соседа 2 в стек.
- Посещаем узел: 2, Родитель: 5, Глубина: 3
- Добавляем узел 2 в путь. Текущий путь: [1, 4, 5, 2]
  * Обнаружен цикл через соседа: 1
  * Добавлен цикл в ответ: [1, 2, 5, 4, 1]
- Посещаем узел: 3, Родитель: 1, Глубина: 1

- Удаляем узел 4 из пути, так как он больше не актуален.
- Удаляем узел 5 из пути, так как он больше не актуален.
- Удаляем узел 2 из пути, так как он больше не актуален.
- Добавляем узел 3 в путь. Текущий путь: [1, 3]
- Посещаем узел: 2, Родитель: 1, Глубина: 1
- Удаляем узел 3 из пути, так как он больше не актуален.
- Добавляем узел 2 в путь. Текущий путь: [1, 2]
  * Добавляем соседа 5 в стек.
- Посещаем узел: 5, Родитель: 2, Глубина: 2
- Добавляем узел 5 в путь. Текущий путь: [1, 2, 5]
  * Добавляем соседа 4 в стек.
- Посещаем узел: 4, Родитель: 5, Глубина: 3
- Добавляем узел 4 в путь. Текущий путь: [1, 2, 5, 4]
  * Обнаружен цикл через соседа: 1
- Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5

Граф содержит 1 простых циклов: $1 \to 2 \to 5 \to 4 \to 1$

- Удалено ребро: $2 \leftrightarrow 5$.
- Добавлено ребро: $3 \leftrightarrow 4$.

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.
  * Добавляем соседа 4 в стек.
- Посещаем узел: 4, Родитель: 1, Глубина: 1
- Добавляем узел 4 в путь. Текущий путь: [1, 4]
  * Добавляем соседа 3 в стек.
  * Добавляем соседа 5 в стек.
- Посещаем узел: 5, Родитель: 4, Глубина: 2
- Добавляем узел 5 в путь. Текущий путь: [1, 4, 5]
- Посещаем узел: 3, Родитель: 4, Глубина: 2
- Удаляем узел 5 из пути, так как он больше не актуален.
- Добавляем узел 3 в путь. Текущий путь: [1, 4, 3]
  * Обнаружен цикл через соседа: 1
  * Добавлен цикл в ответ: [1, 3, 4, 1]
- Посещаем узел: 3, Родитель: 1, Глубина: 1
- Удаляем узел 4 из пути, так как он больше не актуален.
- Удаляем узел 3 из пути, так как он больше не актуален.
- Добавляем узел 3 в путь. Текущий путь: [1, 3]
  * Добавляем соседа 4 в стек.
- Посещаем узел: 4, Родитель: 3, Глубина: 2
- Добавляем узел 4 в путь. Текущий путь: [1, 3, 4]

    - Обнаружен цикл через соседа: 1
    - Добавляем соседа 5 в стек.
  - Посещаем узел: 5, Родитель: 4, Глубина: 3
  - Добавляем узел 5 в путь. Текущий путь: [1, 3, 4, 5]
  - Посещаем узел: 2, Родитель: 1, Глубина: 1
  - Удаляем узел 3 из пути, так как он больше не актуален.
  - Удаляем узел 4 из пути, так как он больше не актуален.
  - Удаляем узел 5 из пути, так как он больше не актуален.
  - Добавляем узел 2 в путь. Текущий путь: [1, 2]
  - Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5

  Граф содержит 1 простых циклов: $1 \to 3 \to 4 \to 1$

- Удалено ребро: $3 \leftrightarrow 4$.
- Добавлено ребро: $3 \leftrightarrow 5$.

