Министерство науки и высшего образования РФ

Пензенский государственный университет

Кафедра «Вычислительная кафедра»

**ОТЧЕТ**

по лабораторной работе №9

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Поиск расстояний в графе»

Выполнила:

студентка группы 22ВВВ2

Расторгуева К.В.

Приняли:

Акифьев И.В.

Митрохин М.А.

Пенза 2023

**Название**

Поиск расстояний в графе.

**Цель работы**

Реализация алгоритма поиска расстояний в графе на основе обходов в глубину и 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 поиска расстояний, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки С++.

**3.**\* Реализуйте процедуру поиска расстояний для графа, представленного списками смежности.

### Задание 2\*

1. Реализуйте процедуру поиска расстояний на основе обхода в глубину.
2. Реализуйте процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.
3. Оцените время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.

**Пояснительный текст к программе**

В алгоритме BFS создается пустая очередь, в которую помещается исходная вершина, из которой начат обход. Расстояние до этой вершины устанавливается равным 0.

Далее итерационно, пока очередь не опустеет, из нее извлекается первый элемент, который становится текущей вершиной. Затем в цикле просматривается v-я строка матрицы смежности графа G(v,i). Как только алгоритм встречает смежную с v не посещенную вершину , эта вершина помещается в очередь и для нее обновляется вектор расстояния . Расстояние до новой i-й вершины вычисляется как расстояние до текущей v-й вершины плюс 1 (так как ребра графа не взвешенные).

Назначение отдельных операторов программы указывается в виде комментариев. Программа завершает свою работу после нажатия на любую клавишу на клавиатуре.

**Результат выполнения программы**

Результат работы программы представлен на рис. 1-2.

