Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе № 8

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Обход графа в ширину»

Выполнили:

студенты группы 22ВВВ2:

Зубриянова А.А.

Кондратьева В.И.

Приняли:

Акифьев И.В.

Митрохин М.А.

Пенза 2023

**Название**

Обход графа в ширину.

**Цель работы**

Научиться выполнять обход графа в ширину.

**Лабораторное задание**

Задание 1

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа G. Выведите матрицу на экран.

2. Для сгенерированного графа осуществите процедуру обхода в ширину, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки С++.

3.\* Реализуйте процедуру обхода в ширину для графа, представленного списками смежности.

Задание 2\*

1. Для матричной формы представления графов реализуйте алгоритм обхода в ширину с использованием очереди, построенной на основе структуры данных «список», самостоятельно созданной в лабораторной работе № 3.

2. Оцените время работы двух реализаций алгоритмов обхода в ширину (использующего стандартный класс **queue** и использующего очередь, реализованную самостоятельно) для графов разных порядков.

**Описание метода решения задачи**

Задание 1:

1. Сгенерировали (используя генератор случайных чисел) матрицу смежности для неориентированного графа G.

2. Для сгенерированного графа осуществили процедуру обхода в ширину, используя функцию void BFS1(int s). При реализации алгоритма в качестве очереди использовали класс **queue** из стандартной библиотеки С++.

3. По аналогичному алгоритму осуществили процедуру обхода в ширину для графа, представленного списками смежности (функция first\_list ).

Задание 2:

1. Для матричной формы представления графов реализовали алгоритм обхода в ширину с использованием очереди, построенной на основе структуры данных «список», которую мы создали самостоятельно в лабораторной работе № 3. Для этого использовали функции: add\_to\_list\_queue(s) и del\_from\_list().

2. Оценили время работы двух реализаций алгоритмов обхода в ширину (использующего стандартный класс **queue** и использующего очередь, реализованную самостоятельно) для графов разных порядков. Для этого использовали функции: add\_to\_list\_queue(s) и del\_from\_list().

**Листинг**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <windows.h>

#include <time.h>

#include <stdlib.h>

#include <locale.h>

#include <conio.h>

#include <iostream>

#include <queue>

using namespace std;

queue <int> Q;

int k;

int size1, h;

int\*\* arr1;

int\* vis;

void print\_vis();

void add\_to\_list\_queue(int s);

void del\_from\_list();

struct Node\* add\_elem(int s);

struct Node\* head = NULL, \* last = NULL;

struct Node

{

int n; //Номер вершины (0..N-1)

Node\* next;

};

Node\* LA; //Массив структур - списков смежности [N]

Node\* make\_LA()

{

Node\* la, \* p;

la = (Node\*)malloc(sizeof(Node) \* size1);

for (int i = 0; i < size1; i++)

{

la[i].n = i;

la[i].next = NULL;

p = &la[i];

for (int j = 0; j < size1; j++)

{

if (arr1[i][j] == 1)

{

p->next = (Node\*)malloc(sizeof(Node));

p = p->next;

p->n = j;

p->next = NULL;

}

}

}

return la;

}

void print\_LA()

{

Node\* p;

for (int i = 0; i < size1; i++)

{

p = &LA[i];

while (p != NULL)

{

printf("%d > ", p->n);

p = p->next;

}

printf("\n");

}

}

void first\_list(int l) // l - вершина, с которой начинаем обход

{

Node\* p;

p = &LA[l];

vis[l] = 1;

printf("%5d", l);

Q.push(l);

while (!Q.empty())

{

l = Q.front();

Q.pop();

//p = &LA[l];

if (vis[l] == 1)

{

p = &LA[l];

}

while (p->next != NULL)

{

if (vis[p->next->n] == 1)

{

p = p->next;

}

else

{

vis[p->next->n] = 1;

p = p->next;

l = p->n;

Q.push(l);

printf("%5d", l);

}

}

}

}

void create\_matrix() {

for (int i = 0; i < size1; i++) {

for (int j = i; j < size1; j++) {

if (i == j) { //главная диагональ == 0

arr1[i][j] = 0;

}

else { //ниже главной диагонали отразить то, что выше

arr1[i][j] = rand() % 2;

arr1[j][i] = arr1[i][j];

