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-33..... (Вагенлейтнер Никита Сергеевич)

Ссылка на репозиторий: .....(https://github.com/MUCTR-IKT-CPP/VagenlejtnerNS_33_alg.git)

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

Дата сдачи: ..... (26.02.2025)

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

Граф — это структура данных, состоящая из вершин (узлов) и рёбер (связей между узлами). Граф может быть ориентированным (направленные рёбра) или неориентированным, а также взвешенным (с весами рёбер) или невзвешенным.

Граф в Python часто представляют в виде:

- Матрицы смежности (2D-список, где graph[i][j] указывает наличие рёбер).
- Списка смежности (словарь, где ключ — вершина, значение — список её соседей).

Выполнение задачи.

Для реализации и выполнения данного алгоритма был задействован язык Python. Были реализованы следующие программные блоки: алгоритмический блок, тестовый блок, аналитический блок полученных результатов работы алгоритма.

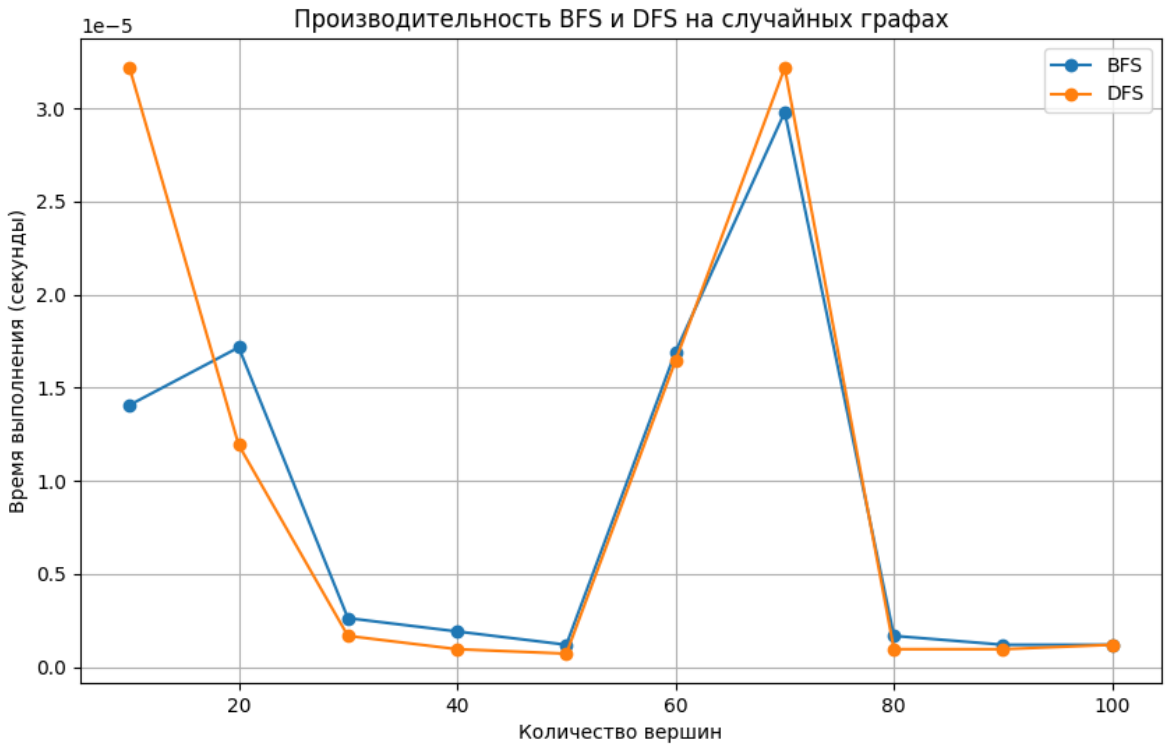
Блок кода отвечающий за реализацию графа и его генератор:

```
class Graph:
    def __init__(self, vertices: int, edges: List[Tuple[int, int]], directed: bool = False):
        self.vertices = vertices
        self.edges = edges
        self.directed = directed
        # Генерация списка смежности
        self.adj_list = defaultdict(list)
        for edge in edges:
            u, v = edge
            self.adj_list[u].append(v)
            if not directed:
                self.adj_list[v].append(u)
    def adjacency_matrix(self) -> List[List[int]]:
        """Возвращает матрицу смежности"""
        matrix = [[0] * self.vertices for _ in range(self.vertices)]
        for u, v in self.edges:
            matrix[u][v] = 1
            if not self.directed:
                matrix[v][u] = 1
        return matrix
    def incidence_matrix(self) -> List[List[int]]:
        """Возвращает матрицу инцидентности"""
        num_edges = len(self.edges)
        matrix = [[0] * num_edges for _ in range(self.vertices)]
        for i, (u, v) in enumerate(self.edges):
            matrix[u][i] = 1
            if not self.directed:
                matrix[v][i] = 1
            else:
                matrix[v][i] = -1
        return matrix
    def adjacency_list(self) -> Dict[int, List[int]]:
        """Возвращает список смежности"""
        return dict(self.adj_list)
    def edge_list(self) -> List[Tuple[int, int]]:
        """Возвращает список всех рёбер"""
        return self.edges
# ---------- Генератор графов ----------
class RandomGraphGenerator:
    def __init__(self,
```

Блоки кода отвечающие за тестирование:

```
    def test_performance():
        generator = RandomGraphGenerator(
            min_vertices=10,
            max_vertices=50,
            min_edges=20,
            max_edges=100,
            max_edges_per_vertex=10,
            directed=False
        )
        vertices_range = range(10, 105, 10)
        bfs_times = []
        dfs_times = []
        for num_vertices in vertices_range:
            graph = generator.generate()
            start, end = random.randint(0, num_vertices - 1), random.randint(0,
num_vertices - 1)
            while start == end:  # Исключаем одинаковые вершины
                end = random.randint(0, num_vertices - 1)
            # Замер времени для BFS
            start_time = time.time()
            bfs_shortest_path(graph, start, end)
            bfs_time = time.time() - start_time
            bfs_times.append(bfs_time)
            # Замер времени для DFS
            start_time = time.time()
            dfs_shortest_path(graph, start, end)
            dfs_time = time.time() - start_time
            dfs_times.append(dfs_time)
```

Результаты работы программы выводится в виде графика зависимости и приведены ниже.



Заключение.

Алгоритмы **BFS (поиск в ширину)** и **DFS (поиск в глубину)** эффективно применяются для обхода графов.

- **BFS** использует очередь, посещает вершины слоями и подходит для нахождения кратчайшего пути в невзвешенных графах.
- **DFS** использует стек (или рекурсию), углубляясь в граф, что полезно для проверки связности, поиска циклов и топологической сортировки.

При тестировании важно проверять алгоритмы на различных типах графов (ориентированные, неориентированные, разреженные, плотные) и учитывать крайние случаи, например, изолированные вершины или циклы.