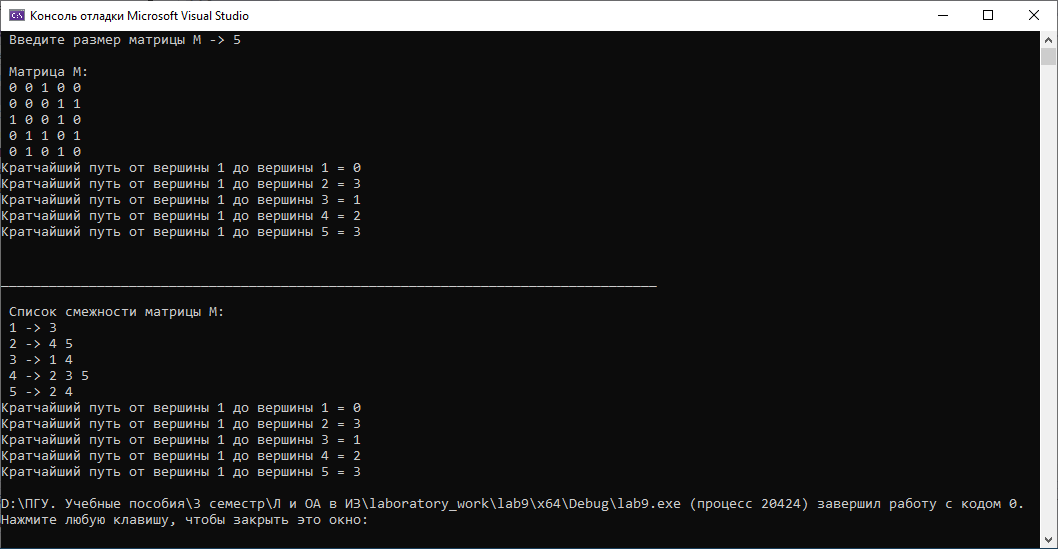


Рисунок 1 – алгоритм поиска расстояний в графе на основе обхода в ширину

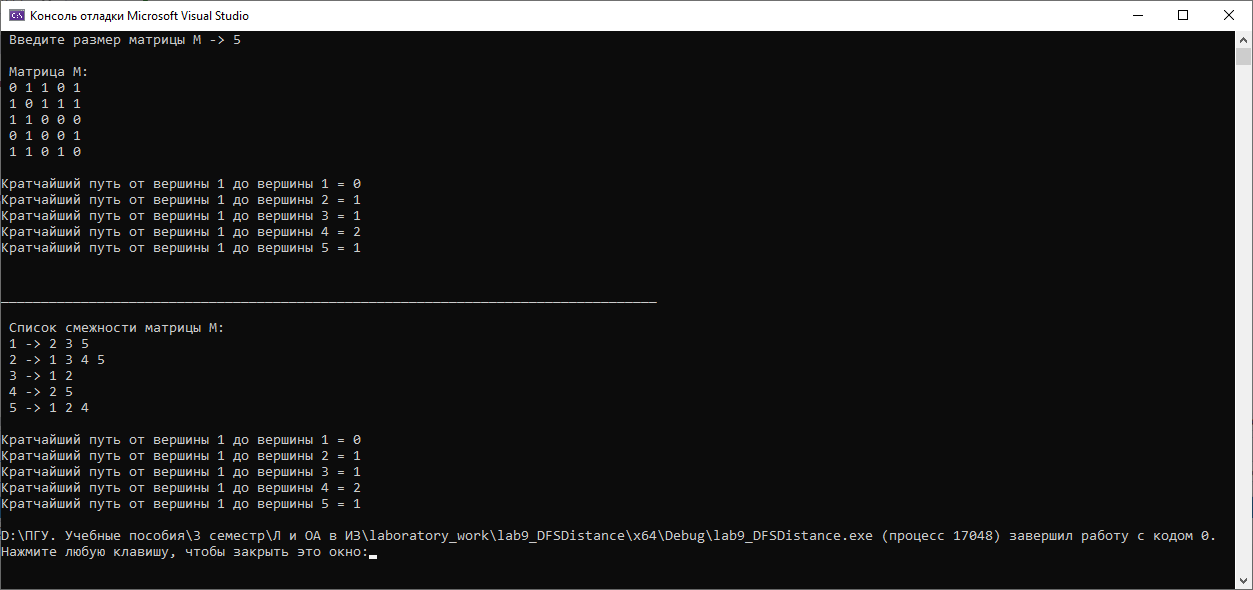
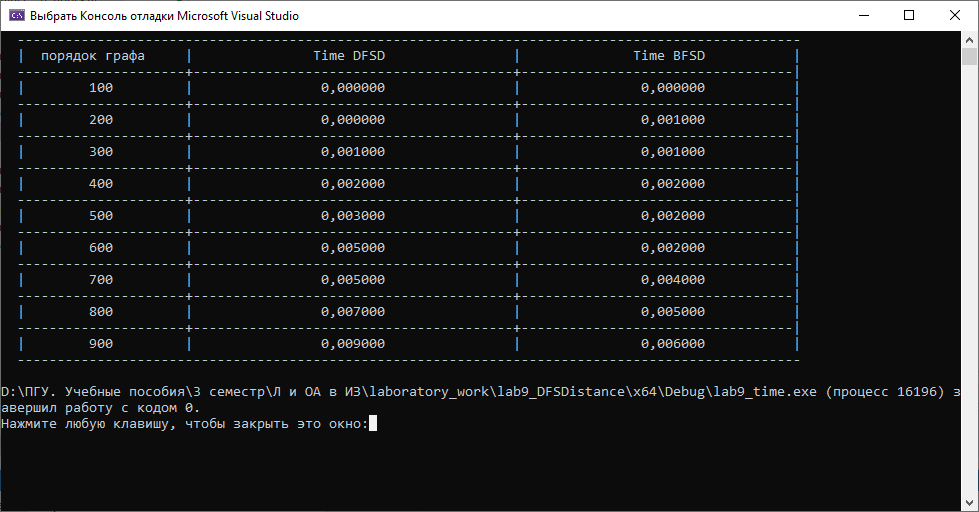
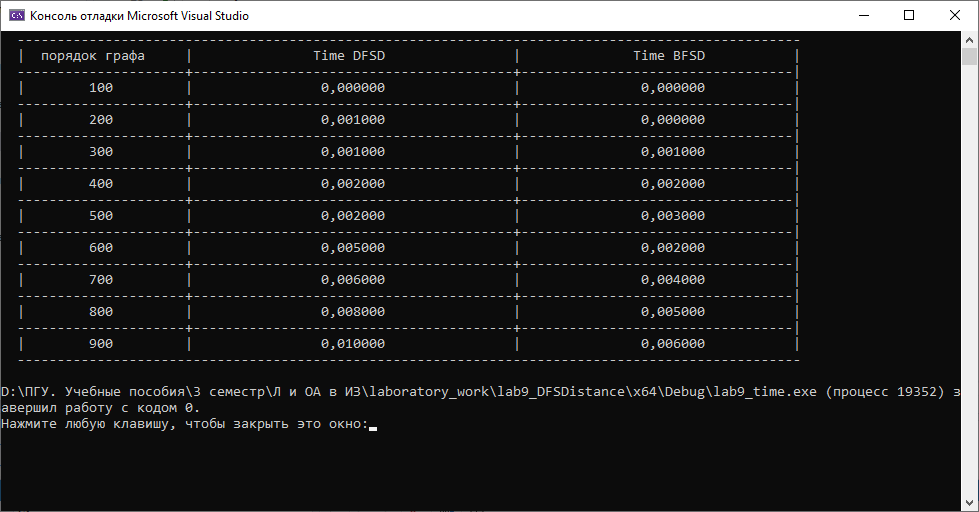
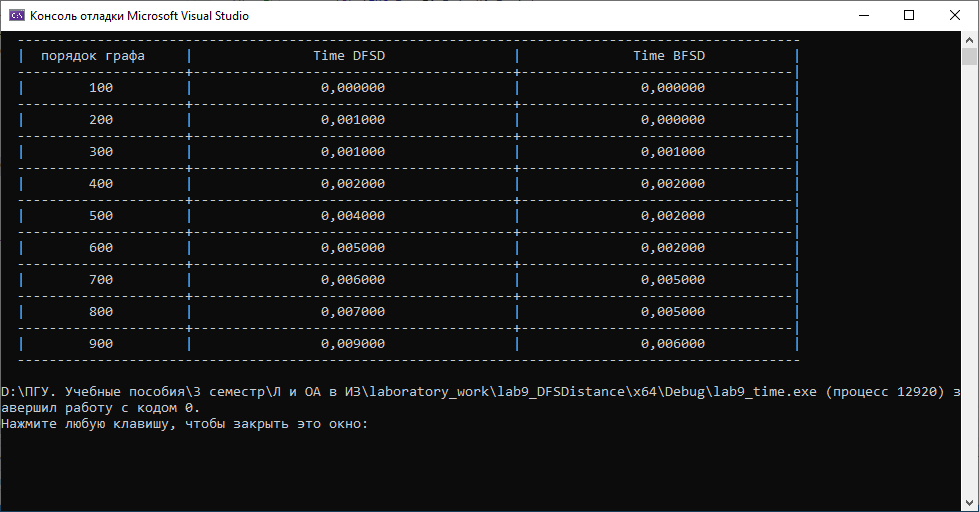


