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 Тернолуцкий Виктор Александрович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/VATernolutski\_36

Приняли:

Крашенинников Роман Сергеевич

Дата сдачи: 17.03.2025

Оглавление

[Описание задачи. 2](#_Toc63548272)

[Описание метода/модели. 2](#_Toc63548273)

[Выполнение задачи. 2](#_Toc63548274)

[Заключение. 2](#_Toc63548275)

# 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.

**Поиск в глубину** (***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. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

**Поиск в ширину** (**BFS**) — один из методов обхода графа. Пусть задан граф G=(V,E) и выделена исходная вершинаs. Алгоритм поиска в ширину систематически обходит все ребра G для «открытия» всех вершин, достижимых из s, вычисляя при этом расстояние (минимальное количество рёбер) от s до каждой достижимой из sвершины. Алгоритм работает как для ориентированных, так и для неориентированных графов.

Поиск в ширину имеет такое название потому, что в процессе обхода мы идём вширь, то есть перед тем как приступить к поиску вершин на расстоянии k+1, выполняется обход вершин на расстоянии k.

# Выполнение задачи.

Для реализации алгоритма использовался язык программирования python.

Он превосходит Java и C++ по скорости, лаконичности и масштабируемости.

Код и описание:

import random

import time

import os

import csv

import matplotlib.pyplot as plt

**def** format\_time(t):

    if t < 1e-6:

        return "0s"

    elif t < 1e-3:

        return **f**"{t\*1e6**:.1f**}µs"

    elif t < 1:

        return **f**"{t\*1e3**:.4g**}ms"

    else:

        return **f**"{t**:.4g**}s"

**class** Graph:

**def** \_\_init\_\_(self, vertices, edges=None, directed=False, edge\_list=None):

        """

        Если edge\_list задан, то граф создаётся по нему (для демонстрации);

        иначе генерируются случайные ребра (при условии, что edges не None).

        """

        self.vertices = vertices

        self.directed = directed

        self.adj\_matrix = [[0 for \_ in range(vertices)] for \_ in range(vertices)]

        self.adj\_list = {i: [] for i in range(vertices)}

        self.edge\_list = []

        if edge\_list is not None:

            self.edge\_list = edge\_list

            self.edges = len(edge\_list)

            for (u, v) in edge\_list:

                if u != v and self.adj\_matrix[u][v] == 0:

                    self.adj\_matrix[u][v] = 1

                    self.adj\_list[u].append(v)

                    if not directed:

                        self.adj\_matrix[v][u] = 1

                        self.adj\_list[v].append(u)

        else:

            self.edges = edges if edges is not None else 0

            self.generate\_edges()

        self.generate\_inc\_matrix()

**def** generate\_edges(self):

        """Генерация случайных рёбер для графа."""

        edge\_count = 0

        random.seed(time.time\_ns())

        while edge\_count < self.edges:

            u = random.randint(0, self.vertices - 1)

            v = random.randint(0, self.vertices - 1)

            if u != v and self.adj\_matrix[u][v] == 0:

                self.adj\_matrix[u][v] = 1

                self.adj\_list[u].append(v)

                self.edge\_list.append((u, v))

                if not self.directed:

                    self.adj\_matrix[v][u] = 1

                    self.adj\_list[v].append(u)

                edge\_count += 1

**def** generate\_inc\_matrix(self):

        """Генерация матрицы инцидентности на основе списка рёбер."""

        num\_edges = len(self.edge\_list)

        self.inc\_matrix = [[0 for \_ in range(num\_edges)] for \_ in range(self.vertices)]

        for i, (u, v) in enumerate(self.edge\_list):

            self.inc\_matrix[u][i] = 1

            if not self.directed:

                self.inc\_matrix[v][i] = 1

**def** print\_adj\_matrix(self):

        print("Выдача матрицы смежности:")

        for row in self.adj\_matrix:

            print("[" + " ".join(str(x) for x in row) + "]")

**def** print\_inc\_matrix(self):

        print("Выдача матрицы инцидентности:")

        for row in self.inc\_matrix:

            print("[" + " ".join(str(x) for x in row) + "]")

**def** print\_adj\_list(self):

        print("Выдача списка смежности:")

