Санкт-Петербургский Политехнический Университет

Высшая школа прикладной математики и вычислительной физики, ФизМех

01.03.02 Прикладная математика и информатика

Лабораторная работа №3

Дисциплина “Дискретная математика”

Тема “Деревья”

Вариант “Проверка свойства древочисленности (субцикличность)”

Выполнил студент гр. 5030102/20201 Мелко Тимофей Андреевич

**Поставленная задача**

Проверить является ли граф деревом, ацикличность, субцикличность, древочисленность.

**Используемый язык программирования**

Python 3.12.6

**Описание проверки свойств**

Граф без циклов называется ациклическим. Будем с помощью алгоритма DFS обходить граф и искать в нем цикл. В случае если цикл найдем, будем записывать любой цикл.

Проверка ацикличности:

Функция is\_acyclic(graph):

1. Создаём пустое множество visited.

2. Создаём пустой словарь parent.

3. Для каждой вершины v от 0 до V - 1:

3.1. Если вершина v не была посещена:

- Создаём стек stack и кладём в него пару (v, -1).

3.2. Пока стек не пуст:

- Извлекаем из стека пару (current, prev).

- Если current не в visited:

- Добавляем current в visited.

- Записываем parent[current] = prev.

- Для каждого соседа neighbor вершины current:

- Если adj\_matrix[current][neighbor] == 1 (есть ребро):

- Если neighbor ещё не посещён:

- Кладём (neighbor, current) в стек.

- Иначе, если neighbor уже посещён и не равен prev:

- Найден цикл. Возвращаем False.

4. Если цикл не найден после обхода всех вершин, возвращаем True.

Нахождение цикла в графе

Функция find\_cycle

- adj\_matrix: Матрица смежности графа.

- V: Количество вершин графа.

Шаги выполнения:

1. Инициализация:

- Создать пустое множество visited для отслеживания посещённых вершин.

- Создать пустой словарь parent для хранения связей между вершинами.

- Инициализировать cycle как пустой список.

2. Определение вспомогательной функции dfs(start):

- Создать стек stack, содержащий кортеж (start, -1) (текущая вершина и её родитель).

- Пока стек не пуст:

- Извлечь вершину vertex и её родителя prev из стека.

- Если vertex ещё не посещена:

- Добавить vertex в visited.

- Установить parent[vertex] = prev.

- Для каждой вершины neighbor от 0 до V-1:

- Если существует ребро adj\_matrix[vertex][neighbor] == 1:

- Если neighbor не посещён, добавить (neighbor, vertex) в стек.

- Если neighbor уже посещён и neighbor != prev:

- Вызвать функцию extract\_cycle(vertex, neighbor, parent) для извлечения цикла.

- Вернуть цикл.

- Если цикл не найден, вернуть None.

3. Итерация по всем вершинам графа:

- Для каждой вершины vertex от 0 до V-1:

- Если vertex ещё не посещена:

- Вызвать dfs(vertex).

- Если цикл найден, вернуть его.

4. Возврат результата:

- Если после проверки всех вершин цикл не найден, вернуть None.

Вспомогательная функция: extract\_cycle(start, end, parent)

- start и end: Вершины, между которыми обнаружен цикл.

- parent: Словарь связей "ребёнок-родитель".

Шаги выполнения:

1. Инициализировать пустой список cycle.

2. Построить путь от start до end:

- Пока start не равен end:

- Добавить start в cycle.

- Обновить start = parent[start].

3. Добавить end в cycle.

4. Замкнуть цикл, добавив первую вершину в конец списка.

5. Развернуть cycle для правильного порядка.

6. Вернуть cycle.

Если граф G + x имеет ровно один простой цикл, z(G + x) = 1, то граф G называется субциклическим. Будем добавлять ребро в граф и проверять количество циклов в графе после добавления. В случае если не выполняется записываем ребро для которого не выполняется.

Проверка субцикличности:  
Функция is\_subcyclic(graph):

max\_cycles – общее количество циклов

1. Для каждой пары вершин (u, v), где u != v:

1.1. Если между u и v нет ребра (adj\_matrix[u][v] == 0):

- Временно добавляем ребро:

adj\_matrix[u][v] = 1

adj\_matrix[v][u] = 1

- Подсчитываем количество циклов в графе:

num\_cycles = count\_cycles()

max\_cycles = max(max\_cylces, num\_cyclse)

- Если num\_cycles > 1:

- Удаляем добавленное ребро:

adj\_matrix[u][v] = 0

adj\_matrix[v][u] = 0

- Возвращаем False и пару (u, v).