Рисунок 2 – алгоритм поиска расстояний в графе на основе обхода в глубину

**Оценка времени работы**







**Выводы**

В ходе выполнения лабораторной работы была разработана на языке Си программа, осуществляющая поиск кратчайших расстояний в графе на основе обходов в ширину и в глубину.

Также была проведена оценка времени работы алгоритмов поиска расстояний на основе обхода в глубину *DFSD* и обхода в ширину *BFSD* для графов разных порядков. Работа алгоритма поиска расстояний на основе обхода в ширину *BFSD* эффективнее, так как *BFS* - это метод обхода, при котором сначала проходим по всем узлам на одном уровне, прежде чем перейти к следующему уровню.

**Приложение**

**Алгоритм поиска расстояний в графе на основе обхода в ширину**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <locale.h>

#include <stdlib.h>

#include <time.h>

#include <queue>

#include <iostream>

struct node {

int top;

struct node\* next, \* next\_list;

};

struct node\* head = NULL, \* head\_top = NULL;

void BFSD(int v\_start, int n, int\*\* G, int\* dist) {

int v;

using namespace std;

queue <int> Q;

Q.push(v\_start);

dist[v\_start] = 0;

while (!Q.empty()) {

v = Q.front();

Q.pop();

for (int i = 1; i <= n; i++) {

if ((G[v][i] == 1) && (dist[v] + G[v][i]<dist[i])) {

Q.push(i);

dist[i] = dist[v] + G[v][i];

}

}

}

for (int i = 1; i <= n; i++) { //вывод результата

if (dist[i] == 1000) continue;

printf("Кратчайший путь от вершины %d до вершины %d = %d\n", v\_start, i, dist[i]);

}

return;

}

void BFSD\_list(struct node\* head\_save, struct node\* head\_top, int\* dist\_4\_list) {

int v;

struct node\* tmp\_top = NULL;

struct node\* start\_head = head\_top;

using namespace std;

queue <int> Q;