*# Для вывода в том же порядке, что в примере, сортируем ключи по убыванию*

        for vertex in sorted(self.adj\_list.keys(), reverse=True):

*# Выводим соседей через пробел*

            neighbors = " ".join(str(n) for n in self.adj\_list[vertex])

            print(**f**"{vertex}: [{neighbors}]")

**def** print\_edge\_list(self):

        print("Выдача списка ребер:")

        for (u, v) in self.edge\_list:

            print(**f**"{u} - {v}")

**def** BFS(self, start, target):

        """Проверка наличия пути от start до target методом поиска в ширину (BFS)."""

        visited = [False] \* self.vertices

        queue = [start]

        while queue:

            node = queue.pop(0)

            if node == target:

                return True

            if not visited[node]:

                visited[node] = True

                queue.extend(self.adj\_list[node])

        return False

**def** DFS(self, start, target):

        """Проверка наличия пути от start до target методом поиска в глубину (DFS)."""

        visited = [False] \* self.vertices

**def** dfs\_recursive(node):

            if node == target:

                return True

            if visited[node]:

                return False

            visited[node] = True

            for neighbor in self.adj\_list[node]:

                if dfs\_recursive(neighbor):

                    return True

            return False

        return dfs\_recursive(start)

**def** shortest\_path\_bfs(self, start, target):

        """Нахождение кратчайшего пути от start до target с использованием BFS."""

        if start == target:

            return [start], True

        visited = [False] \* self.vertices

        prev = [-1] \* self.vertices

        queue = [start]

        visited[start] = True

        while queue:

            node = queue.pop(0)

            for neighbor in self.adj\_list[node]:

                if not visited[neighbor]:

                    visited[neighbor] = True

                    prev[neighbor] = node

                    queue.append(neighbor)

                    if neighbor == target:

                        path = []

                        at = target

                        while at != -1:

                            path.insert(0, at)

                            at = prev[at]

                        return path, True

        return None, False

**def** find\_path\_dfs(self, start, target):

        """Нахождение (любого) пути от start до target с использованием DFS."""

        visited = [False] \* self.vertices

        path = []

**def** dfs\_recursive(node):

            if visited[node]:

                return False

            visited[node] = True

            path.append(node)

            if node == target:

                return True

            for neighbor in self.adj\_list[node]:

                if dfs\_recursive(neighbor):

                    return True

            path.pop()

            return False

        if dfs\_recursive(start):

            return path, True

        return None, False

**def** demo\_graph\_results():

    print("Результаты расчётов:")

    demo\_edge\_list = [(2,3), (1,0), (2,1), (1,4), (2,0),

                      (3,4), (2,4), (3,1), (0,4), (3,0)]

    demo\_graph = Graph(5, directed=False, edge\_list=demo\_edge\_list)

    demo\_graph.print\_adj\_matrix()

    demo\_graph.print\_inc\_matrix()

    demo\_graph.print\_adj\_list()

    demo\_graph.print\_edge\_list()

    start, target = 0, 4

    bfs\_path, bfs\_found = demo\_graph.shortest\_path\_bfs(start, target)

    dfs\_path, dfs\_found = demo\_graph.find\_path\_dfs(start, target)

    if bfs\_found:

        print(**f**"Кратчайший путь (BFS) из {start} в {target}: " + "[" + " ".join(str(x) for x in bfs\_path) + "]")

    else:

        print(**f**"BFS: Пути из {start} в {target} не существует")

    if dfs\_found:

        print(**f**"Путь (DFS) из {start} в {target}: " + "[" + " ".join(str(x) for x in dfs\_path) + "]")

    else:

        print(**f**"DFS: Пути из {start} в {target} не существует")

    print(**f**"Длина пути (BFS): {len(bfs\_path) if bfs\_path else 0}")

    print(**f**"Длина пути (DFS): {len(dfs\_path) if dfs\_path else 0}")

**def** run\_performance\_test(test\_name, vertices\_list, edge\_multiplier, directed=False):

    results = []  *# каждый элемент: (V, E, DFS\_time, BFS\_time)*

    for V in vertices\_list:

        E = V \* edge\_multiplier

        graph = Graph(V, edges=E, directed=directed)

        start, target = 0, V - 1

        t1 = time.perf\_counter\_ns()

        graph.DFS(start, target)

        t2 = time.perf\_counter\_ns()

        dfs\_time = (t2 - t1) / 1e9

        t1 = time.perf\_counter\_ns()

        graph.BFS(start, target)

        t2 = time.perf\_counter\_ns()

        bfs\_time = (t2 - t1) / 1e9

        results.append((V, E, dfs\_time, bfs\_time))

    print(**f**"\n{test\_name}:")

    print("V, E, DFS, BFS")

    for (V, E, dfs\_t, bfs\_t) in results:

        print(**f**"{V},{E},{format\_time(dfs\_t)},{format\_time(bfs\_t)}")

    return results

**def** plot\_performance(test\_name, results):

    vertices = [r[0] for r in results]

    dfs\_times = [r[2] for r in results]

    bfs\_times = [r[3] for r in results]

    plt.figure(figsize=(8, 6))

    plt.title(**f**"BFS vs DFS Execution Times ({test\_name})")

    plt.xlabel("Number of Vertices")

    plt.ylabel("Time (s)")

    plt.plot(vertices, dfs\_times, color='blue', marker='o', label='DFS')

    plt.plot(vertices, bfs\_times, color='red', marker='o', label='BFS')

    plt.legend()

    filename = **f**"results\_{test\_name}.png"

    plt.savefig(filename)

    print(**f**"Сохранён график: {filename}")

**def** main():

    demo\_graph\_results()

    vertices\_test1 = list(range(100, 1001, 100))

    res1 = run\_performance\_test("Test1", vertices\_test1, edge\_multiplier=20, directed=False)

    plot\_performance("Test1", res1)

    vertices\_test2 = list(range(100, 1001, 100))

    res2 = run\_performance\_test("Test2", vertices\_test2, edge\_multiplier=20, directed=True)

    plot\_performance("Test2", res2)

    vertices\_test3 = list(range(10, 101, 10))

    res3 = run\_performance\_test("Test3", vertices\_test3, edge\_multiplier=2, directed=False)

    plot\_performance("Test3", res3)

    vertices\_test4 = list(range(10, 101, 10))

    res4 = run\_performance\_test("Test4", vertices\_test4, edge\_multiplier=2, directed=True)

    plot\_performance("Test4", res4)

    csv\_filename = "performance.csv"

    with open(csv\_filename, "w", newline="") as f:

        writer = csv.writer(f)

        writer.writerow(["Test", "Vertices", "Edges", "DFS\_time", "BFS\_time"])

        for test\_name, res in zip(["Test1", "Test2", "Test3", "Test4"], [res1, res2, res3, res4]):

            for (V, E, dfs\_t, bfs\_t) in res:

                writer.writerow([test\_name, V, E, format\_time(dfs\_t), format\_time(bfs\_t)])

    print(**f**"\nРезультаты замеров записаны в {csv\_filename}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

Кратчайший путь (BFS) из 0 в 9: [0, 4, 8, 9]

Путь (DFS) из 0 в 9: [0, 4, 8, 5, 2, 6, 3, 9]

Длина пути (BFS): 4

Длина пути (DFS): 8

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 19: [0, 19]

Путь (DFS) из 0 в 19: [0, 12, 1, 3, 13, 6, 2, 4, 17, 8, 16, 7, 11, 10, 19]

Длина пути (BFS): 2

Длина пути (DFS): 15

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 29: [0, 12, 13, 29]

Путь (DFS) из 0 в 29: [0, 18, 17, 21, 23, 14, 11, 1, 12, 15, 8, 6, 9, 13, 20, 24, 22, 28, 27, 7, 16, 5, 29]

Длина пути (BFS): 4

Длина пути (DFS): 23

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 39: [0, 10, 39]

Путь (DFS) из 0 в 39: [0, 10, 2, 29, 38, 5, 31, 16, 11, 39]

Длина пути (BFS): 3

Длина пути (DFS): 10

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 49: [0, 22, 23, 49]

Путь (DFS) из 0 в 49: [0, 36, 10, 45, 20, 38, 24, 39, 32, 30, 19, 28, 41, 48, 49]