  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.
    - Добавляем соседа 4 в стек.
  - Посещаем узел: 4, Родитель: 1, Глубина: 1
  - Добавляем узел 4 в путь. Текущий путь: [1, 4]
    - Добавляем соседа 5 в стек.
  - Посещаем узел: 5, Родитель: 4, Глубина: 2
  - Добавляем узел 5 в путь. Текущий путь: [1, 4, 5]
    - Добавляем соседа 3 в стек.
  - Посещаем узел: 3, Родитель: 5, Глубина: 3
  - Добавляем узел 3 в путь. Текущий путь: [1, 4, 5, 3]
    - Обнаружен цикл через соседа: 1
    - Добавлен цикл в ответ: [1, 3, 5, 4, 1]
  - Посещаем узел: 3, Родитель: 1, Глубина: 1
  - Удаляем узел 4 из пути, так как он больше не актуален.
  - Удаляем узел 5 из пути, так как он больше не актуален.
  - Удаляем узел 3 из пути, так как он больше не актуален.
  - 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 4 в стек.
  - Посещаем узел: 4, Родитель: 5, Глубина: 3
  - Добавляем узел 4 в путь. Текущий путь: [1, 3, 5, 4]
    - Обнаружен цикл через соседа: 1
  - Посещаем узел: 2, Родитель: 1, Глубина: 1

– Удаляем узел 3 из пути, так как он больше не актуален.
– Удаляем узел 5 из пути, так как он больше не актуален.
– Удаляем узел 4 из пути, так как он больше не актуален.
– Добавляем узел 2 в путь. Текущий путь: [1, 2]
– Завершен обход для узла 1. Посещенные узлы: 1, 2, 3, 4, 5

Граф содержит 1 простых циклов: $1 \to 3 \to 5 \to 4 \to 1$

- Удалено ребро: $3 \leftrightarrow 5$.

Граф субциклический

Проверка 5 G: ациклический и древочисленный Граф является ациклическим и древочисленным, то есть граф — это дерево.

Проверка 6 G: древочисленный и субциклический (за двумя исключениями) Граф является древочисленным и субциклическим (за двумя исключениями), то есть граф — это дерево.

Проверка 7 G: ациклический и субциклический Граф является ациклическим и субциклическим, то есть граф — это дерево.

5.2 Циклический древочисленный и субциклический граф

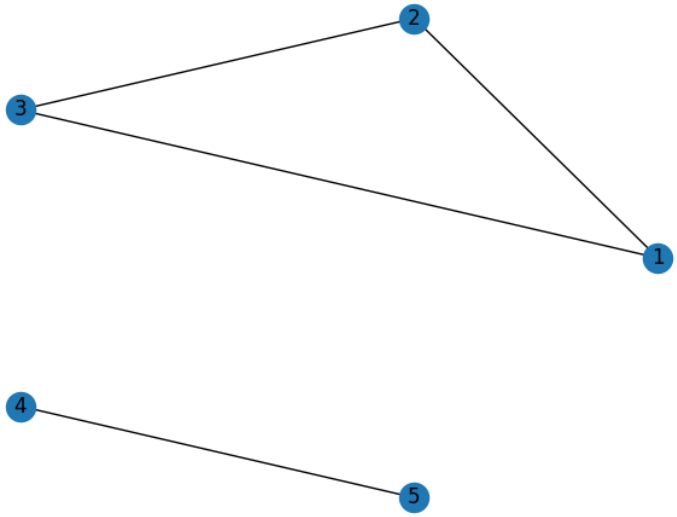

Рис. 2: Пример циклического древочисленного и субциклического графа

Проверка ацикличности: Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
Граф цикличен

Проверка древочисленности: Количество узлов: 5, Количество рёбер: 4
Граф древочисленный

Проверка субцикличности:

- Добавлено ребро: $1 \leftrightarrow 4$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Добавлено ребро: $1 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Добавлено ребро: $2 \leftrightarrow 4$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Добавлено ребро: $2 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Добавлено ребро: $3 \leftrightarrow 4$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Добавлено ребро: $3 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$

Граф субциклический

Проверка 5 G: ациклический и древочисленный Граф является древочисленным, но не ациклическим.

Проверка 6 G: древочисленный и субциклический (за двумя исключениями) Граф является исключением.

