Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительной техники»

**ОТЧЕТ**

по лабораторной работе №8

по курсу «Логика и основы алгоритмизации в инженерных задачах.»

на тему: «Обход графа в ширину.»

Выполнили**:**

студенты группы 22ВВВ2

Коробкин В. Ю.

Горбатов К. В.

Проверили:

Митрохин М. А.

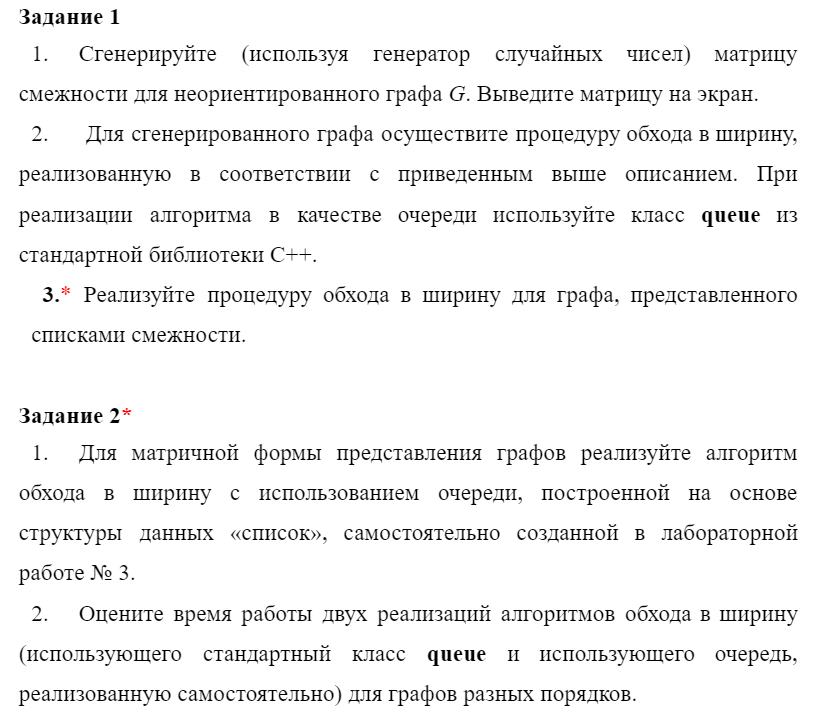
Акифьев И.В.

Пенза 2023

**Цель работы**

Научиться реализовывать алгоритм поиска в ширину.

**Лабораторное задание**

****

**Листинг**

Программа 8.1 log – Задание 1 (1,2)

from collections import deque

import random

import time

def generate\_random\_adjacency\_matrix(n, p):

adjacency\_matrix = [[0] \* n for \_ in range(n)]

for i in range(n):

for j in range(i + 1, n):

# Генерируем случайное число в диапазоне [0, 1]

random\_value = random.random()

# Если случайное число меньше или равно p, то устанавливаем ребро

if random\_value <= p:

adjacency\_matrix[i][j] = 1

adjacency\_matrix[j][i] = 1

return adjacency\_matrix

def bfs(adj\_matrix, start):

visited = [False] \* len(adj\_matrix)

queue = deque([start])

visited[start] = True

start\_time = time.time() # Засекаем время начала выполнения

while queue:

vertex = queue.popleft()

print(f"Посещаем вершину {vertex}")

for adjacent, connected in enumerate(adj\_matrix[vertex]):

if connected and not visited[adjacent]:

queue.append(adjacent)

visited[adjacent] = True

end\_time = time.time() # Засекаем время окончания выполнения

elapsed\_time = end\_time - start\_time # Вычисляем время выполнения

print(f"Время выполнения BFS: {elapsed\_time} секунд")

#вероятность ребра #####################################

p = 0.5 #####################################

while True:

print("Введите -1 если хотите завершить программу ")

n = int(input("Введите размер матрицы смежности: "))

if n == -1:

break

# Генерируем матрицу смежности

adjacency\_matrix = generate\_random\_adjacency\_matrix(n, p)

# Определяем размер графа

edge\_count = sum(sum(row) for row in adjacency\_matrix) // 2

# Определяем количество вершин

graph\_size = len(adjacency\_matrix)

# Выводим информацию о графе

print("Матрица смежности:")

for row in adjacency\_matrix:

print(row)

start\_vertex = int(input("Введите начальную вершину для обхода в ширину: "))

bfs(adjacency\_matrix, start\_vertex)

Программа 8.2 log – Задание 1 (3)

from collections import deque

import random

def generate\_random\_adjacency\_matrix(n, p):

adjacency\_matrix = [[0] \* n for \_ in range(n)]

for i in range(n):

for j in range(i + 1, n):