- Иначе:

- Удаляем добавленное ребро:

adj\_matrix[u][v] = 0

adj\_matrix[v][u] = 0

Псевдокод для подсчёта количества уникальных циклов в графе

Функция count\_cycles

- adj\_matrix: Матрица смежности графа.

- V: Количество вершин графа.

1. Инициализировать множество all\_cycles для хранения уникальных циклов.

2. Для каждой вершины start от 0 до V-1:

- Создать стек stack, содержащий кортеж (start, [start], [False] \* V), где:

- current: текущая вершина.

- path: список, представляющий путь от начальной вершины.

- visited: список булевых значений, отслеживающий посещение вершин.

3. Пока стек не пуст:

- Извлечь кортеж (current, path, visited) из стека.

- Установить visited[current] = True (пометить текущую вершину как посещённую).

- Для каждой вершины neighbor от 0 до V-1:

- Если существует ребро adj\_matrix[current][neighbor] == 1:

- Если neighbor == start и длина path больше 2:

- Создать кортеж cycle из отсортированного path.

- Добавить cycle в all\_cycles.

- Иначе, если neighbor не посещена (visited[neighbor] == False):

- Добавить в стек новый кортеж (neighbor, path + [neighbor], копия visited).

- После завершения обработки текущей вершины установить visited[current] = False (снять пометку о посещении).

4. Вернуть размер множества all\_cycles (количество уникальных циклов).

Граф G, в котором q(G) = p(G) − 1, называется древочисленным

Проверка древочисленности:  
Функция is\_drevocislen(graph):

1. Инициализируем переменную edge\_count = 0 для подсчёта количества рёбер.

2. Для каждой вершины u от 0 до V - 1:

2.1. Для каждой вершины v от u + 1 до V - 1:

- Если adj\_matrix[u][v] == 1 (есть ребро между u и v):

- Увеличиваем edge\_count на 1.

3. После завершения подсчёта рёбер:

- Если edge\_count == V - 1:

- Возвращаем True (граф древочисленный).

- Иначе:

- Возвращаем False (граф не древочисленный).

Подсчет циклов

- Создаем пустое множество all\_cycles для хранения уникальных циклов.

- Определяем V как размерность матрицы смежности (число вершин).

Для каждой вершины start\_vertex в диапазоне от 0 до V:

- Инициализируем стек stack с элементом (start\_vertex, [start\_vertex], [False] \* V)

(текущая вершина, путь до неё, массив посещённых вершин).

Пока стек не пуст:

- Извлекаем current\_vertex, path, visited из стека.

- Помечаем current\_vertex как посещённую: visited[current\_vertex] = True.

Для каждого соседа neighbor вершины current\_vertex:

- Если adj\_matrix[current\_vertex][neighbor] == 1 (есть ребро между вершинами):

- Если neighbor == start\_vertex и длина path > 2:

- Найден цикл (возврат в стартовую вершину).

- Сортируем путь path и добавляем его в all\_cycles.

- Иначе, если neighbor ещё не посещён:

- Кладём (neighbor, path + [neighbor], copy(visited)) в стек.

- Сбрасываем посещённость current\_vertex: visited[current\_vertex] = False.

- Возвращаем размер all\_cycles как количество уникальных циклов.

Проверка графа является ли оно деревом будем с помощью утверждения 7 из теоремы параграфа “Основные свойства свободных деревьев”. G — ациклический и субциклический, z(G) = 0 & z(G + x) = 1. В случае положительной проверки будем записывать в файл: «Граф является деревом», иначе «Граф не является дерево»  
Проверка на дерево:  
 - Если is\_sybsiclic и is\_asyclic:

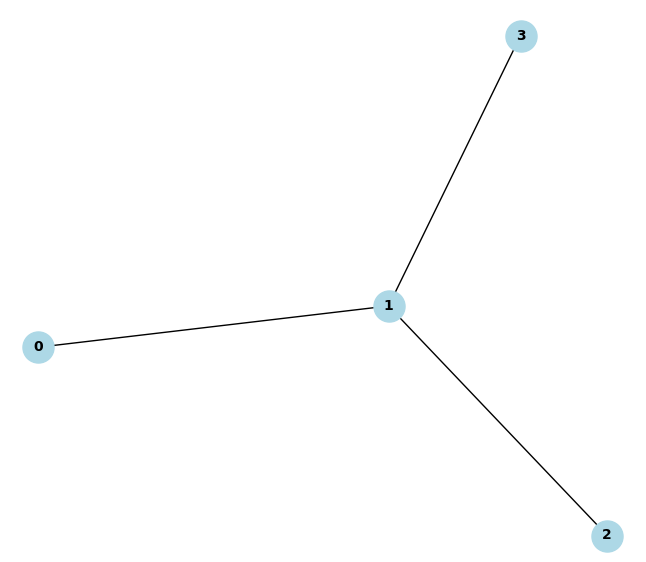
Вернуть True

Иначе:  
 Вернуть False

**Пример работы**

Рассмотрим графы для примеров проверки свойств

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 |



Граф является ацикличным.

