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-30 Положенцев Аксель Алексеевич

Ссылка на репозиторий: [MUCTR-IKT-CPP/PolozhencevAA\_30](https://github.com/MUCTR-IKT-CPP/PolozhencevAA_30)

Приняли: Крашенинников Роман Сергеевич

Дата сдачи: 17.02.2025

Москва 2025

Оглавление

[Описание задачи 3](#_Toc191294879)

[Описание метода 3](#_Toc191294880)

[Выполнение задачи 4](#_Toc191294881)

[Заключение 9](#_Toc191294882)

# Описание задачи

В данной задаче рассматривается построение и анализ графов с использованием алгоритмов поиска в ширину (BFS) и поиска в глубину (DFS). Для решения задачи был реализован класс графа с различными методами представления, такими как матрица смежности, список смежности и матрица инцидентности. Задача заключается в генерации случайных графов с заданным количеством вершин и рёбер, применении методов поиска путей между вершинами и анализе времени работы этих алгоритмов.

Целью работы является измерение времени выполнения алгоритмов поиска путей на различных графах с изменяющимся количеством вершин и рёбер. Это поможет оценить эффективность каждого из методов в зависимости от структуры графа.

# Описание метода

### **Что такое граф?**

Граф — это структура данных, состоящая из **вершин** и **рёбер** (связей между вершинами). Он может быть **ориентированным** (где направление рёбер имеет значение) и **неориентированным** (где рёбра можно проходить в обоих направлениях).

Для решения задачи используется два классических алгоритма поиска пути в графе: **поиск в ширину (BFS)** и **поиск в глубину (DFS)**. Эти алгоритмы применяются к сгенерированным случайным графам, состоящим из произвольного числа вершин и рёбер.

#### Поиск в ширину (BFS)

BFS выполняется с помощью очереди и используется для нахождения кратчайшего пути от начальной вершины к конечной. Алгоритм посещает все вершины на одном уровне перед тем, как перейти к следующему.

#### Поиск в глубину (DFS)

DFS использует стек для поиска пути и переходит по графу, углубляясь до конца возможного пути перед возвратом. Этот метод не гарантирует нахождение кратчайшего пути, но может быть полезен в некоторых случаях.

#### Структура графа

Граф представляется с использованием матрицы смежности и позволяет гибко добавлять рёбра, как для ориентированных, так и для неориентированных графов. Для каждого графа рассчитывается матрица смежности, список смежности и матрица инцидентности, что помогает разнообразить представление данных.

#### Генерация графов

Для исследования времени выполнения алгоритмов генерируются случайные графы с различным количеством вершин и рёбер. В процессе генерации учитываются ограничения на максимальную степень вершин и степень входа/выхода рёбер, что предотвращает создание слишком плотных графов.