# Генерируем случайное число в диапазоне [0, 1]

random\_value = random.random()

# Если случайное число меньше или равно p, то устанавливаем ребро

if random\_value <= p:

adjacency\_matrix[i][j] = 1

adjacency\_matrix[j][i] = 1

return adjacency\_matrix

def convert\_to\_adjacency\_list(adj\_matrix):

adj\_list = {}

for i, row in enumerate(adj\_matrix):

adj\_list[i] = [j for j, val in enumerate(row) if val == 1]

return adj\_list

def bfs(adj\_list, start):

visited = [False] \* len(adj\_list)

queue = deque([start])

visited[start] = True

while queue:

vertex = queue.popleft()

print(f"Посещаем вершину {vertex}")

for adjacent in adj\_list[vertex]:

if not visited[adjacent]:

queue.append(adjacent)

visited[adjacent] = True

#вероятность ребра #####################################

p = 0.5 #####################################

while True:

print("Введите -1 если хотите завершить программу ")

n = int(input("Введите размер матрицы смежности: "))

if n == -1:

break

# Генерируем матрицу смежности

adjacency\_matrix = generate\_random\_adjacency\_matrix(n, p)

# Получаем список смежности из матрицы смежности

adjacency\_list = convert\_to\_adjacency\_list(adjacency\_matrix)

# Определяем размер графа

edge\_count = sum(sum(row) for row in adjacency\_matrix) // 2

# Определяем количество вершин

graph\_size = len(adjacency\_matrix)

# Выводим список смежности

print("Список смежности:")

for vertex, neighbors in adjacency\_list.items():

print(f"{vertex}: {neighbors}")

start\_vertex = int(input("Введите начальную вершину для обхода в ширину: "))

bfs(adjacency\_list, start\_vertex)

Программа 8.3 log – Задание 2 (1)

import random

class Node:

def \_\_init\_\_(self, inf):

self.inf = inf # полезная информация

self.next = None # ссылка на следующий элемент

head = None # указатель на первый элемент очереди

tail = None # указатель на последний элемент очереди

dlinna = 0 # переменная для хранения длины очереди

def get\_struct():

s = input("Введите начальную вершину для обхода в ширину: ") # вводим данные

if not s:

print("Запись не была произведена")

return None

p = Node(s)

p.next = None

return p # возвращаем экземпляр созданной структуры Node

def get\_struct2(adjacent3):

s = adjacent3 # вводим данные

if not s:

print("Запись не была произведена")

return None

p = Node(s)

p.next = None

return p # возвращаем экземпляр созданной структуры Node

def enqueue():

global head, tail, dlinna

p = get\_struct()

if head is None and p is not None: # если очереди нет, то устанавливаем голову и хвост очереди

head = p

tail = p

elif head is not None and p is not None: # добавляем элемент в конец очереди

tail.next = p

tail = p

dlinna += 1 # увеличиваем длину очереди

def enqueue2(adjacent2):

global head, tail, dlinna

p = get\_struct2(adjacent2)

if head is None and p is not None: # если очереди нет, то устанавливаем голову и хвост очереди

head = p

tail = p

elif head is not None and p is not None: # добавляем элемент в конец очереди

tail.next = p

tail = p

dlinna += 1 # увеличиваем длину очереди

def dequeue():

global head, dlinna

if head is None:

print("Очередь пуста")

return

removed = head

head = head.next

dlinna -= 1

return removed.inf

def generate\_random\_adjacency\_matrix(n, p):

adjacency\_matrix = [[0] \* n for \_ in range(n)]

for i in range(n):

for j in range(i + 1, n):

random\_value = random.random()

if random\_value <= p:

adjacency\_matrix[i][j] = 1

adjacency\_matrix[j][i] = 1

return adjacency\_matrix

def bfs(adj\_matrix, start):

visited = [False] \* len(adj\_matrix)

#Используем список как очередь

enqueue()

visited[start] = True

while dlinna:

vertex = int(head.inf)

dequeue() # Извлекаем элемент из начала очереди

print(f"Посещаем вершину {vertex}")

for adjacent, connected in enumerate(adj\_matrix[vertex]):

if connected and not visited[adjacent]:

enqueue2(adjacent) # Добавляем в конец очереди

visited[adjacent] = True

# Параметры графа

p = 0.5

while True:

print("Введите -1 если хотите завершить программу")

n = int(input("Введите размер матрицы смежности: "))

if n == -1:

break

# Генерируем матрицу смежности

adjacency\_matrix = generate\_random\_adjacency\_matrix(n, p)

# Вывод информации о графе

print("Матрица смежности:")

for row in adjacency\_matrix:

print(row)

start\_vertex = 0

bfs(adjacency\_matrix, start\_vertex)

