Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 5

Выполнил студент группы.....КС-38.....(Нергарян Геворг Гарегинович)
Ссылка на репозиторий: .....(https://github.com/MUCTR-IKT-CPP/Nergaryan_KC-38_Algos)

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович
.....Лобанов Алексей Владимирович
.....Крашенинников Роман Сергеевич

Дата сдачи: …….20.05.2023……..

Оглавление

Описание задачи. 3

Описание метода/модели. 4

Выполнение задачи. 6

Заключение. 11

Описание задачи.

1. Создайте взвешенный граф, состоящий из [10, 20, 50, 100] вершин.
    - Каждая вершина графа связана со случайным количеством вершин, минимум с [3, 4, 10, 20].
    - Веса ребер задаются случайным значением от 1 до 20.
    - Каждая вершина графа должна быть доступна, т.е. до каждой вершины графа должен обязательно существовать путь до каждой вершины, не обязательно прямой.
2. Выведите получившийся граф в виде матрицы смежности. Пример вывода данных:
3. Для каждого графа требуется провести серию из 5 - 10 тестов, в зависимости от времени затраченного на выполнение одного теста., необходимо:
    - **Вариант 1**. Найти кратчайшие пути между всеми вершинами графа и их длину с помощью алгоритма Дейкстры.
    - **Вариант 2**. Построить минимальное остовное дерево взвешенного связного неориентированного графа с помощью алгоритма Прима.
    - **Вариант 3**. Найти кратчайшие пути между всеми вершинами графа и их длину с помощью алгоритма Флойда — Уоршелла.
    - **Вариант 4**. Построить минимальное остовное дерево взвешенного связного неориентированного графа с помощью алгоритма Краскала.
4. В рамках каждого теста, необходимо замерить потребовавшееся время на выполнение задания из пункта 3 для каждого набора вершин. По окончанию всех тестов необходимо построить график используя полученные замеры времени, где на ось абсцисс (X) нанести N – количество вершин, а на ось ординат(Y) - значения затраченного времени.

Описание метода/модели.

Взвешенный граф - это граф, в котором каждому ребру присвоено числовое значение, называемое весом. Вес может представлять различные характеристики, такие как расстояние, стоимость, пропускная способность и т. д. Взвешенный граф используется для моделирования ситуаций, где ребра имеют разные степени важности или стоимости.

Вершины графа - это основные элементы графа, которые представляют собой отдельные объекты или сущности. Вершины обычно обозначаются точками или кругами и могут быть связаны друг с другом ребрами. Взвешенные графы могут быть как ориентированными, где ребра имеют направление, так и неориентированными, где ребра не имеют направления.

Вес ребра - это числовое значение, присвоенное каждому ребру в графе, чтобы указать его стоимость, расстояние или другую характеристику. Вес может быть положительным или отрицательным числом, или даже нулем, в зависимости от конкретной ситуации. Например, в графе, моделирующем систему дорог, вес ребра может представлять расстояние между двумя городами или время пути.

Матрица смежности - это способ представления графа в виде квадратной матрицы, где строки и столбцы соответствуют вершинам графа. Значение в ячейке (i, j) матрицы указывает на наличие ребра между вершинами i и j. В случае взвешенного графа, значение в ячейке матрицы представляет вес ребра между соответствующими вершинами. Если между вершинами нет ребра, обычно используется специальное значение (например, бесконечность) или ноль.

Преимуществом матрицы смежности является простота проверки наличия ребра между двумя вершинами и быстрый доступ к весу ребра. Однако, она требует больше памяти для хранения информации о графе, особенно для больших и разреженных графов. Кроме того, при изменении структуры графа, например, при добавлении или удалении вершин или ребер, матрица смежности может потребовать обновления всей матрицы, что может быть затратным с точки зрения времени и памяти.

Алгоритм Прима - это алгоритм для построения минимального остовного дерева взвешенного связного неориентированного графа. Остовное дерево - это подграф исходного графа, который содержит все вершины исходного графа и является деревом, то есть не содержит циклов.

Алгоритм Прима начинает с выбора произвольной вершины в качестве начальной и помечает ее как посещенную. Затем, на каждой итерации, выбирается ребро минимального веса, которое соединяет посещенную вершину с непосещенной вершиной. Это ребро добавляется в остовное дерево, и соответствующая вершина помечается как посещенная.