Проверка 7 G: ациклический и субциклический Граф является субциклическим, но не ациклическим.

5.3 Ациклический не древочисленный и не субциклический граф

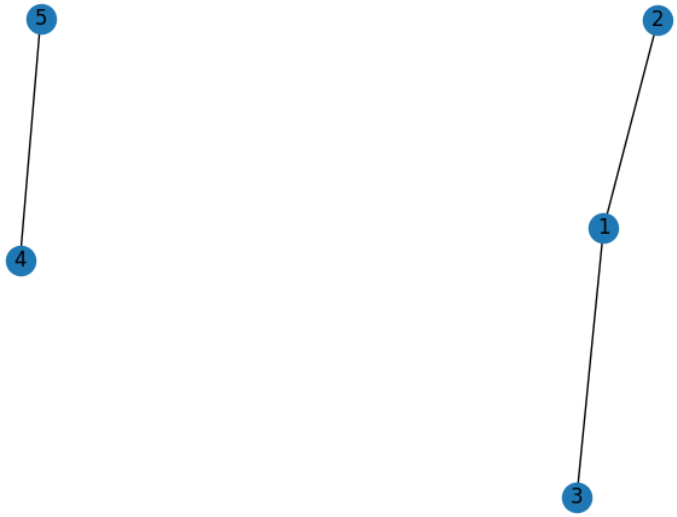

Рис. 3: Пример ациклического не древочисленного и не субциклического графа

Проверка ацикличности: Граф не содержит цикл
Граф ацикличен

Проверка древочисленности: Количество узлов: 5, Количество рёбер: 3
Граф не древочисленный

Проверка субцикличности:

- Добавлено ребро: $1 \leftrightarrow 4$. Граф не содержит цикл
- Добавлено ребро: $1 \leftrightarrow 5$. Граф не содержит цикл
- Добавлено ребро: $2 \leftrightarrow 3$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Добавлено ребро: $2 \leftrightarrow 4$. Граф не содержит цикл
- Добавлено ребро: $2 \leftrightarrow 5$. Граф не содержит цикл
- Добавлено ребро: $3 \leftrightarrow 4$. Граф не содержит цикл
- Добавлено ребро: $3 \leftrightarrow 5$. Граф не содержит цикл

Граф не субциклический

Проверка 5 G: ациклический и древочисленный Граф является ациклическим, но не древочисленным.

Проверка 6 G: древочисленный и субциклический (за двумя исключениями) Граф не является ни древочисленным, ни субциклическим.

Проверка 7 G: ациклический и субциклический Граф является ациклическим, но не субциклическим.

5.4 Циклический древочисленный и не субциклический граф

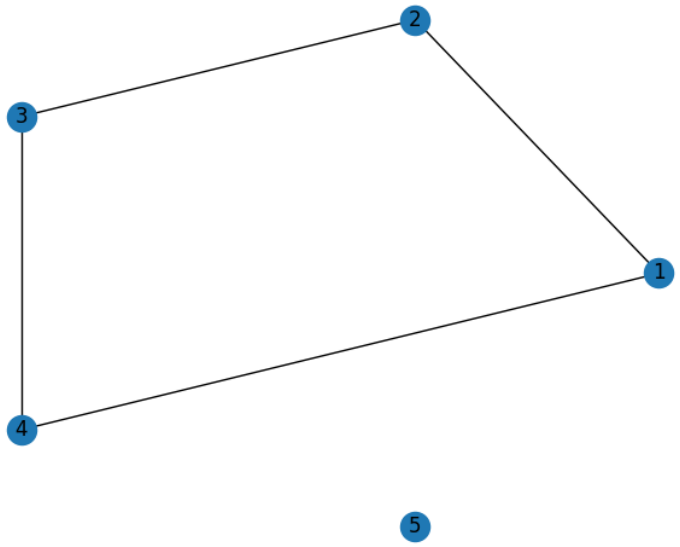

Рис. 4: Пример циклического не древочисленного и не субциклического графа

Проверка ацикличности: Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 4 \to 1$
Граф цикличен