**Результаты выполнения программ:**

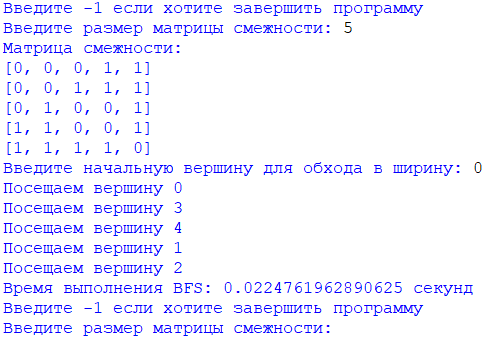


Рисунок 1. – Результат выполнения программы 8.1 log.py

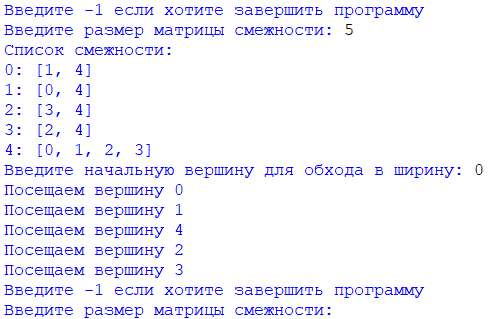


Рисунок 2. – Результат выполнения программы 8.2 log.py

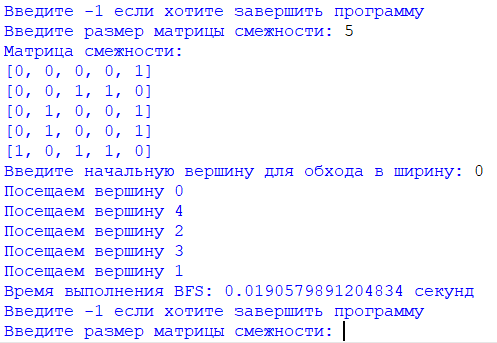


Рисунок 3. – Результат выполнения программы 8.3 log.py

**Задание 2 (2)**

Оценка времени работы алгоритмов обхода в ширину производилась с помощью функции time в программах 8.1 log.py и 8.2 log.py.

Зависимость времени работы алгоритма от размера графа:

*Алгоритм с функцией deque:*

100х100 - 0 секунд (бесконечно маленькое число)

500x500 - 0.0070 секунд

1000x1000 - 0.0484 секунд

10000x10000 - 2.7470 секунд

20000x20000 - 10.9358 секунд

Матрицы большего размера генерации не подлежат

*Алгоритм с использованием очереди:*

100х100 – 0 секунд (бесконечно маленькое число)

500x500 - 0.0066 секунд

1000x1000 - 0.0267 секунд

10000x10000 - 2.8482 секунд

20000x20000 - 11.3704 секунд

Матрицы большего размера генерации не подлежат

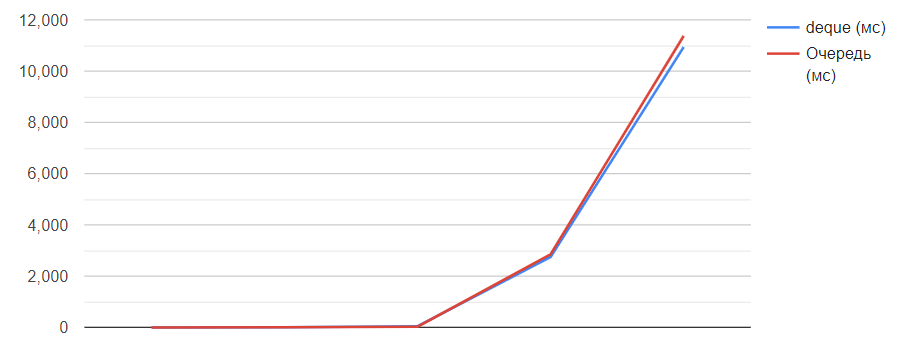


Рисунок 4. – Графики зависимости времени работы от размера матриц

**Из этих данных можно понять что:**

*Алгоритм с функцией deque* рациональней использовать для матриц больших размеров, начиная от 10000.

*Алгоритм с использованием очереди* рациональней использовать для матриц маленьких и средних размеров, не более 10000.

Можно отметить что на языке *python* особой разницы в этих алгоритмах нет, так как этот язык не позволяет создавать матрицу более 20000, и из – за этого *алгоритм с функцией deque* не может сильно оторваться по эффективности от *алгоритма с использованием очереди*.

Вывод: в ходе выполнения лабораторной работы были получены знания разработки программа, реализующей алгоритм обход графа в ширину.