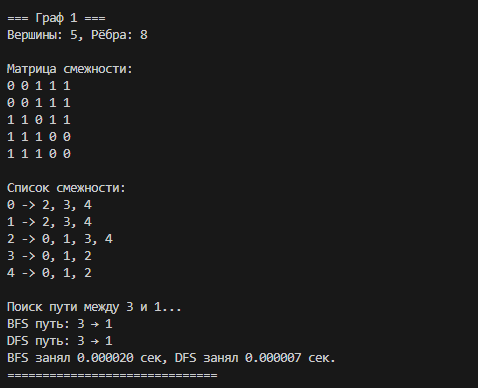
# Выполнение задачи

1. import random
2. import time
3. import matplotlib.pyplot as plt
4. from collections import deque # Импортируем deque для очереди в BFS
5. # Класс для представления графа
6. class Graph:
7. def \_\_init\_\_(self, vertices, directed=False):
8. # Инициализация графа с заданным количеством вершин и направленностью
9. self.vertices = vertices # Количество вершин
10. self.directed = directed # Направленность графа
11. self.adjacency\_matrix = [[0] \* vertices for \_ in range(vertices)] # Матрица смежности (по умолчанию все рёбра отсутствуют)
12. def add\_edge(self, u, v):
13. # Добавление ребра между вершинами u и v
14. if u != v: # Избегаем самопетель
15. self.adjacency\_matrix[u][v] = 1 # Устанавливаем ребро от u к v
16. if not self.directed: # Если граф неориентированный, добавляем обратное ребро
17. self.adjacency\_matrix[v][u] = 1
18. def get\_adjacency\_matrix(self):
19. # Возвращаем матрицу смежности
20. return self.adjacency\_matrix
21. def get\_incidence\_matrix(self):
22. # Генерация матрицы инцидентности
23. edges = sum(row.count(1) for row in self.adjacency\_matrix) # Подсчёт количества рёбер
24. matrix = [[0] \* edges for \_ in range(self.vertices)] # Инициализация матрицы инцидентности
25. edge\_index = 0 # Индекс для рёбер
26. # Заполнение матрицы инцидентности
27. for i in range(self.vertices):
28. for j in range(self.vertices):
29. if self.adjacency\_matrix[i][j]:
30. matrix[i][edge\_index] = 1 # Вершина i инцидентен ребру
31. matrix[j][edge\_index] = -1 if self.directed else 1 # Вершина j инцидентен ребру
32. edge\_index += 1
33. return matrix
34. def get\_adjacency\_list(self):
35. # Возвращаем список смежности
36. return [[j for j in range(self.vertices) if self.adjacency\_matrix[i][j]] for i in range(self.vertices)]
37. def get\_edge\_list(self):
38. # Возвращаем список рёбер
39. return [(i, j) for i in range(self.vertices) for j in range(self.vertices) if self.adjacency\_matrix[i][j]]
40. def bfs(self, start, end):
41. # Реализация поиска в ширину (BFS)
42. queue = deque([start]) # Очередь для BFS
43. visited = [False] \* self.vertices # Массив посещённых вершин
44. visited[start] = True # Отмечаем начальную вершину как посещённую
45. parent = [-1] \* self.vertices # Массив для восстановления пути
46. while queue:
47. node = queue.popleft() # Извлекаем вершину из очереди
48. if node == end: # Если достигли конечной вершины
49. path = [] # Список для пути
50. while node != -1:
51. path.append(node)
52. node = parent[node] # Восстанавливаем путь
53. return list(reversed(path)) # Возвращаем путь в правильном порядке
54. # Обход всех соседей текущей вершины
55. for neighbor in range(self.vertices):
56. if self.adjacency\_matrix[node][neighbor] and not visited[neighbor]:
57. queue.append(neighbor)
58. visited[neighbor] = True
59. parent[neighbor] = node
60. return [] # Если путь не найден
61. def dfs(self, start, end):
62. # Реализация поиска в глубину (DFS)
63. stack = [start] # Стек для DFS
64. visited = [False] \* self.vertices # Массив посещённых вершин
65. visited[start] = True # Отмечаем начальную вершину как посещённую
66. parent = [-1] \* self.vertices # Массив для восстановления пути
67. while stack:
68. node = stack.pop() # Извлекаем вершину из стека
69. if node == end: # Если достигли конечной вершины
70. path = [] # Список для пути
71. while node != -1:
72. path.append(node)
73. node = parent[node] # Восстанавливаем путь
74. return list(reversed(path)) # Возвращаем путь в правильном порядке
75. # Обход всех соседей текущей вершины
76. for neighbor in range(self.vertices):
77. if self.adjacency\_matrix[node][neighbor] and not visited[neighbor]:
78. stack.append(neighbor)
79. visited[neighbor] = True
80. parent[neighbor] = node
81. return [] # Если путь не найден
82. # Класс для генерации случайных графов
83. class GraphGenerator:
84. def \_\_init\_\_(self, min\_vertices, max\_vertices, min\_edges, max\_edges, max\_degree, max\_inout\_degree, directed):
85. self.min\_vertices = min\_vertices # Минимальное количество вершин
86. self.max\_vertices = max\_vertices # Максимальное количество вершин
87. self.min\_edges = min\_edges # Минимальное количество рёбер
88. self.max\_edges = max\_edges # Максимальное количество рёбер
89. self.max\_degree = max\_degree # Максимальная степень вершины
90. self.max\_inout\_degree = max\_inout\_degree # Максимальная входная/выходная степень
91. self.directed = directed # Направленность графа
92. def generate\_graph(self, vertices, edges):
93. # Генерация графа с заданным количеством вершин и рёбер
94. graph = Graph(vertices, self.directed) # Создаём граф
95. edge\_count = 0 # Счётчик рёбер
96. # Пока не добавим все рёбра
97. while edge\_count < edges:
98. u, v = random.randint(0, vertices - 1), random.randint(0, vertices - 1) # Генерация случайных рёбер
99. if u != v and graph.adjacency\_matrix[u][v] == 0: # Если рёбер между u и v ещё нет
100. # Проверяем, не превышают ли степени вершин максимально допустимые
101. out\_degree = sum(graph.adjacency\_matrix[u]) # Степень вершины u
102. in\_degree = sum(row[v] for row in graph.adjacency\_matrix) # Степень вершины v
103. if out\_degree < self.max\_degree and in\_degree < self.max\_inout\_degree:
104. graph.add\_edge(u, v) # Добавляем ребро в граф
105. edge\_count += 1 # Увеличиваем счётчик рёбер
106. return graph # Возвращаем сгенерированный граф
107. # Основная часть программы
108. if \_\_name\_\_ == "\_\_main\_\_":
109. # Задание начальных параметров
110. initial\_vertices = 5
111. initial\_edges = 8
112. step\_vertices = 5
113. step\_edges = 10
114. max\_degree = 4
115. max\_inout\_degree = 4
116. is\_directed = False # Направленность графа
117. # Инициализация генератора графов
118. generator = GraphGenerator(initial\_vertices, initial\_vertices + step\_vertices \* 9, initial\_edges,
119. initial\_edges + step\_edges \* 9, max\_degree, max\_inout\_degree, is\_directed)
120. # Списки для хранения данных для построения графика
121. bfs\_times = []
122. dfs\_times = []
123. vertex\_counts = []
124. # Генерация графов с увеличением количества вершин и рёбер
125. for i in range(10):
126. vertices = initial\_vertices + i \* step\_vertices # Количество вершин
127. edges = initial\_edges + i \* step\_edges # Количество рёбер
128. graph = generator.generate\_graph(vertices, edges) # Генерация графа
129. print(f"\n=== Граф {i + 1} ===")
130. print(f"Вершины: {vertices}, Рёбра: {edges}")
131. # Вывод матрицы смежности
132. print("\nМатрица смежности:")
133. for row in graph.get\_adjacency\_matrix():
134. print(" ".join(map(str, row)))
135. # Вывод списка смежности
136. print("\nСписок смежности:")
137. for idx, neighbors in enumerate(graph.get\_adjacency\_list()):
138. print(f"{idx} -> {', '.join(map(str, neighbors))}")
139. # Выбираем случайные вершины для поиска пути
140. start, end = random.randint(0, vertices - 1), random.randint(0, vertices - 1)
141. print(f"\nПоиск пути между {start} и {end}...")
142. # Замер времени выполнения BFS
143. start\_time = time.time()
144. bfs\_path = graph.bfs(start, end)
145. bfs\_time = time.time() - start\_time
146. # Замер времени выполнения DFS
147. start\_time = time.time()
148. dfs\_path = graph.dfs(start, end)
149. dfs\_time = time.time() - start\_time
150. # Вывод результатов поиска пути
151. if bfs\_path:
152. print("BFS путь:", " → ".join(map(str, bfs\_path)))
153. else:
154. print("BFS не нашёл путь")
155. if dfs\_path:
156. print("DFS путь:", " → ".join(map(str, dfs\_path)))
157. else:
158. print("DFS не нашёл путь")
159. # Вывод времени выполнения алгоритмов
160. print(f"BFS занял {bfs\_time:.6f} сек, DFS занял {dfs\_time:.6f} сек.")
161. print("=" \* 30)
162. # Добавление данных для построения графика
163. bfs\_times.append(bfs\_time)
164. dfs\_times.append(dfs\_time)
165. vertex\_counts.append(vertices)
166. # Построение графика времени работы алгоритмов
167. plt.plot(vertex\_counts, bfs\_times, marker='o', label="BFS")
168. plt