Q.push(start\_head->top);

dist\_4\_list[start\_head->top] = 0;

while (!Q.empty()) {

v = Q.front();

start\_head = head\_save;

while (start\_head != NULL) {

if (start\_head->top == v) break;

start\_head = start\_head->next;

}

Q.pop();

tmp\_top = start\_head->next\_list;

while (tmp\_top != NULL) {

if (dist\_4\_list[v] + 1 < dist\_4\_list[tmp\_top->top]) {

Q.push(tmp\_top->top);

dist\_4\_list[tmp\_top->top] = dist\_4\_list[v] + 1;

}

tmp\_top = tmp\_top->next;

}

}

start\_head = head\_save;

while (start\_head != NULL) { //вывод результата

if (dist\_4\_list[start\_head->top] == 1000) continue;

printf("Кратчайший путь от вершины %d до вершины %d = %d\n", head\_save->top, start\_head->top, dist\_4\_list[start\_head->top]);

start\_head = start\_head->next;

}

return;

}

struct node\* get\_struct(int\*\* G, int n) {

struct node\* p = NULL;

struct node\* tmp\_top = NULL;

struct node\* tmp = NULL;

struct node\* head\_top\_first = NULL;

int flag\_first = 0;

for (int i = 1; i <= n; i++) {

((p = (node\*)malloc(sizeof(struct node))) == NULL); // выделение памяти под новую вершину

p->top = i;

p->next = NULL;

if (flag\_first == 0) {

head\_top\_first = p;

head\_top = p;

flag\_first++;

tmp\_top = head\_top;

}

else {

tmp\_top->next = p;

tmp\_top = p;

}

}

tmp\_top->next = NULL;

for (int i = 1; i <= n; i++) {

flag\_first = 0;

for (int j = 1; j <= n; j++) {

if (G[i][j] == 1) {

((p = (node\*)malloc(sizeof(struct node))) == NULL); // выделение память под новый элемент вершины

p->top = j;

p->next = NULL;

if (flag\_first == 0) {

head = p;

flag\_first++;

tmp = head;

}

else {

tmp->next = p;

tmp = p;

}

}

}

head\_top->next\_list = head;

head\_top = head\_top->next;

head = NULL;

}

return head\_top\_first;

}

/\* Просмотр содержимого списка. \*/

void review(struct node\* head\_top) {

struct node\* tmp\_top = NULL;

if (head\_top == NULL) {

printf("Список пуст\n");

return;

}

while (head\_top != 0) {

printf(" %d -> ", head\_top->top);

tmp\_top = head\_top->next\_list;

while (tmp\_top != NULL) {

printf("%d ", tmp\_top->top);

tmp\_top = tmp\_top->next;

}

head\_top = head\_top->next;

printf("\n");

}

return;

}

int main() {

int n, v\_start;

int comp\_sv = 1;

setlocale(0, "rus");

printf(" Введите размер матрицы М -> ");

scanf("%d", &n);

printf("\n");

int\*\* G = (int\*\*)malloc(n \* sizeof(int\*));

for (int i = 1; i <= n; i++) {

G[i] = (int\*)malloc(n \* sizeof(int));

}

int\* dist = (int\*)malloc(n \* sizeof(int\*));

for (int i = 1; i <= n; i++) {

dist[i] = 1000;

}

int\* dist\_4\_list = (int\*)malloc(n \* sizeof(int\*));

for (int i = 1; i <= n; i++) {

dist\_4\_list[i] = 1000;

}

printf(" Матрица М:\n");

srand(time(0));

for (int i = 1; i <= n; i++) { //формирование матрицы смежности М

for (int j = 1; j <= n; j++) {

if (j >= i) {

if (i == j) G[i][j] = 0;

else {

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

G[j][i] = G[i][j];

}

}

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

}

printf("\n");

}

int flag\_sv = 0; //проверка графа на связность

for (int i = 1; i <= n; i++) {

flag\_sv = 0;

for (int j = 1; j <= n; j++) {

if (G[i][j] == 1) flag\_sv++;

}

if (flag\_sv == 0) comp\_sv++;

}

if (comp\_sv == 1) { //вывод результата

v\_start = 1;