На каждой итерации алгоритма Прима, мы выбираем ребро минимального веса, которое связывает посещенные и непосещенные вершины. Таким образом, остовное дерево постепенно строится, пока все вершины не будут посещены.

Алгоритм Прима может быть реализован с использованием минимальной кучи (min-heap), которая позволяет эффективно выбирать ребро минимального веса на каждой итерации. Минимальная куча поддерживает свойство того, что наименьший элемент всегда находится на вершине кучи.

После завершения алгоритма Прима, мы получаем минимальное остовное дерево, которое содержит все вершины исходного графа и имеет минимальную сумму весов ребер.

Алгоритм Прима эффективен и имеет временную сложность O(E log V), где E - количество ребер, а V - количество вершин в графе. Он гарантирует построение минимального остовного

дерева и может быть использован, например, для оптимизации сетевых или транспортных систем, где требуется связать все узлы с минимальной стоимостью.

Выполнение задачи.

Код на Языке Python

```
import random
import networkx as nx
from collections import deque
import timeit
import matplotlib.pyplot as plt
import scipy


# Класс для создания различных графов
class GraphGen:
    # Конструктор класса
    def __init__(self, min_nodes, max_nodes, min_edges, max_edges, max_edges_per_node,
min_edges_per_node,
                 is_directed=False, max_incoming_edges=None, max_outgoing_edges=None):
        self.min_nodes = min_nodes
        self.max_nodes = max_nodes
        self.min_edges = min_edges
        self.max_edges = max_edges
        self.max_edges_per_node = max_edges_per_node
        self.is_directed = is_directed
        self.max_incoming_edges = max_incoming_edges
        self.max_outgoing_edges = max_outgoing_edges
        self.min_edges_per_node = min_edges_per_node

    # Метод на получение матрицы смежности графа selected_graph
    def adjacency_matrix(self, selected_graph):
        return nx.adjacency_matrix(selected_graph).todense()

    # Метод на генерацию графа (рандомного)
    def generate_graph(self):
        # генерируем случайное число узлов
        n = random.randint(self.min_nodes, self.max_nodes)
        # генерируем случайное число ребер
        m = random.randint(self.min_edges, min(self.max_edges, n * (n - 1) // 2))
        # создаем граф, в зависимости от параметра будет он ориентированный или нет
        new_graph = nx.Graph() if not self.is_directed else nx.DiGraph()
        # создаем список узлов
        nodes = list(range(n))
        # добавляем узлы в граф
        new_graph.add_nodes_from(nodes)
        edges = []
        max_degrees = [0] * n
        # проходим ао всем узлам в графе
        for i in range(n):
            # случайное количество ребер, которые будут присоединены к узлу i
            k = random.randint(self.min_edges_per_node, min(self.max_edges_per_node, n
- 1))
            # если граф не ориентированный, учитываем уже имеющиеся ребра
            if not self.is_directed:
                k = min(k, n - 1 - len(new_graph.adj[i]))
            # выбираем случайные узлы для присоединения ребер к узлу i
            j_candidates = set(nodes) - set(new_graph.adj[i]) - set([i])
            j_candidates = random.sample(list(j_candidates), min(len(j_candidates), k))
            # проходим по всем узлам j, которые были выбраны для присоединения ребер к
узлу
            for j in j_candidates:
                # если узел j еще не достиг максимального количества входящих и
исходящих ребер
                if (self.max_incoming_edges is None or new_graph.in_degree(j) <
self.max_incoming_edges) and (
                        self.max_outgoing_edges is None or new_graph.out_degree(i) <
self.max_outgoing_edges):
```

```
                    # добавляем ребро (i, j, weight)
                    edges.append((i, j, random.randint(1, 20)))
                    # увеличиваем степени вершин i и j
                    max_degrees[i] += 1
                    max_degrees[j] += 1
        # перемешиваем список ребер и добавляем первые m ребер в граф
        random.shuffle(edges)
        new_graph.add_weighted_edges_from(edges[:m])
        # находим максимальные входящую и исходящую степень вершин (только для
ориентированных графов)
        max_in_degrees = max_out_degrees = 0
        for i in range(n):
            if self.is_directed:
                max_in_degrees = max(max_in_degrees, new_graph.in_degree(i))
                max_out_degrees = max(max_out_degrees, new_graph.out_degree(i))
            else:
                max_degrees[i] = new_graph.degree(i)

        return new_graph, {
            'максимальное количество узлов в графе': n,
            'минимальное количество ребер в графе': self.min_edges,
            'максимальное количество ребер в графе': self.max_edges,
            'максимальное количество ребер к каждому узлу': self.max_edges_per_node,
            'Направленый граф': self.is_directed,
            # максимальное количество входящих ребер, которые могут быть присоединены к
каждому узлу (только для ориентированных графов).
            'максимальное количество входящих ребер, присоединяемых к узлам':
self.max_incoming_edges,
            # максимальное количество исходящих ребер, которые могут быть присоединены
к каждому узлу (только для ориентированных графов).
            'максимальное количество исходящих ребер, присоединяемых к узлам':
self.max_outgoing_edges,
            'максимальная степень вершины в графе': max(max_degrees),
            'максимальная входящая степень вершины': max_in_degrees,
            'максимальная исходящая степень вершины': max_out_degrees,