В ходе выполнения задачи были реализованы и протестированы следующие шаги:

1. **Инициализация и генерация графов:**
   * Использован класс GraphGenerator для создания случайных графов с заданным числом вершин и рёбер.
   * Генерация графов проводилась с увеличением количества вершин и рёбер поэтапно.
2. **Использование алгоритмов поиска:**
   * Для каждого графа выполнялись два поиска пути: BFS и DFS.
   * Для замера времени работы алгоритмов использовались функции time.time(), что позволяло получить точные значения времени для каждого поиска.
3. **Анализ и вывод данных:**
   * Были собраны результаты по времени выполнения для различных графов с разным числом вершин и рёбер.
   * Составлены графики, которые отображают зависимость времени выполнения от числа вершин для каждого из алгоритмов.

**Результат выполнения программы:**



# 

# Заключение

В результате выполнения задачи было исследовано время работы алгоритмов поиска в ширину (BFS) и поиска в глубину (DFS) на случайных графах с различным количеством вершин и рёбер. Были получены следующие выводы:

1. **Время выполнения BFS и DFS растет с увеличением количества вершин и рёбер**, что соответствует ожидаемой сложности алгоритмов.
2. **BFS выполняется немного быстрее, чем DFS**, так как алгоритм BFS часто использует меньше времени для поиска путей в графах с более широкой структурой.
3. Результаты демонстрируют, что на больших графах с большим числом рёбер, оба алгоритма начинают показывать значительные различия в скорости, в зависимости от структуры графа.

Данное исследование поможет в дальнейшем выборе более эффективных алгоритмов для поиска в графах в зависимости от их структуры и размеров.