Старт из вершины 0:

Стек: [(0, -1)].

Из 0 идём к 1 (ребро 0-1).

Из 1 идём к 2 (ребро 1-2), затем возвращаемся.

Из 1 идём к 3 (ребро 1-3), затем возвращаемся.

Нет повторного посещения уже пройденной вершины, нет цикла.

Старт из вершин 1, 2, 3 также не выявляет циклов.

Вывод: Граф ацикличен.

Граф является древочисленным (q = p - 1).

Количество вершин p=4.

Сумма всех элементов матрицы = 1+3+1+1=6. Делим на 2: q=3.

p−1=3.

Граф является субциклическим.

Пары несвязанных вершин: (0-2), (0-3), (2-3).

Для пары (0-2):

Добавляем ребро 0-2.

Проверяем наличие циклов: один цикл (0-1-2-0).

Количество циклов = 1, граф остаётся субциклическим.

Для пары (0-3):

Добавляем ребро 0-3.

Проверяем наличие циклов: один цикл (0-1-3-0).

Количество циклов = 1, граф остаётся субциклическим.

Для пары (2-3):

Добавляем ребро 2-3.

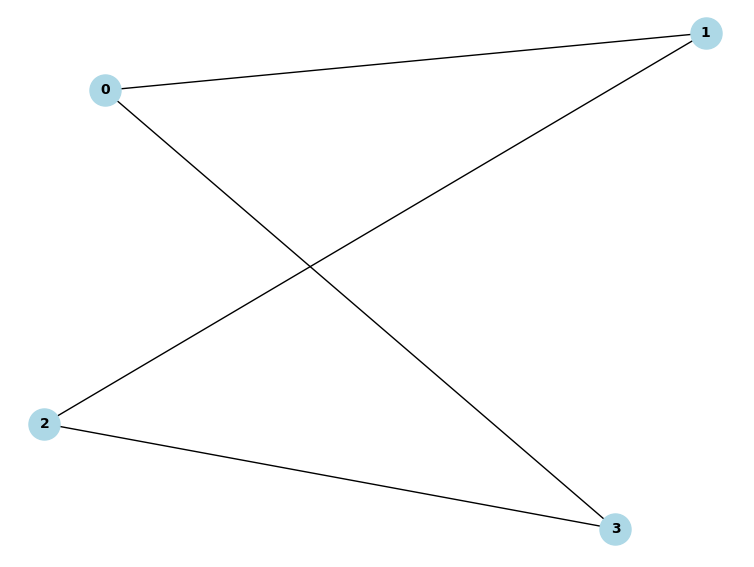
Проверяем наличие циклов: один цикл (1-2-3-1).

Количество циклов = 1, граф остаётся субциклическим.

Вывод: Граф является субциклическим.

Граф является деревом.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 |
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 1 | 0 | 1 | 0 |



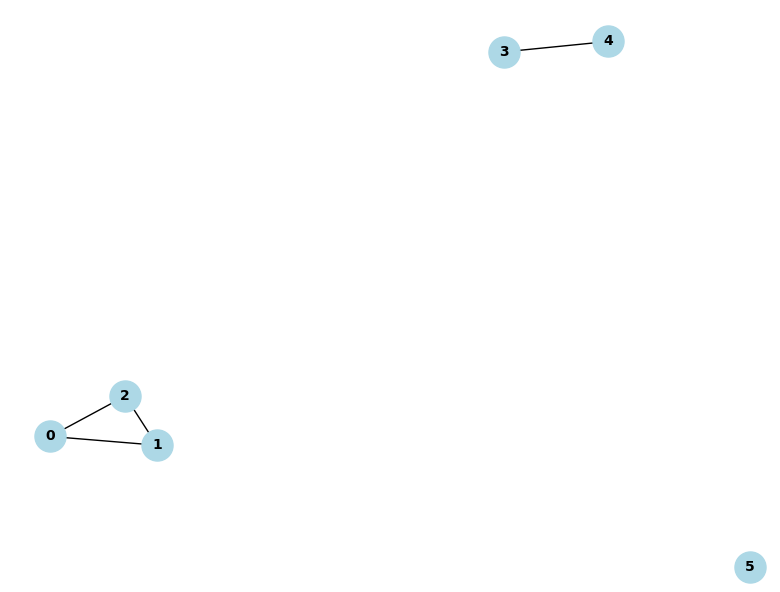
Граф содержит цикл: [1, 0, 3, 2, 1]

Граф не является древочисленным (q != p - 1).