            'минимальная степень вершины в графе': min(max_degrees),

        }

    # Метод на генерацию случайных графов в размере num_graphs
    # каждый последующий граф  с возрастающим количеством вершин и ребер
    def generate_graphs(self, num_graphs):
        graphs = []
        for i in range(num_graphs):
            graph, metadata = self.generate_graph()
            graphs.append((graph, metadata))
        return graphs


# Функция на вывод информации графика (вывод матрицы смежности,
# вывод минимального остового дерева методом Прима и замер времени)
def primAlg(graphs_list, graph_gen, graphs_times):
    for i, (graph, metadata) in enumerate(graphs_list):
        start = random.choice(list(graph.nodes()))  # выбираем рандомно начальную
вершину
        end = random.choice(list(graph.nodes()))  # выбираем рандомно конечную вершину

print('=========================================================================')
        print(f"Граф {i + 1}:")
        print("-------------------------------------------------")
        print("Матрица смежности:")
        print(graph_gen.adjacency_matrix(graph))
        print("-------------------------------------------------")
        print("Алгоритм Прима (минимальное остовое дерево):")
        prim_start = timeit.default_timer()

        # Алгоритм Прима
```

```
        #
        # Инициализируйте алгоритм, выбрав исходную вершину
        # Найдите ребро с минимальным весом, подключенное к исходному узлу и другому
узлу, и добавьте его в дерево
        # Продолжайте повторять этот процесс, пока мы не найдем минимальное связующее
дерево

        # Передаем матрицу смежности в переменную
        matr = graph_gen.adjacency_matrix(graph)
        selected_node = [0] * len(matr)
        no_edge = 0
        selected_node[0] = True
        print("Edge : Weight\n")
        # Пока длина остового дерева не равна размеру графа, выполняем цикл
        while (no_edge < len(matr) - 1):
            a = 0
            b = 0
            minimum = 999
            for m in range(len(matr)):
                if selected_node[m]:
                    for n in range(len(matr)):
                        if ((not selected_node[n]) and matr[m][n]):
                            # Если вершина не была выбрана, добавляем ее
                            # и меняем минимальное значение
                            if minimum > matr[m][n]:
                                minimum = matr[m][n]
                                a = m
                                b = n
            print(str(a) + "-" + str(b) + ":" + str(matr[a][b]))
            selected_node[b] = True
            no_edge += 1

        prim_end = timeit.default_timer()
        prim_time = prim_end - prim_start
        graphs_times.append(prim_time)
        print('ВРЕМЯ ВЫПОЛНЕНИЯ АЛГОРИТМА: ', prim_time)
        print("-------------------------------------------------")
        print("Доп данные по графу:")
        print(metadata)
        print("=================================================================\n")


# Функция на нахождение среднего значения
def findAvg(list_of_times):
    summ = 0
    for elem in list_of_times:
        summ += elem
    return summ / len(list_of_times)


# Списки для записи времени (для построения графика)
min_prim_times = []
max_prim_times = []
avg_prim_times = []

# Создаем генераторы для графов из [10,20,50,100] вершин
generator_10nodes = GraphGen(min_nodes=10, max_nodes=10, min_edges=40, max_edges=45,
max_edges_per_node=9,
                             min_edges_per_node=3)
generator_20nodes = GraphGen(min_nodes=20, max_nodes=20, min_edges=185, max_edges=190,
max_edges_per_node=9,
                             min_edges_per_node=4)
generator_50nodes = GraphGen(min_nodes=50, max_nodes=50, min_edges=1220,
max_edges=1225, max_edges_per_node=20,
                             min_edges_per_node=10)
generator_100nodes = GraphGen(min_nodes=100, max_nodes=100, min_edges=4930,
max_edges=4950, max_edges_per_node=30,
```

```
                                min_edges_per_node=20)

# Генерируем 5 рандомных графов с разным кол-вом вершин
graphs_10 = generator_10nodes.generate_graphs(5)
graphs_20 = generator_20nodes.generate_graphs(5)
graphs_50 = generator_50nodes.generate_graphs(5)
graphs_100 = generator_100nodes.generate_graphs(5)

# Списки для хранения времени  с графов (алгоритма Прима)
prim_times_10 = []
prim_times_20 = []
prim_times_50 = []
prim_times_100 = []

# Производим работу алгоритма Прима с выводом данных
primAlg(graphs_10, generator_10nodes, prim_times_10)
primAlg(graphs_20, generator_20nodes, prim_times_20)
primAlg(graphs_50, generator_50nodes, prim_times_50)
primAlg(graphs_100, generator_100nodes, prim_times_100)

# Добавляем значения лучших, худших и средних значений по времени для графов
min_prim_times.extend([min(prim_times_10), min(prim_times_20), min(prim_times_50),
min(prim_times_100)])
max_prim_times.extend([max(prim_times_10), max(prim_times_20), max(prim_times_50),
max(prim_times_100)])
avg_prim_times.extend([findAvg(prim_times_10), findAvg(prim_times_20),
findAvg(prim_times_50), findAvg(prim_times_100)])

# Построение графика для сравнения алгоритма прима для графов с разным кол-вом вершин
plt.title('График на сравнение резльтатов нахождния остового дерева (Прима)')
plt.plot([10, 20, 50, 100], min_prim_times, label="График лучшего времени")
plt.plot([10, 20, 50, 100], max_prim_times, label="График худшего времени")
plt.plot([10, 20, 50, 100], avg_prim_times, label="График среднего времени")
plt.xlabel("Кол-во узлов")
plt.ylabel("Время, сек.")
plt.legend()
plt.show()
```