BFSD(v\_start, n, G, dist);

printf("\n\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n");

printf(" Список смежности матрицы М:\n");

head = get\_struct(G, n);

review(head); //формирование списка смежности матрицы М

BFSD\_list(head, head, dist\_4\_list);

}

else printf("\n\n Граф не связный. Количество компонент связности = %d", comp\_sv);

return 0;

}

**Алгоритм поиска расстояний в графе на основе обхода в глубину**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <locale.h>

#include <stdlib.h>

#include <time.h>

struct node {

int top;

struct node\* next, \* next\_list;

};

struct node\* head = NULL, \* head\_top = NULL;

void DFSD(int v, int n, int\*\* G, int\* dist, int start\_dist) {

dist[v] = start\_dist;

for (int i = 1; i <= n; i++) {

if ((G[v][i] == 1) && (dist[v] + G[v][i] < dist[i])) {

dist[i] = dist[v] + G[v][i];

DFSD(i, n, G, dist, start\_dist + 1);

}

}

return;

}

void DFSD\_list(struct node\* head\_save, struct node\* head\_top, int\* dist\_4\_list, int start\_dist) {

struct node\* tmp\_head = NULL;

struct node\* tmp\_top = NULL;

dist\_4\_list[head\_top->top] = start\_dist;

tmp\_top = head\_top->next\_list;

while (tmp\_top != NULL) {

if (start\_dist + 1 < dist\_4\_list[tmp\_top->top]) { //dist\_4\_list[head\_top->top] + 1 < dist\_4\_list[tmp\_top->top]

dist\_4\_list[tmp\_top->top] = start\_dist + 1;

head\_top = head\_save;

while (head\_top != NULL) {

if (head\_top->top == tmp\_top->top) break;

else head\_top = head\_top->next;

}

tmp\_head = head\_top;

DFSD\_list(head\_save, tmp\_head, dist\_4\_list, start\_dist + 1);

}

tmp\_top = tmp\_top->next;

}

return;

}

struct node\* get\_struct(int\*\* G, int n) {

struct node\* p = NULL;

struct node\* tmp\_top = NULL;

struct node\* tmp = NULL;

struct node\* head\_top\_first = NULL;

int flag\_first = 0;

for (int i = 1; i <= n; i++) {

((p = (node\*)malloc(sizeof(struct node))) == NULL); // выделение памяти под новую вершину

p->top = i;

p->next = NULL;

if (flag\_first == 0) {

head\_top\_first = p;

head\_top = p;

flag\_first++;

tmp\_top = head\_top;

}

else {

tmp\_top->next = p;

tmp\_top = p;

}

}

tmp\_top->next = NULL;

for (int i = 1; i <= n; i++) {

flag\_first = 0;

for (int j = 1; j <= n; j++) {

if (G[i][j] == 1) {

((p = (node\*)malloc(sizeof(struct node))) == NULL); // выделение память под новый элемент вершины

p->top = j;

p->next = NULL;

if (flag\_first == 0) {

head = p;

flag\_first++;

tmp = head;

}

else {

tmp->next = p;

tmp = p;

}

}

}

head\_top->next\_list = head;

head\_top = head\_top->next;

head = NULL;

}

return head\_top\_first;

}

/\* Просмотр содержимого списка. \*/

void review(struct node\* head\_top) {

struct node\* tmp\_top = NULL;

if (head\_top == NULL) {

printf("Список пуст\n");

return;

}

while (head\_top != 0) {

printf(" %d -> ", head\_top->top);

tmp\_top = head\_top->next\_list;

while (tmp\_top != NULL) {

printf("%d ", tmp\_top->top);

tmp\_top = tmp\_top->next;

}

head\_top = head\_top->next;

printf("\n");

}

return;

}

int main() {

int n, v\_start;

int comp\_sv = 1;

struct node\* start\_head = NULL;

setlocale(0, "rus");

printf(" Введите размер матрицы М -> ");

scanf("%d", &n);

printf("\n");

int\*\* G = (int\*\*)malloc(n \* sizeof(int\*));

for (int i = 1; i <= n; i++) {

G[i] = (int\*)malloc(n \* sizeof(int));

}

int\* dist = (int\*)malloc(n \* sizeof(int\*));

for (int i = 1; i <= n; i++) {

dist[i] = 1000;

}

int\* dist\_4\_list = (int\*)malloc(n \* sizeof(int\*));

for (int i = 1; i <= n; i++) {

dist\_4\_list[i] = 1000;

