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 № 4

Выполнил студент группы ....КС-38.. Прилепский Артем Сергеевич
Ссылка на репозиторий: ....github.com/news1d/Algorithms_and_structures

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

Дата сдачи: 24.03.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо реализовать генератор случайных графов, генератор должен содержать следующие параметры:

- Максимальное/Минимальное количество генерируемых вершин
- Максимальное/Минимальное количество генерируемых ребер
- Максимальное количество ребер связанных с одной вершины
- Генерируется ли направленный граф
- Максимальное количество входящих и выходящих ребер

Сгенерированный граф должен быть описан в рамках одного класса (этот класс не должен заниматься генерацией), и должен обладать обязательно следующими методами:

- Выдача матрицы смежности
- Выдача матрицы инцидентности
- Выдача список смежности
- Выдача списка ребер

В качестве проверки работоспособности, требуется сгенерировать 10 графов с возрастающим количеством вершин и ребер (количество выбирать в зависимости от сложности расчета для вашего отдельно взятого ПК). На каждом из сгенерированных графов требуется выполнить поиск кратчайшего пути или подтвердить его отсутствие из точки А в точку Б, выбирающиеся случайным образом заранее, поиском в ширину и поиском в глубину, замерев время требуемое на выполнение операции. Результаты замеров наложить на график и проанализировать эффективность применения обоих методов к этой задаче.

Описание метода/модели.

Граф — математическая абстракция реальной системы любой природы, объекты которой обладают парными связями. Граф как математический объект есть совокупность двух множеств — множества самих объектов, называемого множеством вершин, и множества их парных связей, называемого множеством рёбер.

Поиск в глубину (англ. *Depth-first search, **DFS***) — один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра.

Поиск в ширину (англ. *breadth-first search*, **BFS**) — один из методов обхода графа. Пусть задан граф и выделена исходная вершина. Алгоритм поиска в ширину систематически обходит все ребра для «открытия» всех вершин, достижимых из, вычисляя при этом расстояние (минимальное количество рёбер) от до каждой достижимой из вершины. Алгоритм работает как для ориентированных, так и для неориентированных графов.

Выполнение задачи.

Для реализации программы был выбран язык программирования Python. Граф реализован в виде класса с методами получения матрицы смежности, матрицы инцидентности, списка смежности и списка ребер.

1) В конструкторе класса получаем сгенерированный граф.
2) Функция получения матрицы смежности.
3) Функция получения матрицы инцидентности.
4) Функция получения списка смежности графа
5) Функция получения списка всех ребер графа
6) Функция поиска кратчайшего пути с помощью алгоритма поиска в ширину.
7) Функция поиска кратчайшего пути с помощью алгоритма поиска в глубину.