В результате выполнения программы были получены матрицы смежности для каждого случая [10, 20, 50, 100] вершин.

```
Матрица смежности:
[[ 0 19 13 19  0  0 13 17 10  6]
 [19  0  2  7 14 13  0  3  0  0]
 [13  2  0  7 20  4  8 16  5  7]
 [19  7  7  0 13 16  9  0  0  9]
 [ 0 14 20 13  0  0 14  8  2 13]
 [ 0 13  4 16  0  0  1 15  0  5]
 [13  0  8  9 14  1  0 16  0 10]
 [17  3 16  0  8 15 16  0  0 20]
 [10  0  5  0  2  0  0  0  0 19]
 [ 6  0  7  9 13  5 10 20 19  0]]
```

```
[[ 0 10  9  6  5  0 12  0  0  0  0  0 15 10  0  0  0  0 20  0]
 [10  0  0  0  0 11  0 11  0  0  0  0  0 17  0  3 20  2 15  3]
 [ 9  0  0  0  0  3  0  0  0  0  3 17  4  0 18  0  0 20  7  0]
 [ 6  0  0  0 12  0  0 12 20 19  8  0  0  0  0  0  0  1 17  0]
 [ 5  0  0 12  0  0  1  0  0  6 13 19  0  9 14  0  0  0 19  0]
 [ 0 11  3  0  0  0 11  0  0  4  0  0  0  8  2 15  3  0  2 10]
 [12  0  0  0  1 11  0 10  0  4 20 10 18  0  7  0  5  2  0  6]
 [ 0 11  0 12  0  0 10  0  0  0  0  2  0 16  3  0  9  0 10 14]
 [ 0  0  0 20  0  0  0  0  0 17 10  0 17 12  0  0 19  0 12  5]
 [ 0  0  0 19  6  4  4  0 17  0  5  0 20  4 13 16 14  0 10  0]
 [ 0  0  3  8 13  0 20  0 10  5  0  9  0  0 15 14 15 13 20 18]
 [ 0  0 17  0 19  0 10  2  0  0  9  0  0 12  0  0  7 14  0 11]
 [15  0  4  0  0  0 18  0 17 20  0  0  0  0  6 14  0 12  3  0]
 [10 17  0  0  9  8  0 16 12  4  0 12  0  0  0  0  1 15  7  0]
 [ 0  0 18  0 14  2  7  3  0 13 15  0  6  0  0 16  4  9 17  0]
 [ 0  3  0  0  0 15  0  0  0 16 14  0 14  0 16  0 16 17  0  0]
 [ 0 20  0  0  0  3  5  9 19 14 15  7  0  1  4 16  0  0  0  9]
 [ 0  2 20  1  0  0  2  0  0  0 13 14 12 15  9 17  0  0  5  8]
 [20 15  7 17 19  2  0 10 12 10 20  0  3  7 17  0  0  5  0  0]
 [ 0  3  0  0  0 10  6 14  5  0 18 11  0  0  0  0  9  8  0  0]]
```