}

}

}

}

void print\_matrix() {

printf(" ");

for (int i = 0; i < size1; i++) {

printf("%4d ", i);

}

printf("\n");

printf("-----------------------------------------------------------\n");

for (int i = 0; i < size1; i++) {

printf("%4d|", i);

for (int j = 0; j < size1; j++) {

printf("%4d ", arr1[i][j]);

}

printf("\n");

}

printf("\n");

}

void print\_vis() {

printf("\nМассив vis:\n");

for (int i = 0; i < size1; i++) {

printf("%7d", vis[i]);

}

printf("\n\n");

}

void BFS1(int s) {

vis[s] = 1;

printf("%5d", s);

Q.push(s);

while (!Q.empty()) {

s = Q.front();

Q.pop();

for (int i = 0; i < size1; i++) {

if ((arr1[s][i] == 1) && (vis[i] == 0)) {

printf("%5d", i);

vis[i] = 1;

Q.push(i);

}

}

}

}

void BFS2(int s) { //s - с этой вершины начинаем

vis[s] = 1;

printf("%5d", s);

add\_to\_list\_queue(s);

while (head != 0) {

s = last->n;

del\_from\_list();

for (int i = 0; i < size1; i++) {

if ((arr1[s][i] == 1) && (vis[i] == 0)) {

printf("%5d", i);

vis[i] = 1;

add\_to\_list\_queue(i);

}

}

}

}

void clean\_vis() {

for (int i = 0; i < size1; i++) {

vis[i] = 0;

}

}

void add\_to\_list\_queue(int s)

{

Node\* p = NULL;

p = add\_elem(s);

if (head == NULL) // если очереди нет, то устанавливаем голову очереди

{

head = p;

last = p;

}

else // если очередь уже есть, то вставляем в начало

{

p->next = head;

head = p;

}

}

struct Node\* add\_elem(int s) // Создаем новый элемент

{

Node\* p = NULL;

p = (Node\*)malloc(sizeof(struct Node)); // выделяем память под новый элемент списка

if (p == NULL) // если память не выделилась

{

printf("Ошибка при распределении памяти\n");

exit(1);

}

p->n = s;

p->next = NULL;

return p; // возвращаем указатель на созданный элемент (где его создали, в каком месте памяти)

}

void del\_from\_list()

{

Node\* struc = head;

if (struc->next == NULL) // удаляем единственный элемент

{

free(struc);

head = NULL;

last = NULL;

return;

}

while (struc)

{

if (struc->next == last) // удаляем последний элемент

{

free(struc->next);

struc->next = NULL;

last = struc;

break;

}

struc = struc->next;

}

}

int main() {

srand(time(NULL));

setlocale(LC\_ALL, "Russian");

printf("Задание 1\n");

printf("Введите размер матрицы смежности M1: ");

scanf("%d", &size1);

printf("\n");

vis = (int\*)malloc(sizeof(int) \* size1);

arr1 = (int\*\*)malloc(sizeof(int\*) \* size1);

for (int i = 0; i < size1; i++) {

arr1[i] = (int\*)malloc(sizeof(int) \* size1);

}

printf("Матрица M1:\n");

create\_matrix();

print\_matrix();

clean\_vis();

printf("=======================================================================\n\n");

printf("Задание 2\n");

printf("Введите вершину (от 0 до %d) для начала обхода: ", size1 - 1);

scanf("%d", &h);

int l = h;

k = h;

printf("\n");

printf("я был в вершинах:");

clock\_t start, end;

double time;

start = clock();

BFS1(h);

end = clock();

time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("\n");

printf("Время работы стандартной queue:%lf\n", time);

clean\_vis();

printf("\n");

printf("=======================================================================\n\n");

printf("Задание 3\n");

printf("Список смежности S1:\n");

LA = make\_LA();

print\_LA();

printf("\n");

printf("я был в вершинах:");

first\_list(l);

printf("\n");

printf("=======================================================================\n\n");

printf("Задание 4\n");

clean\_vis();

printf("я был в вершинах:");

int s = h;

start = clock();

BFS2(s);

end = clock();

time = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("\n");

printf("Время работы нашей queue:%lf\n", time);

for (int i = 0; i < size1; ++i) free(arr1[i]);

free(arr1);

free(vis);

\_getch();

return 0;