Проверка древочисленности: Количество узлов: 5, Количество рёбер: 4
Граф древочисленных

Проверка субцикличности:

- Добавлено ребро: $1 \leftrightarrow 3$. Граф содержит 3 простых циклов: $1 \to 2 \to 3 \to 1$, $1 \to 2 \to 3 \to 4 \to 1$, $1 \to 3 \to 4 \to 1$
- Добавлено ребро: $1 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 4 \to 1$
- Добавлено ребро: $2 \leftrightarrow 4$. Граф содержит 3 простых циклов: $1 \to 2 \to 3 \to 4 \to 1$, $1 \to 2 \to 4 \to 1$, $2 \to 3 \to 4 \to 2$
- Добавлено ребро: $2 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 4 \to 1$
- Добавлено ребро: $3 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 4 \to 1$
- Добавлено ребро: $4 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 4 \to 1$

Граф не субциклический

Проверка 5 G: ациклический и древочисленный Граф является древочисленным, но не ациклическим.

Проверка 6 G: древочисленный и субциклический (за двумя исключениями) Граф является древочисленным, но не субциклическим.

Проверка 7 G: ациклический и субциклический Граф не является ни ациклическим, ни субциклическим.

5.5 Циклический не древочисленный и не субциклический граф

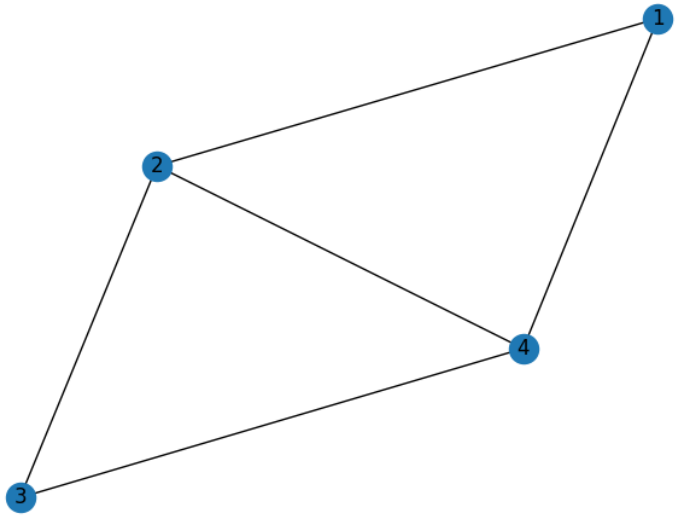

Рис. 5: Пример циклического не древочисленного и не субциклического графа

Проверка ацикличности: Граф содержит 3 простых циклов: $1 \to 2 \to 3 \to 4 \to 1$, $1 \to 2 \to 4 \to 1$, $2 \to 3 \to 4 \to 2$

Граф цикличен

Проверка древочисленности: Количество узлов: 4, Количество рёбер: 5

Граф не древочисленный

Проверка субцикличности:

- Добавлено ребро: $1 \leftrightarrow 3$. Граф содержит 7 простых циклов: $1 \to 2 \to 3 \to 1$, $1 \to 2 \to 3 \to 4 \to 1$, $1 \to 2 \to 4 \to 1$, $1 \to 2 \to 4 \to 3 \to 1$, $1 \to 3 \to 2 \to 4 \to 1$, $1 \to 3 \to 4 \to 1$, $2 \to 3 \to 4 \to 2$

Граф не субциклический

Проверка 5 G: ациклический и древочисленный Граф не является ни ациклическим, ни древочисленным.

Проверка 6 G: древочисленный и субциклический (за двумя исключениями) Граф не является ни древочисленным, ни субциклическим.

Проверка 7 G: ациклический и субциклический Граф не является ни ациклическим, ни субциклическим.