Код класса:

```
class Graph:
    def __init__(self, num_vertices, max_edges, max_edges_per_vertex, is_directed,
max_in_out_edges):
        self.graph = generate_graph(num_vertices=num_vertices, max_edges=max_edges,
                                    max_edges_per_vertex=max_edges_per_vertex,
is_directed=is_directed,
                                    max_in_out_edges=max_in_out_edges)

    # возвращаем матрицу смежности графа
    def adjacency_matrix(self):
        n = len(self.graph.nodes())
        # создаем пустую матрицу размера n x n, заполненную нулями
        matrix = np.zeros((n, n))
        # проходим по списку ребер и устанавливаем соответствующие значения элементов
матрицы в 1
        for v1, v2 in self.graph.edges():
            matrix[v1][v2] = 1
            if not self.graph.is_directed():
                matrix[v2][v1] = 1
        return matrix

    # возвращаем матрицу инцидентности графа
    def incidence_matrix(self):
    # Получаем список всех ребер
    edges = list(self.graph.edges())
    # Создаем матрицу
    num_edges = len(edges)
    num_vertices = self.graph.number_of_nodes()
    inc_matrix = np.zeros((num_vertices, num_edges))
    # Заполняем матрицу инцидентности
    if self.graph.is_directed():
        for i, (u, v) in enumerate(edges):
            inc_matrix[u][i] = 1
            inc_matrix[v][i] = -1
    else:
        for i, (u, v) in enumerate(edges):
```

```
            inc_matrix[u][i] = 1
            inc_matrix[v][i] = 1
    return inc_matrix

    # возвращаем список смежности графа
    # ключи - вершины графа, а значения - списки смежных вершин
    def adjacency_list(self):
        # Создаем пустой словарь, который будет использоваться для хранения списка 
смежности
        adj_list = {}
        for u, v in self.graph.edges():
            # Проверяем, есть ли вершина u или v в словаре, и если ее нет, то добавляем 
эту вершину в словарь со значением по умолчанию - пустым списком []
            if u not in adj_list:
                adj_list[u] = []
            if v not in adj_list:
                adj_list[v] = []
            # Добавляем вершину v в список смежности для вершины u
            adj_list[u].append(v)
            # Если граф неориентированный, то добавляем вершину u в список смежности 
для вершины v
            if not self.graph.is_directed():
                adj_list[v].append(u)
        return adj_list

    # возвращаеv список всех ребер графа
    def edge_list(self):
        edges = []
        # Проходимся по всем ребрам графа и добавляем их в список, u и v - вершины, 
которые соединяет это ребро
        for u, v in self.graph.edges():
            edges.append((u, v))
        return edges

    # поиск кратчайшего пути в графе с помощью алгоритма поиска в ширину
    def shortest_path_bfs(self, start_node, end_node):
        # Добавляем стартовую вершину в очередь и отмечаем ее как посещенную
        queue = [(start_node, [start_node])]
        visited = {start_node}

        # Если начальная вершина и конечная вершина совпадают, то возвращаем единичный 
путь
        if start_node == end_node:
            return [start_node]

        # Ищем путь от start до end, обрабатывая каждую вершину в очереди
        while queue:
            node, path = queue.pop(0)
            neighbors = self.graph[node]
            for neighbor in neighbors:
                if neighbor not in visited:
                    if neighbor == end_node:
                        return path + [neighbor]
                    else:
                        queue.append((neighbor, path + [neighbor]))
                        visited.add(neighbor)

        # Если мы дошли до конца очереди и не нашли конечную вершину, то путь не 
существует
        return None

    # поиск кратчайшего пути в графе с помощью алгоритма поиска в глубину
    def shortest_path_dfs(self, start_node, end_node, path=None, shortest=None):
        # Если путь не был передан как аргумент, инициализируем его пустым списком
        if path is None:
            path = []
```

```
        # Добавляем начальную вершину в путь
        path = path + [start_node]

        # Если начальная вершина и конечная вершина совпадают, то мы нашли искомый путь
и возвращаем его
        if start_node == end_node:
            return path

        # Для каждой вершины, смежной с начальной, выполняем следующие действия
        for node in self.graph[start_node]:
            # Если данная вершина не принадлежит текущему пути, то продолжаем поиск
            if node not in path:
                # Если кратчайший путь ещё не найден или текущий путь короче ранее
найденного, то запускаем
                # рекурсивный поиск пути из данной вершины
                if shortest is None or len(path) < len(shortest):
                    newpath = self.shortest_path_dfs(node, end_node, path, shortest)
                    # Если путь найден, то присваиваем его кратчайшему пути
                    if newpath is not None:
                        shortest = newpath

        # Возвращаем кратчайший путь, если он найден, иначе - None
        return shortest
```

Генерация графов производится отдельной функцией.

Код функции:

```
def generate_graph(num_vertices, max_edges, max_edges_per_vertex, is_directed,
max_in_out_edges):
    # создаем граф
    G = nx.DiGraph() if is_directed else nx.Graph()
    # добавляем вершины
    G.add_nodes_from(range(num_vertices))
    # создаем случайное количество ребер
    num_edges = random.randint(0, max_edges)
    # добавляем случайные ребра
    while G.number_of_edges() < num_edges:
        # выбираем случайную пару вершин
        v1, v2 = random.sample(list(G.nodes()), 2)
        # проверяем, что ребра еще нет и максимальное количество ребер у вершины не
превышено
        if G.number_of_edges(v1) < max_edges_per_vertex and G.number_of_edges(v2) <
max_edges_per_vertex and not G.has_edge(v1, v2):
            # добавляем ребро
            G.add_edge(v1, v2)
            if not is_directed and random.choice([True, False]):
                G.add_edge(v2, v1)
    # создаем случайные количество входящих и выходящих ребер
    if is_directed:
        for v in G.nodes():
            num_in_edges = random.randint(0, max_in_out_edges)
            num_out_edges = random.randint(0, max_in_out_edges)
            in_edges = [e for e in G.in_edges(v)]
            out_edges = [e for e in G.out_edges(v)]
            while len(in_edges) < num_in_edges:
                # выбираем случайную вершину, не являющуюся текущей
                u = random.choice([u for u in G.nodes() if u != v])
                # добавляем ребро
                G.add_edge(u, v)
                in_edges.append((u, v))
            while len(out_edges) < num_out_edges:
                # выбираем случайную вершину, не являющуюся текущей
                u = random.choice([u for u in G.nodes() if u != v])
                # добавляем ребро
```

```
            G.add_edge(v, u)
            out_edges.append((v, u))
    return G
```

Рассмотрим работу программы. Для теста возьмем направленный граф из 7 вершин.

```
Матрица смежности графа:
[0 1 0 0 0 0 1]
[1 0 1 1 0 0 1]
[0 0 0 0 1 1 1]
[1 1 0 0 1 1 1]
[0 0 1 0 0 1 0]
[1 0 1 0 1 0 0]
[0 0 0 1 1 0 0]

Матрица инцидентности графа:
[1 1 0 0 -1 0 0 0 0 0 -1 0 0 0 0 0 -1 0 0 0 0]
[0 -1 1 1 1 1 0 0 0 -1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 -1 1 1 1 0 0 0 0 0 -1 0 0 0 -1 0 0]
[0 0 -1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 -1 0]
[0 0 0 0 0 0 -1 0 0 0 0 -1 0 0 1 1 0 -1 0 0 -1]
[0 0 0 0 0 0 0 -1 0 0 0 0 -1 0 0 -1 1 1 1 0 0]
[-1 0 0 -1 0 0 0 0 -1 0 0 0 0 -1 0 0 0 0 0 1 1]
```

Матрица смежности и матрица инцидентности графа

Список смежности графа реализован таким образом: ключи – вершина графа, а значения – списки смежных вершин.