}

printf(" Матрица М:\n");

srand(time(0));

for (int i = 1; i <= n; i++) { //формирование матрицы смежности М

for (int j = 1; j <= n; j++) {

if (j >= i) {

if (i == j) G[i][j] = 0;

else {

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

G[j][i] = G[i][j];

}

}

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

}

printf("\n");

}

int flag\_sv = 0; //проверка графа на связность

for (int i = 1; i <= n; i++) {

flag\_sv = 0;

for (int j = 1; j <= n; j++) {

if (G[i][j] == 1) flag\_sv++;

}

if (flag\_sv == 0) comp\_sv++;

}

if (comp\_sv == 1) { //вывод результата

v\_start = 1;

printf("\n");

DFSD(v\_start, n, G, dist, 0);

for (int i = 1; i <= n; i++) { //вывод результата

if (dist[i] == 1000) continue;

printf("Кратчайший путь от вершины %d до вершины %d = %d\n", v\_start, i, dist[i]);

}

printf("\n\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\n\n");

printf(" Список смежности матрицы М:\n");

head = get\_struct(G, n);

review(head); //формирование списка смежности матрицы М

printf("\n");

DFSD\_list(head, head, dist\_4\_list, 0);

start\_head = head;

while (start\_head != NULL) { //вывод результата

if (dist\_4\_list[start\_head->top] == 1000) continue;

printf("Кратчайший путь от вершины %d до вершины %d = %d\n", head->top, start\_head->top, dist\_4\_list[start\_head->top]);

start\_head = start\_head->next;

}

}

else printf("\n\n Граф не связный. Количество компонент связности = %d\n\n", comp\_sv);

return 0;

}

**Оценка времени**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <locale.h>

#include <stdlib.h>

#include <time.h>

#include <queue>

#include <stack>

#include <iostream>

void BFSD(int v\_start, int n, int\*\* G, int\* dist) {

int v;

using namespace std;

queue <int> Q;

Q.push(v\_start);

dist[v\_start] = 0;

while (!Q.empty()) {

v = Q.front();

Q.pop();

for (int i = 1; i <= n; i++) {

if ((G[v][i] == 1) && (dist[v] + G[v][i] < dist[i])) {

Q.push(i);

dist[i] = dist[v] + G[v][i];

}

}

}

return;

}

void DFSD(int v, int n, int\*\* G, int\* dist) {

v = 1;

using namespace std;

stack <int> s;

dist[v] = 0;

s.push(v);

while (!s.empty()) {

v = s.top();

s.pop();

for (int i = n; i > 0; i--) {

if ((G[v][i] == 1) && (dist[v] + 1 < dist[i])) {

s.push(i);

dist[i] = dist[v] + 1;

}

}

}

return;

}

int main() {

int n, v\_start;

int comp\_sv = 0;

float Time\_DFSD, Time\_BFSD;

setlocale(0, "rus");

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

printf(" | порядок графа | Time DFSD | Time BFSD | \n");

n = 100;

while (n < 1000) {

int\*\* G = (int\*\*)malloc(n \* sizeof(int\*));

for (int i = 1; i <= n; i++) {

G[i] = (int\*)malloc(n \* sizeof(int));

}

int\* dist = (int\*)malloc(n \* sizeof(int\*));

for (int i = 1; i <= n; i++) {

dist[i] = 1000;

}

srand(time(0));

for (int i = 1; i <= n; i++) { //формирование матрицы смежности М

for (int j = 1; j <= n; j++) {

if (j >= i) {

if (i == j) G[i][j] = 0;

else {

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

G[j][i] = G[i][j];

}

}

}

}

clock\_t start1 = clock();

v\_start = 1; //обход в глубину

DFSD(v\_start, n, G, dist);

clock\_t end1 = clock();

Time\_DFSD = ((double)(end1 - start1) / CLOCKS\_PER\_SEC);

for (int i = 1; i <= n; i++) {

dist[i] = 1000;

}

clock\_t start2 = clock();

v\_start = 1; //обход в ширину

BFSD(v\_start, n, G, dist);

clock\_t end2 = clock();

Time\_BFSD = ((double)(end2 - start2) / CLOCKS\_PER\_SEC);

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

printf(" | %d | %f | %f |\n", n, Time\_DFSD, Time\_BFSD);

n = n + 100;

}

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

return 0;

}