Длина пути (BFS): 4

Длина пути (DFS): 15

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 59: [0, 34, 22, 55, 3, 4, 59]

Путь (DFS) из 0 в 59: [0, 34, 22, 13, 7, 55, 2, 21, 19, 27, 18, 15, 37, 25, 16, 11, 3, 4, 29, 49, 41, 23, 40, 38, 12, 5, 30, 47, 35, 43, 33, 8, 59]

Длина пути (BFS): 7

Длина пути (DFS): 33

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 69: [0, 53, 57, 52, 69]

Путь (DFS) из 0 в 69: [0, 18, 11, 24, 9, 2, 27, 37, 67, 56, 61, 8, 13, 55, 10, 49, 15, 60, 25, 50, 23, 47, 65, 64, 33, 66, 63, 7, 44, 42, 12, 4, 69]

Длина пути (BFS): 5

Длина пути (DFS): 33

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 79: [0, 1, 66, 79]

Путь (DFS) из 0 в 79: [0, 1, 35, 70, 59, 76, 50, 44, 13, 68, 3, 32, 74, 79]

Длина пути (BFS): 4

Длина пути (DFS): 14

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 89: [0, 33, 31, 89]

Путь (DFS) из 0 в 89: [0, 18, 49, 70, 39, 86, 72, 67, 17, 24, 52, 7, 79, 78, 69, 68, 88, 57, 84, 22, 16, 59, 60, 58, 51, 63, 33, 62, 25, 42, 74, 43, 23, 28, 54, 6, 4, 81, 47, 32, 15, 82, 13, 46, 36, 9, 10, 26, 37, 41, 31, 89]

Длина пути (BFS): 4

Длина пути (DFS): 52

--------------------------------------------------

Кратчайший путь (BFS) из 0 в 99: [0, 94, 61, 69, 99]

Путь (DFS) из 0 в 99: [0, 94, 70, 74, 93, 10, 75, 92, 99]

Длина пути (BFS): 5

Длина пути (DFS): 9

--------------------------------------------------

PS D:\wowik> & D:/wowik/.venv/Scripts/python.exe "d:/wowik/import random.py"

Результаты расчётов:

Выдача матрицы смежности:

[0 1 1 1 1]

[1 0 1 1 1]

[1 1 0 1 1]

[1 1 1 0 1]

[1 1 1 1 0]

Выдача матрицы инцидентности:

[0 1 0 0 1 0 0 0 1 1]

[0 1 1 1 0 0 0 1 0 0]

[1 0 1 0 1 0 1 0 0 0]

[1 0 0 0 0 1 0 1 0 1]

[0 0 0 1 0 1 1 0 1 0]

Выдача списка смежности:

4: [1 3 2 0]

3: [2 4 1 0]

2: [3 1 0 4]

1: [0 2 4 3]

0: [1 2 4 3]

Выдача списка ребер:

2 - 3

1 - 0

2 - 1

1 - 4

2 - 0

3 - 4

2 - 4

3 - 1

0 - 4

3 - 0

Кратчайший путь (BFS) из 0 в 4: [0 4]

Путь (DFS) из 0 в 4: [0 1 2 3 4]

Длина пути (BFS): 2

Длина пути (DFS): 5

Test1:

V, E, DFS, BFS

100,2000,23.7µs,163.6µs

200,4000,156.8µs,255.2µs

300,6000,131.0µs,208.5µs

400,8000,10.0µs,98.0µs

500,10000,139.2µs,649.3µs

600,12000,142.6µs,1.102ms

700,14000,286.4µs,1.035ms

800,16000,183.6µs,664.4µs

900,18000,25.02ms,931.3µs

1000,20000,288.9µs,3.82ms

Сохранён график: results\_Test1.png

Test2:

V, E, DFS, BFS

100,2000,46.1µs,35.3µs

200,4000,92.3µs,125.1µs

300,6000,27.4µs,80.9µs

400,8000,267.2µs,274.7µs

500,10000,174.2µs,832.6µs

600,12000,335.4µs,1.493ms

700,14000,83.3µs,102.6µs

800,16000,45.1µs,286.1µs

900,18000,164.5µs,3.725ms

1000,20000,382.4µs,1.482ms

Сохранён график: results\_Test2.png

Test3:

V, E, DFS, BFS

10,20,4.7µs,4.8µs

20,40,2.5µs,2.6µs

30,60,4.5µs,3.7µs

40,80,1.9µs,3.0µs

50,100,6.2µs,8.0µs

60,120,13.0µs,18.5µs

70,140,12.5µs,9.0µs

80,160,18.2µs,7.6µs

90,180,25.0µs,29.3µs

100,200,5.7µs,17.8µs

Сохранён график: results\_Test3.png

Test4:

V, E, DFS, BFS

10,20,5.3µs,4.7µs

20,40,5.7µs,4.6µs

30,60,1.8µs,1.8µs

40,80,1.2µs,0s

50,100,9.7µs,1.8µs

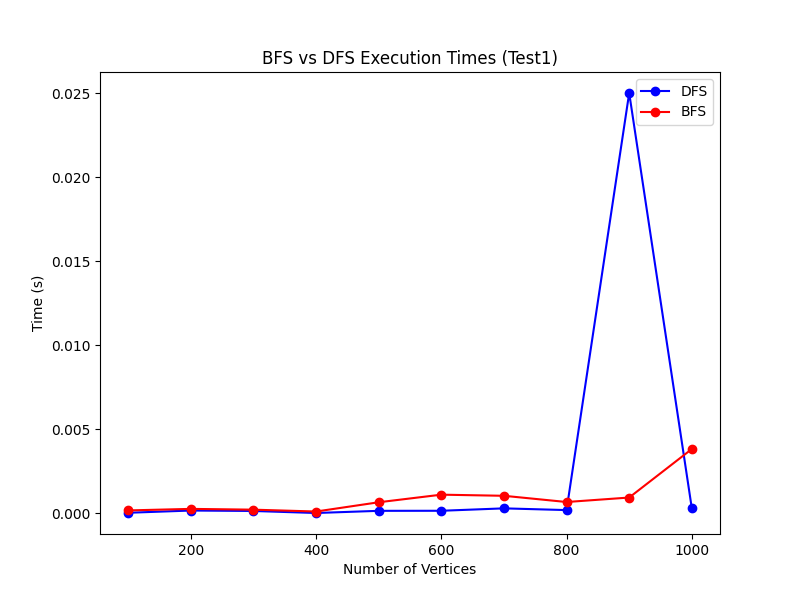
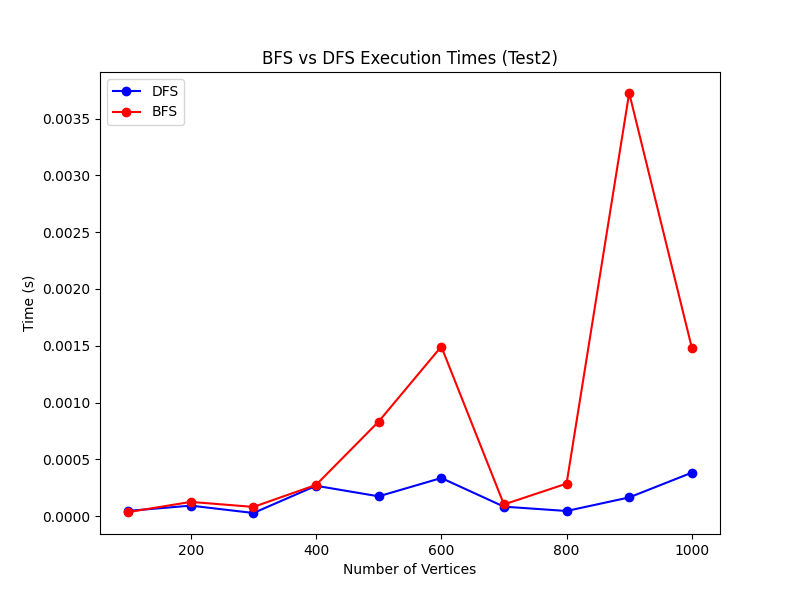
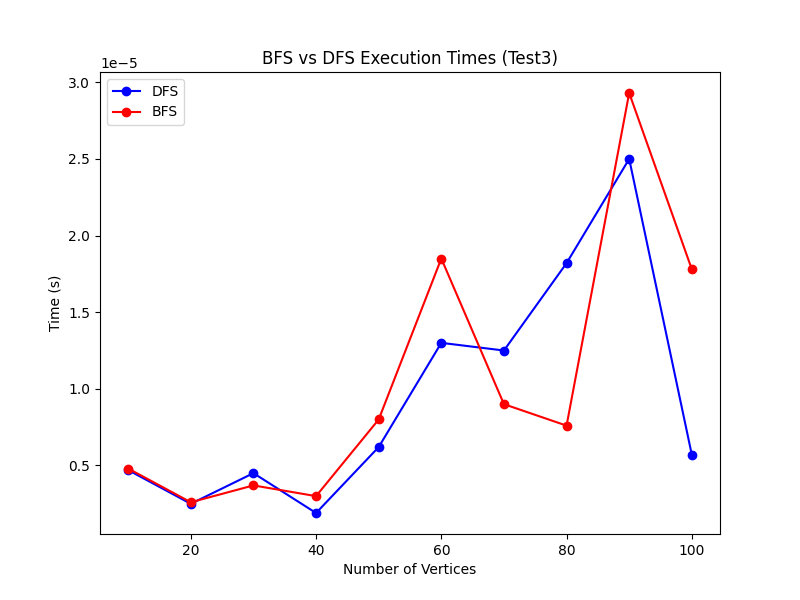
60,120,14.7µs,12.8µs