Субцикличность нарушена при добавлении ребра (0, 2).

Граф не является деревом.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 2 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |



Граф содержит цикл: [1, 0, 2, 1]

Граф не является древочисленным (q != p - 1).

Граф является субциклическим.

Граф не является деревом.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | 1 | 2 |
| 0 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 |



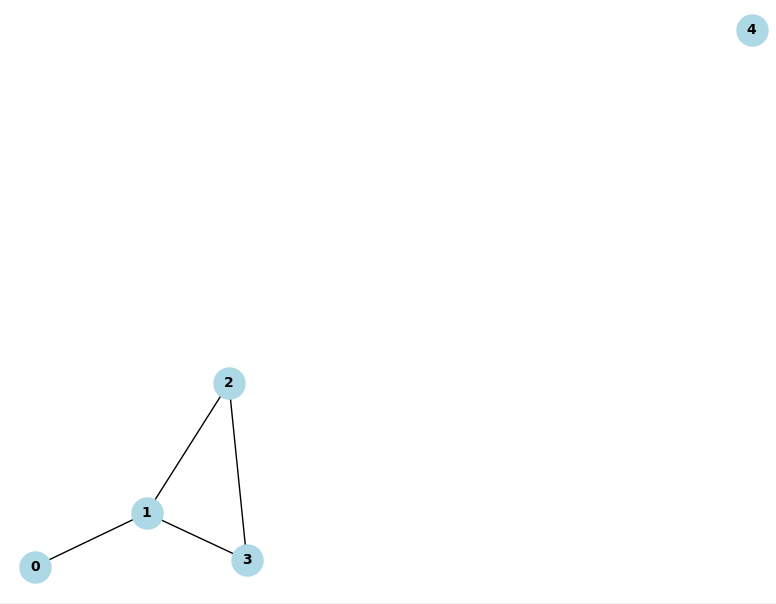
Граф является ацикличным.

Граф не является древочисленным (q != p - 1).

Субцикличность нарушена при добавлении ребра Любого

Граф не является деревом.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 |



Граф содержит цикл: [2, 1, 3, 2]

Граф является древочисленным (q = p - 1).

Субцикличность нарушена при добавлении ребра (0, 2).

Граф не является деревом.

**Сложность**

Ацикличность. Алгоритм проверки ацикличности основывается на поиске цикла через DFS. Алгоритм обхода графа с использованием DFS посещает каждую вершину и каждое ребро один раз. Для поиска цикла используется множество посещенных вершин и структура родителя для отслеживания предков. Это имеет сложность O(V+E).

Субцикличность. Алгоритм проверки ацикличности основывается на подсчете циклов через DFS. Основной цикл проходит по всем парам вершин, а для каждой пары вызывается подсчет циклов, дающий итоговую сложность O(V^2 (V + E)).

**Входные и выходные данные**

Входные данные. Квадратная матрица n×n, где n — количество вершин в графе. Каждая строка матрицы представляет связи (ребра) для одной вершины.

Выходные данные записываем в файл в виде

if граф ацикличный:

        f.write("Граф является ацикличным.")

else:

        f.write (f"Граф содержит цикл: {найденный цикл}")

if граф древочисленный:

        f.write ("Граф является древочисленным (q = p - 1).")

else:

        f.write ("Граф не является древочисленным (q != p - 1).")

if граф субцикличный:

        f.write ("Граф является субциклическим.")

else:

        f.write (f"Субцикличность нарушена при добавлении ребра {найденное ребро}.")

if граф дерево ():

        f.write ("Граф является деревом.")

else:

        f.write ("Граф не является деревом.")

**Область применимости**

Проверка графа на свойства и дальнейшая работа с ним особенно полезен в области сетей, алгоритмической оптимизации, системного проектирования и научных исследований.

**Представление графов в программе**

Для представления графа в программе я буду использовать матрицу смежности. Доступ к данным в матрице занимает O(1) что делает возможным прямую и быструю работу с каждой парой вершин. Кроме того, добавление или удаление ребра также выполняется за O(1), так как достаточно изменить один элемент матрицы.

**Вывод**

Данный код предоставляет универсальный инструментарий для анализа свойств графов, включая проверку их ацикличности, связности, субцикличности, древовидности и других характеристик. Он эффективно реализует алгоритмы на основе матрицы смежности, что делает его подходящим для решения задач в широком спектре областей, таких как теория графов, сетевой анализ, оптимизация маршрутов, проектирование систем и научные исследования.