5.6 Циклический не древочисленный и субциклический граф

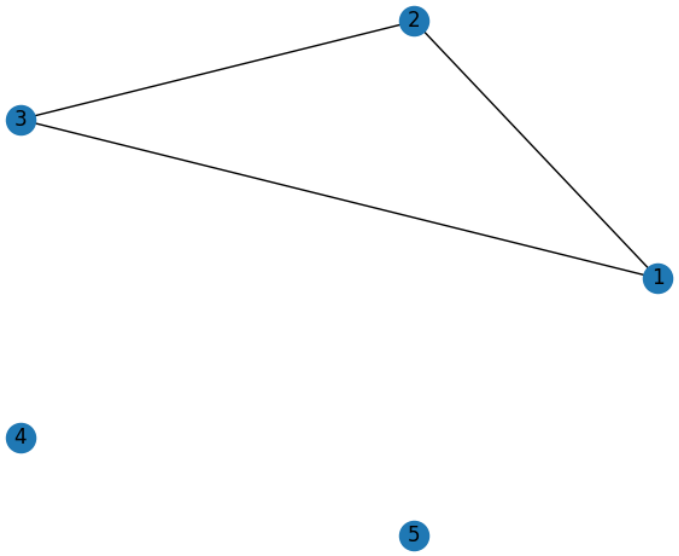

Рис. 6: Пример циклического не древочисленного и субциклического графа

Проверка ацикличности: Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
Граф цикличен

Проверка древочисленности: Количество узлов: 5, Количество рёбер: 3
Граф не древочисленный

Проверка субцикличности:

- Добавлено ребро: $1 \leftrightarrow 4$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Удалено ребро: $1 \leftrightarrow 4$.
- Добавлено ребро: $1 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Удалено ребро: $1 \leftrightarrow 5$.
- Добавлено ребро: $2 \leftrightarrow 4$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Удалено ребро: $2 \leftrightarrow 4$.
- Добавлено ребро: $2 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Удалено ребро: $2 \leftrightarrow 5$.
- Добавлено ребро: $3 \leftrightarrow 4$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Удалено ребро: $3 \leftrightarrow 4$.
- Добавлено ребро: $3 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Удалено ребро: $3 \leftrightarrow 5$.
- Добавлено ребро: $4 \leftrightarrow 5$. Граф содержит 1 простых циклов: $1 \to 2 \to 3 \to 1$
- Удалено ребро: $4 \leftrightarrow 5$.

Граф субциклический

Проверка 5 G: ациклический и древочисленный Граф не является ни ациклическим, ни древочисленным.

Проверка 6 G: древочисленный и субциклический (за двумя исключениями) Граф является субциклическим, но не древочисленным.

Проверка 7 G: ациклический и субциклический Граф является субциклическим, но не ациклическим.

6 Интерфейс приложения

6.1 Пример запуска

Приложение можно запустить из командной строки с использованием следующих аргументов:

- –nodes - список узлов графа в формате: узел1.
- –edges - список рёбер графа в формате: узел1 узел2.
- –json-load - загрузка графа из указанного JSON файла.
- -v или –verbose - вывод дополнительной информации о процессе выполнения.

6.1.1 Задание рёбер графа

В этом примере мы задаем граф, указывая его рёбра через параметр –edges. Узлы графа будут автоматически добавлены на основе указанных рёбер.

python main.py --edges A B C D E F

В данном случае приложение создаст граф с рёбрами A-B, C-D и E-F, используя алгоритм по умолчанию dfs_iter для проверки циклов.

6.1.2 Задание узлов и рёбер графа

Если необходимо явно указать узлы графа, можно использовать параметр –nodes вместе с –edges.

python main.py --nodes A B C D --edges A B C D

В этом примере мы создаем граф с узлами A, B, C и D, а также рёбрами A-B и C-D.

6.1.3 Загрузка графа из JSON файла

Если граф уже сохранен в формате JSON, его можно загрузить с помощью параметра –json-load.

python main.py --json-load graph.json

В этом случае приложение загрузит граф из файла graph.json и выполнит проверку циклов с использованием алгоритма по умолчанию.