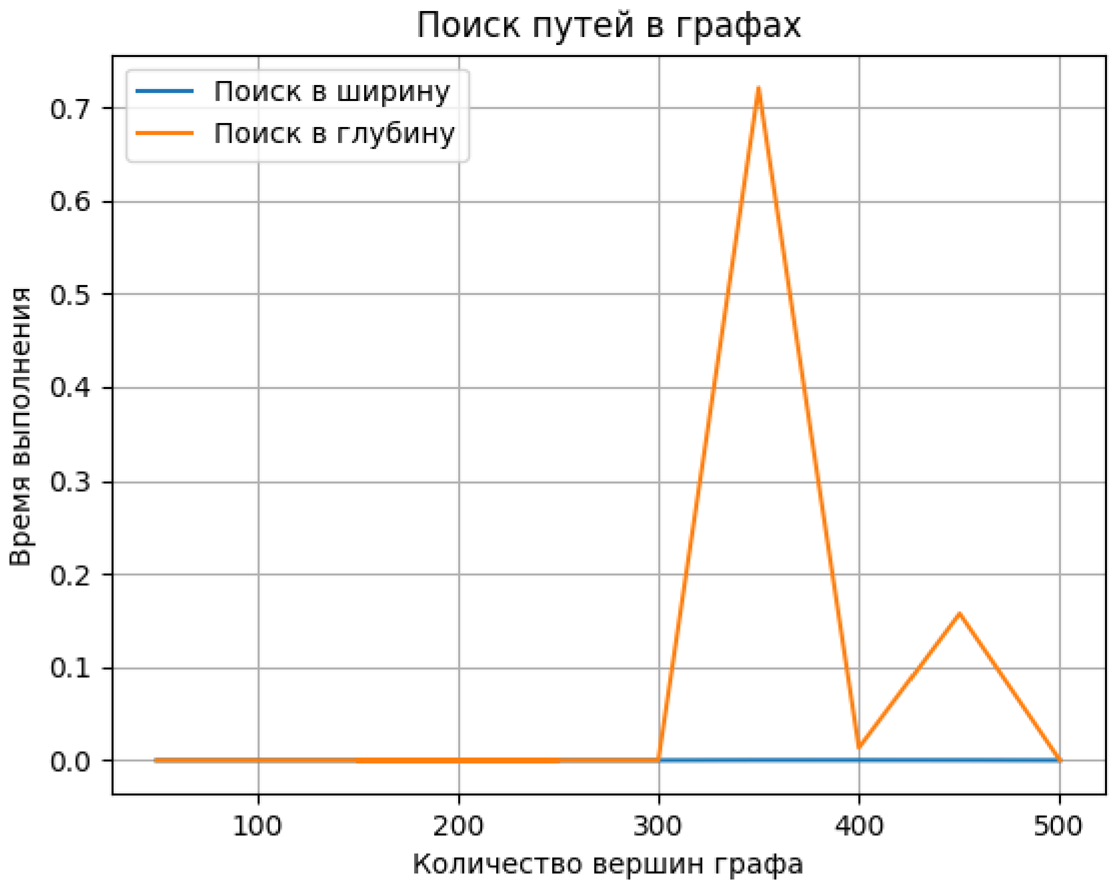
```
Список смежности графа:
{0: [6, 1], 6: [3, 4], 1: [3, 6, 0, 2], 3: [1, 0, 4, 5, 6], 2: [4, 5, 6], 4: [2, 5], 5: [0, 4, 2]}

Список всех ребер графа:
[(0, 6), (0, 1), (1, 3), (1, 6), (1, 0), (1, 2), (2, 4), (2, 5), (2, 6), (3, 1), (3, 0), (3, 4), (3, 5), (3, 6), (4, 2), (4, 5), (5, 0), (5, 4), (5, 2), (6, 3), (6, 4)]
```

Список смежности и список всех ребер графа

```
Тест №7:
Поиск в ширину:
Кратчайший путь: 6
Время выполнения: 0.00019
Поиск в глубину:
Кратчайший путь: 6
Время выполнения: 0.72064
```

Рисунок 1Пример работы программы



Результаты замеров

Заключение.

По графику видно, что метод поиска в глубину порой работает намного дольше, чем метод поиска в ширину. Поиск в глубину задействует рекурсию и просматривает все возможные пути от каждой точки до каждой, а метод поиска в ширину обходит граф внешним образом, используя очередь вершин и список посещенных вершин.