}

**Результаты работы программы**

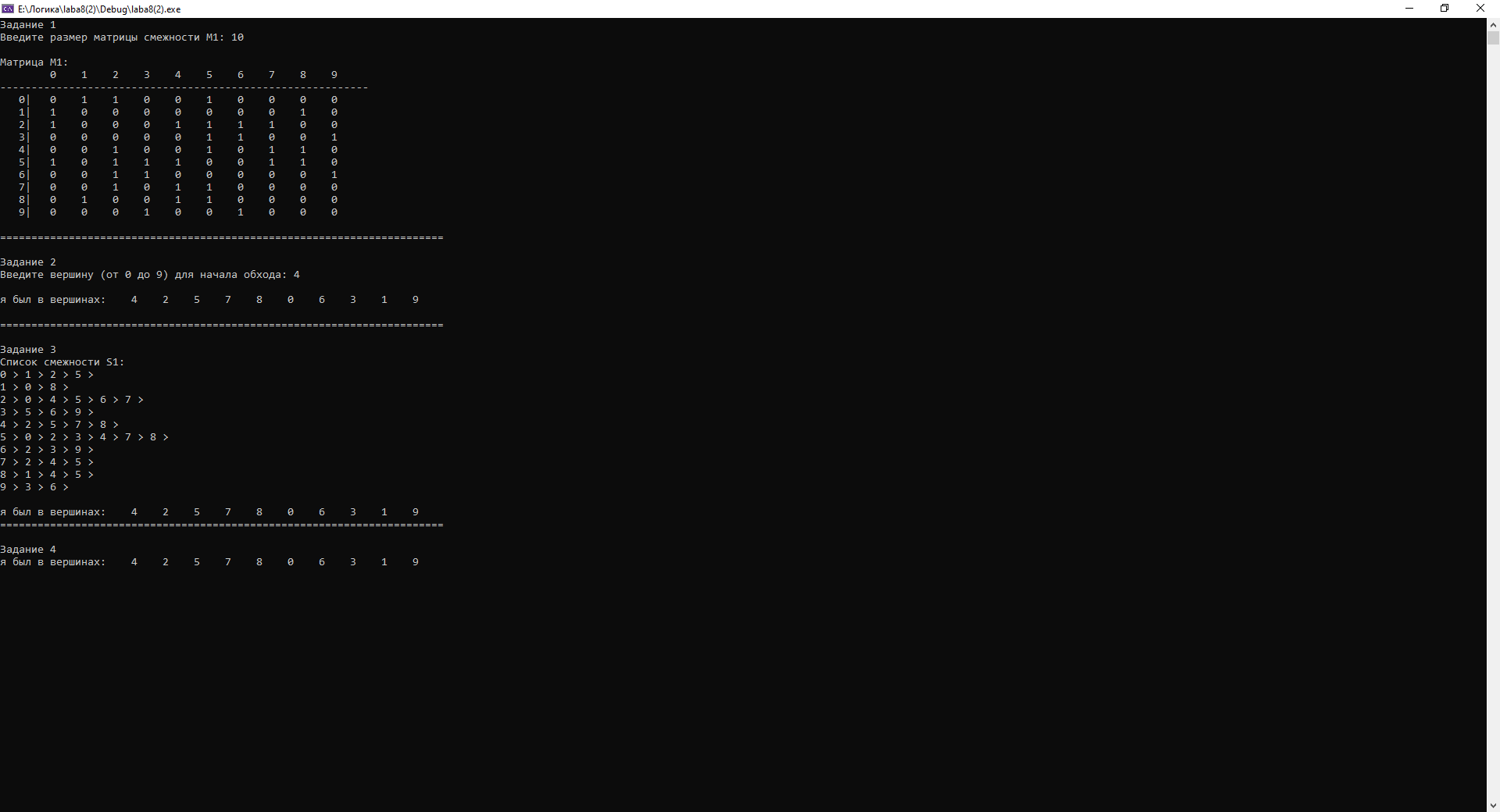


Рисунок 1 – Обход графа в глубину: рекурсивная реализация, с помощью списков смежности и через стек

Сравнение времени работы алгоритмов обхода в ширину, реализованных с помощью стандартной очереди и нашей очереди для графов разных порядков:



Рисунок 2 – Граф из 10 вершин



Рисунок 3 – Граф из 50 вершин

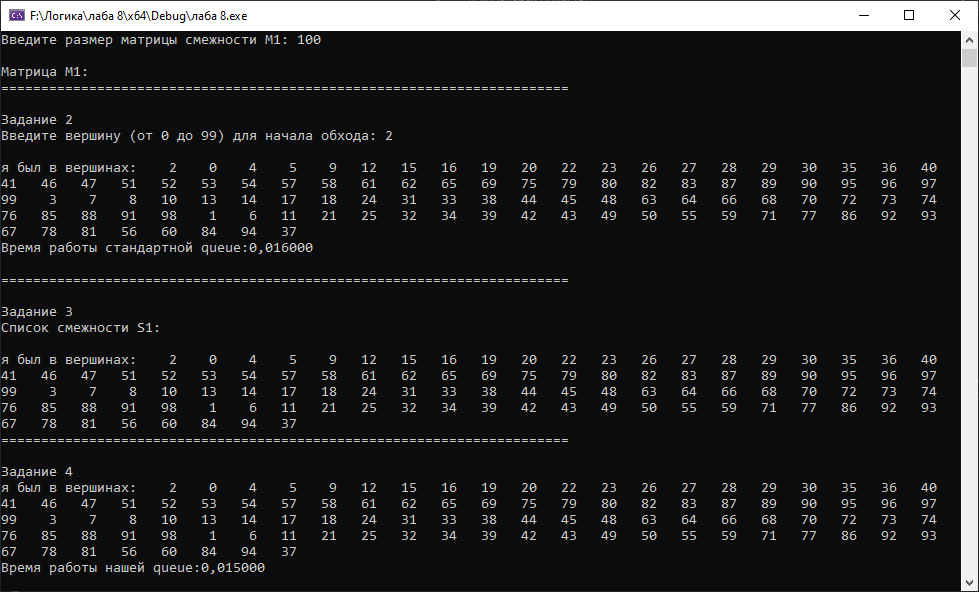


Рисунок 4 – Граф из 100 вершин

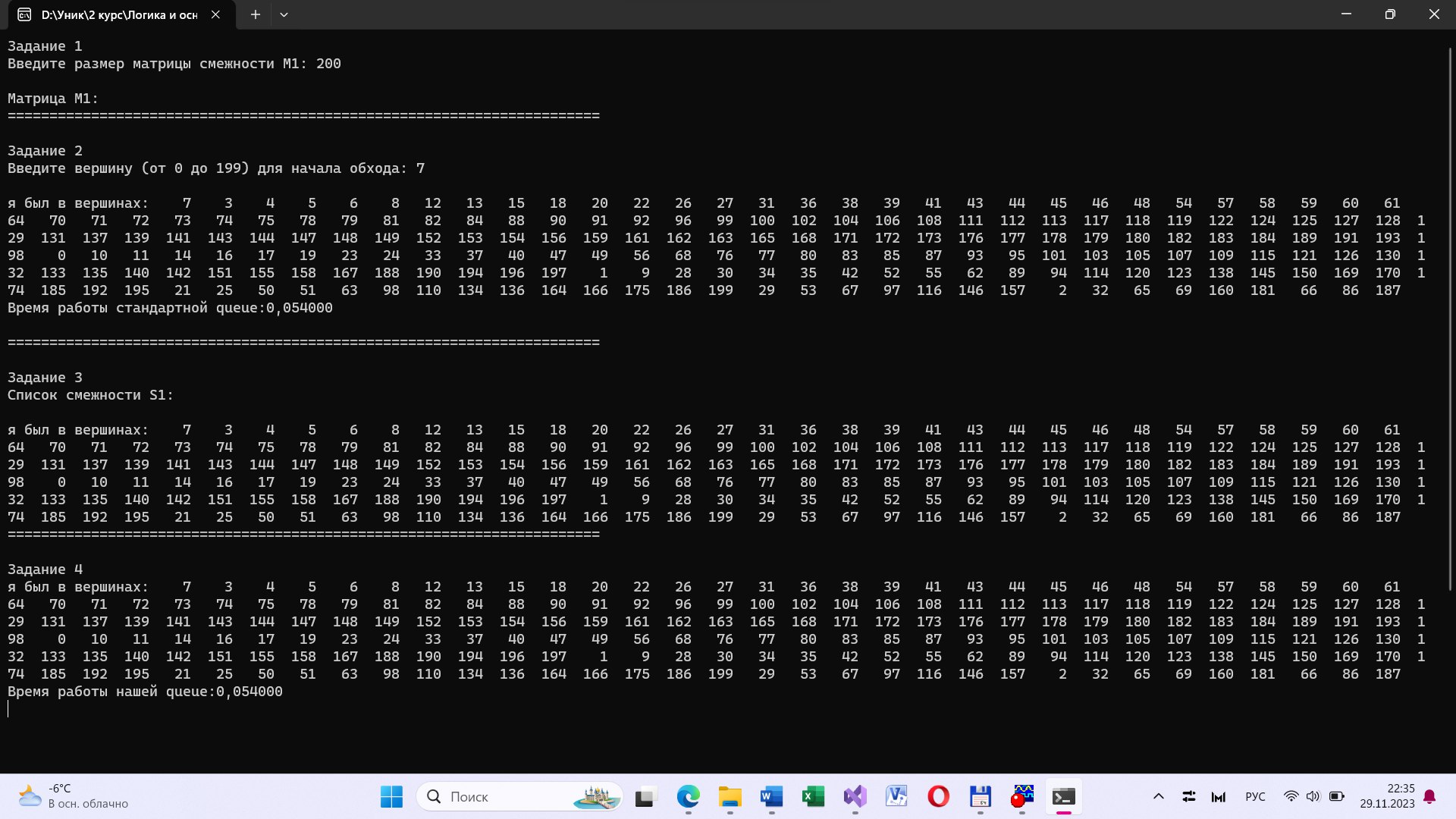


Рисунок 5 – Граф из 200 вершин

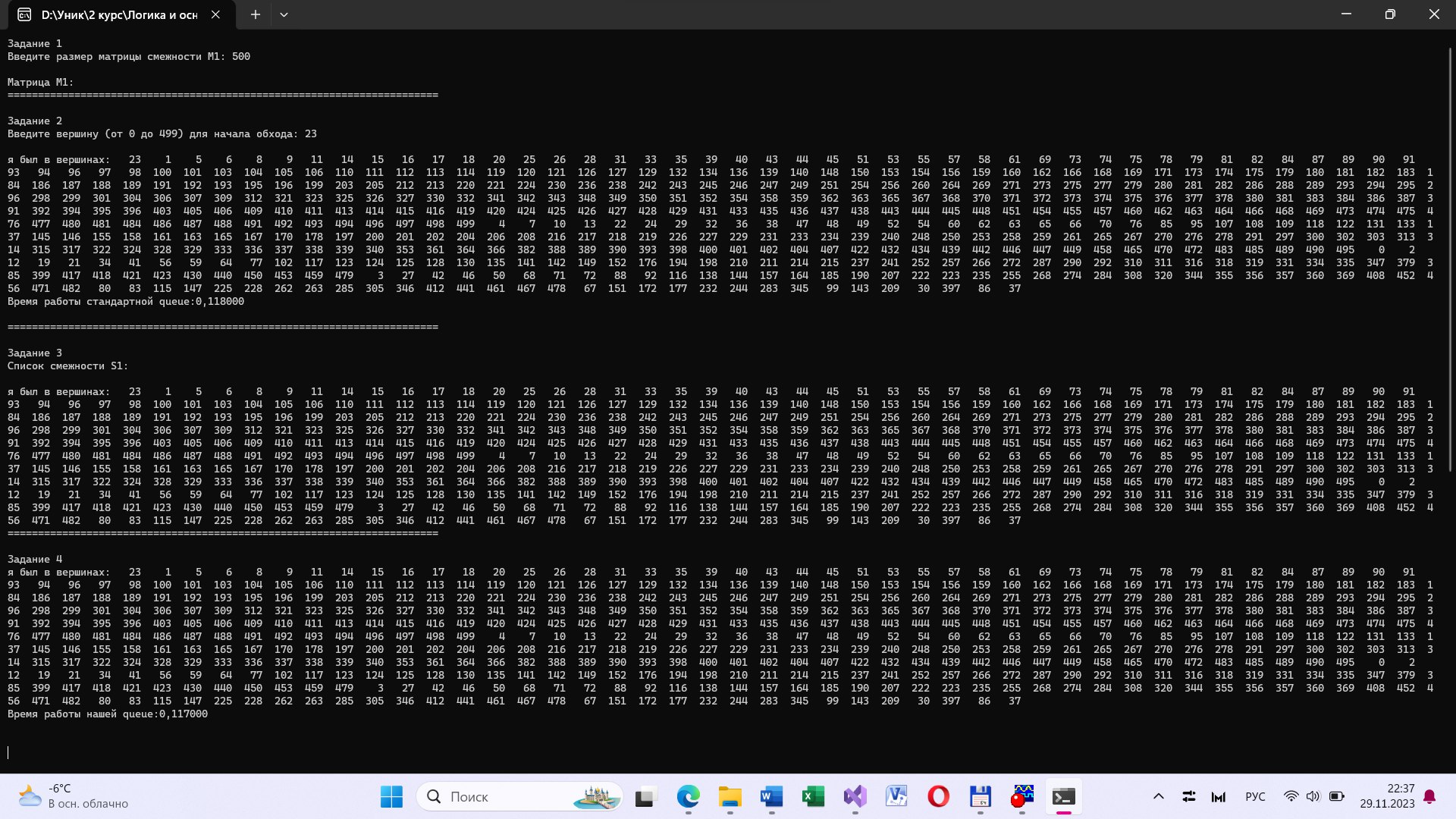


Рисунок 6 – Граф из 500 вершин

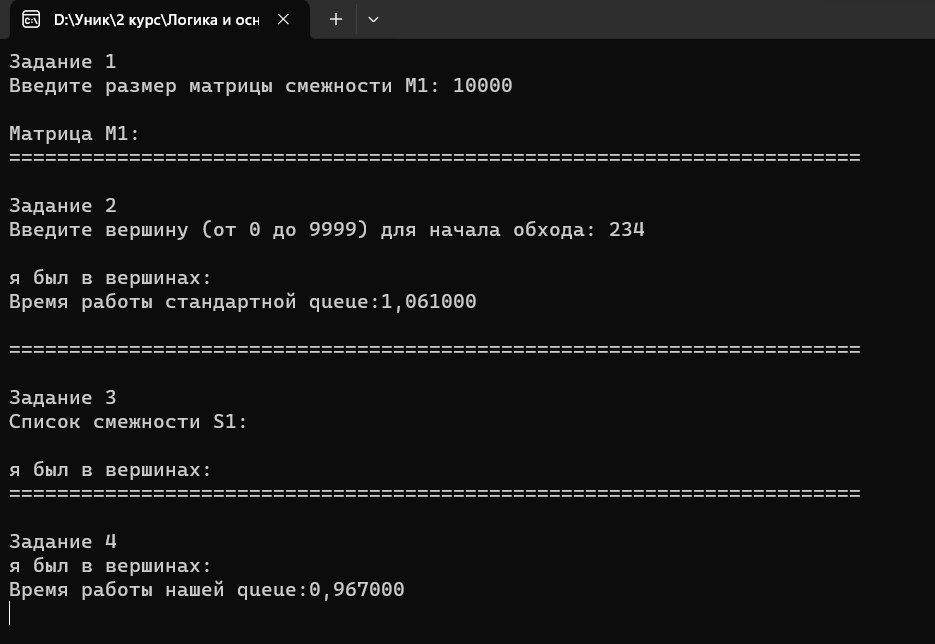


Рисунок 7 – Граф из 10000 вершин

|  |  |  |
| --- | --- | --- |
| **Количество вершин графа** | **Стандартная queue** | **Наша queue** |
| 10 | 0,002 | 0,001 |
| 50 | 0,008 | 0,007 |
| 100 | 0,016 | 0,015 |
| 200 | 0,054 | 0,054 |
| 500 | 0,118 | 0,117 |
| 10000 | 1,061 | 0,967 |

Таким образом, функция, использующая очередь, которую мы реализовали самостоятельно, работает быстрее, чем функция, использующая стандартный класс **queue**. Это связано с тем, что стандартная очередь **queue** работает с различными типами данных, а наша **queue** работает только с типом int.

**Вывод**

В ходе выполнения лабораторной работы была разработана программа, выполняющая обход графа в ширину. Результаты работы программы совпали с ожидаемыми результатами, следовательно, программа работает без ошибок.

Получили опыт в создании проектов в среде Microsoft Visual Studio, приобрели навыки программирования алгоритмов.