6.1.4 Включение режима подробного вывода

Для получения дополнительной информации о процессе выполнения можно использовать флаг -v или –verbose.

python main.py --edges A B C D -v

При использовании этого флага приложение будет выводить подробную информацию о процессе проверки свойств графа.

7 Заключение

В ходе выполнения лабораторной работы была реализована программа для проверки графа с использованием утверждений 5, 6 и 7 из теоремы параграфа “Основные свойства свободных деревьев”.

Программа позволяет:

- Проверять граф на выполнение условий ацикличности, древочисленности и субцикличности.
- Определять, является ли граф деревом, на основании выполнения условий.
- Выводить подробные сообщения об ошибках, если граф не соответствует необходимым условиям.
- Обрабатывать графы различного размера и сложности, что делает программу универсальным инструментом для работы с графами.

Результаты работы программы подтвердили теоретические положения:

1. Если нарушено хотя бы одно из условий (древочисленность, ацикличность или субцикличность), граф не будет деревом.
2. Программа успешно справляется с графами, содержащими циклы, и корректно определяет их несоответствие условиям деревьев.
3. Программа предоставляет пользователю возможность визуализировать структуру графа.

8 Приложение

В этом разделе приведен код программы.

```
from collections import defaultdict, deque
import sys
import argparse
import json


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):
        sys.stdout = self.original_stdout
        self.file.close()
        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 reverse(arr: list, start: int, end: int):
                while start < end:
                    arr[start], arr[end] = arr[end], arr[start]
                    start += 1
                    end -= 1

            def rotate(arr: list):
                min_index = arr.index(min(arr))
                n = len(arr)
                if arr[(min_index + 1) % n] > arr[min_index - 1]:
                    min_index = n - min_index - 1
                else:
                    arr.reverse()
                reverse(arr, 0, n - min_index - 1)
```

```
                reverse(arr, n - min_index, n - 1)
                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 draw(self):
        import matplotlib.pyplot as plt
        import networkx as nx
        G = nx.Graph()
        for n in self.graph.keys():
            G.add_node(n)
        for u in self.graph:
            for v in self.graph[u]:
                G.add_edge(u, v)
        pos = nx.spring_layout(G)
        pos = nx.circular_layout(G)
        nx.draw(G, pos, with_labels=True)
        plt.show()

    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():
    parser = argparse.ArgumentParser(
        description='Проверка свойства древочисленности субцикличность() графа.')

    graph_group = parser.add_argument_group('Граф', 'Способы задания графа')
    graph_group.add_argument(
        '--edges',
        nargs='+',
        help='Список рёбер графа в формате: узел1 узел2 например(: A B C D - для рёбер A-B
    и C-D)',
        required=False)
    graph_group.add_argument('--nodes', nargs='+',
                             help='Список узлов графа например(: A B C D)',
                             required=False)
    graph_group.add_argument(
        '--json-load',
        help='Загрузить граф из JSON файла например(: graph.json)',
        default='graph.json',
```

```
271             required=False)
272
273     options_group = parser.add_argument_group(
274         'Опции', 'Дополнительные параметры')
275     options_group.add_argument(
276         '-v',
277         '--verbose',
278         action='store_true',
279         help='Выводить дополнительную информацию о процессе')
280     args = parser.parse_args()
281
282     g = Graph()
283
284     if args.edges:
285         for node in args.nodes:
286             g.add_node(node)
287         for i in range(0, len(args.edges), 2):
288             g.add_edge(args.edges[i], args.edges[i + 1])
289     elif args.json_load:
290         g.load_tree_from_json(args.json_load)
291     else:
292         raise ValueError("Unknown algorithm")
293
294     with RedirectPrint('output.txt'):
295         g.check_tree(args.verbose)
296
297     if args.verbose:
298         g.draw()
299
300
301 if __name__ == "__main__":
302     main()
```

Листинг 1: Код программы на Python