Для каждого графа строится минимальное остовное дерево взвешенного связного неориентированного графа с помощью алгоритма Прима и замеряется время для этого.

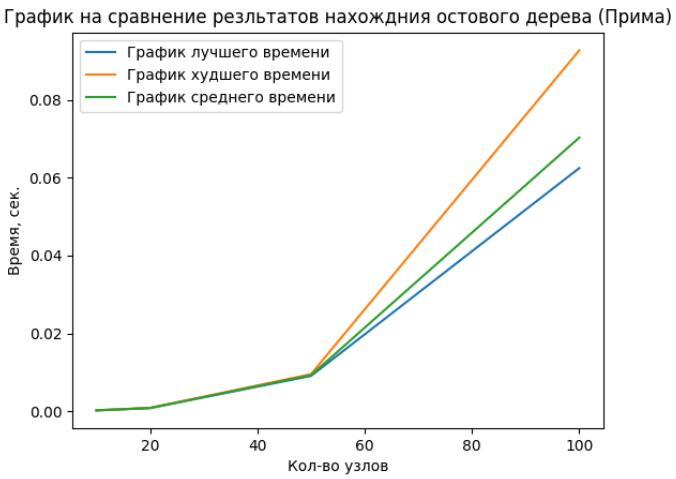
```
Алгоритм Прима (минимальное остовое дерево):
Edge : Weight

0-9:6
9-5:5
5-6:1
5-2:4
2-1:2
1-7:3
2-8:5
8-4:2
1-3:7
ВРЕМЯ ВЫПОЛНЕНИЯ АЛГОРИТМА:  0.0003055000015592668
```

```
Алгоритм Прима (минимальное остовое дерево):
Edge : Weight

0-4:5
4-6:1
6-17:2
17-3:1
17-1:2
1-15:3
1-19:3
6-9:4
9-5:4
5-14:2
5-18:2
5-2:3
2-10:3
5-16:3
16-13:1
14-7:3
7-11:2
18-12:3
19-8:5
ВРЕМЯ ВЫПОЛНЕНИЯ АЛГОРИТМА:  0.0008891999968909658
```

После всего этого, используя полученные данные строим график.



Заключение.

В результате выполнения исследования по построению минимального остовного дерева взвешенного связного неориентированного графа с помощью алгоритма Прима, мы получили следующие выводы.

Алгоритм Прима является эффективным методом для построения минимального остовного дерева взвешенного связного неориентированного графа. Он обеспечивает построение остовного дерева, которое содержит все вершины исходного графа и имеет минимальную сумму весов ребер.

Алгоритм Прима позволяет последовательно выбирать ребра минимального веса, связывающие посещенные и непосещенные вершины. Это позволяет строить остовное дерево шаг за шагом, добавляя на каждой итерации ребро с минимальным весом.

Важным аспектом алгоритма Прима является использование минимальной кучи (min-heap) для эффективного выбора ребра минимального веса. Минимальная куча обеспечивает быстрый доступ к наименьшему элементу и позволяет оптимизировать время выполнения алгоритма.

Алгоритм Прима имеет временную сложность O(E log V), где E - количество ребер, а V - количество вершин в графе. Это делает его эффективным даже для больших графов.

Использование минимального остовного дерева может быть полезным в различных областях, таких как сетевое планирование, транспортные системы, оптимизация маршрутов и других ситуациях, где требуется связывание всех вершин с минимальными затратами или стоимостью.

В заключение, алгоритм Прима предоставляет эффективный и надежный подход для построения минимального остовного дерева взвешенного связного неориентированного графа. Его применение может привести к оптимизации стоимости, времени или других ресурсов, что делает его полезным инструментом в различных сферах.