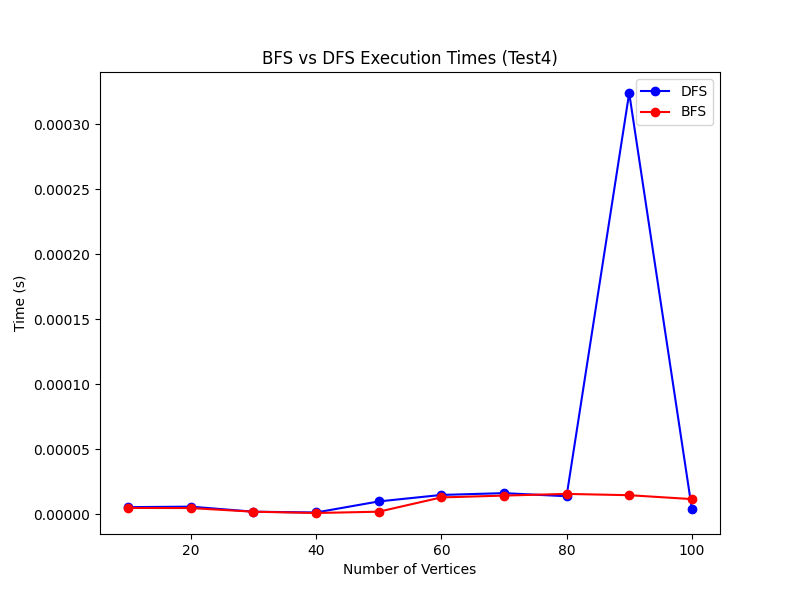
70,140,16.1µs,14.2µs

80,160,13.7µs,15.5µs

90,180,324.1µs,14.5µs

100,200,4.2µs,11.5µs

Сохранён график: results\_Test4.png   



# Заключение.

Алгоритмы обхода графа в ширину (BFS) и в глубину (DFS) не всегда быстро находят путь, поскольку их время работы напрямую зависит от количества вершин V и рёбер E.

**BFS** последовательно проверяет все вершины на каждом уровне, что это может быть долго, если граф сильно связный или содержит много рёбер.

**DFS** может "уходить" по одному пути в глубину, прежде чем вернуться и попробовать другие маршруты, это часто приводит к поиску неэффективного пути, особенно в графах с большим числом рёбер.

Оба алгоритма применяются в различных ситуациях в зависимости от требований к задаче: BFS лучше подходит для нахождения кратчайшего пути, а DFS может быть полезен в задачах, требующих полного обхода всех возможных путей.