Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

Отчёт

*по лабораторной работе №5*

*По дисциплине: «Логика и основы алгоритмизации в инженерных задачах»*

*На тему: «Обход графа в ширину»*

**Выполнил студенты группы 19ВВ3:**

Ерёмин А.А

**Принял:**

Митрохин М.А

Пенза 2020

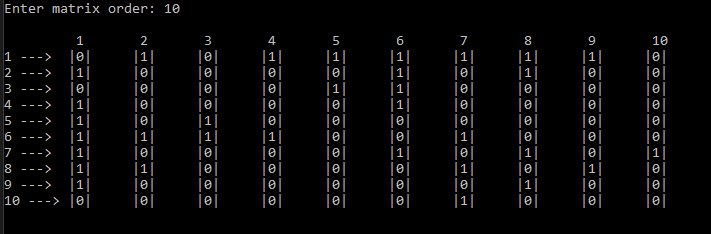
**Цель работы:** познакомиться с алгоритмом обхода графа в ширину. Реализовать данный алгоритм на матрицах и списках смежности.

Основная идея такого обхода в ширину состоит в том, чтобы посещать вершины по уровням удаленности от исходной вершины. Удалённость в данном случае понимается как количество ребер, по которым необходимо прейти до достижения вершины.

Алгоритм обхода в ширину продолжает осматривать уровень за уровнем, пока не пройдет все доступные вершины

**Задание 1.**

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



**Листинг(Задание 1.1):**

typedef struct Graph {

int\*\* Matrix;

int\* Vertexes;

int MatrixOrder;

};

int\*\* matrix\_malloc(int MatrixOrder) {

int\*\* Matrix = (int\*\*)malloc(sizeof(int) \* MatrixOrder);

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

Matrix[i] = (int\*)malloc(sizeof(int) \* MatrixOrder);

}

return Matrix;

}

int\* vertexes\_malloc(int MatrixOrder) {

int\* Vertexes = (int\*)malloc(sizeof(int) \* MatrixOrder);

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

Vertexes[i] = i + 1;

}

return Vertexes;

}

Graph\* graph\_create(int MatrixOrder) {

Graph\* pGraph = (Graph\*)malloc(sizeof(Graph));

pGraph->Matrix = matrix\_malloc(MatrixOrder);

pGraph->Vertexes = vertexes\_malloc(MatrixOrder);

pGraph->MatrixOrder = MatrixOrder;

return pGraph;

}

void graph\_free(Graph\* GraphG) {

int MatrixOrder = GraphG->MatrixOrder;

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

free(GraphG->Matrix[i]);

}

free(GraphG->Matrix);

free(GraphG->Vertexes);

free(GraphG);

}

int randFunc() {

if (rand() % 101 <= 30) {

return 1;

}

else {

return 0;

}

}

void graph\_random(Graph\* GraphG) {

int MatrixOrder = GraphG->MatrixOrder;

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

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

if (i == j) {

GraphG->Matrix[i][j] = 0;

}

else {

GraphG->Matrix[i][j] = GraphG->Matrix[j][i] = randFunc();

}

}

}

}

void matrix\_print(Graph\* GraphG) {

printf("\n");

int MatrixOrder = GraphG->MatrixOrder;

printf("\t");

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

printf(" %d\t", GraphG->Vertexes[i]);

}

printf("\n");

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

printf("%d --->\t", GraphG->Vertexes[i]);

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

printf("|%d|\t", GraphG->Matrix[i][j]);

}

printf("\n");

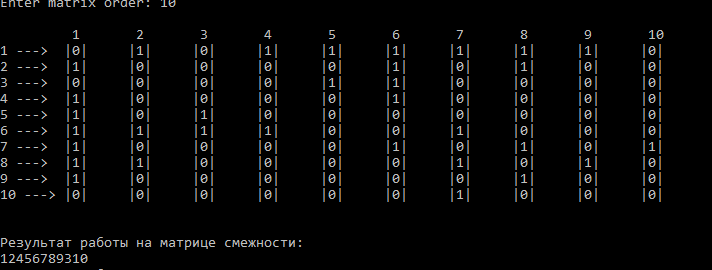
}

printf("\n");

}

**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++.

Обход начинается с вершины 1.



**Листинг (Задание 1.2):**

double BFS(Graph\* graph, int vertex, int\* visitedArray) {

clock\_t start, end;

std::queue <int> Q;

start = clock();

Q.push(vertex);

visitedArray[vertex] = 1;

while (!Q.empty()) {

vertex = Q.front();

printf("%d", vertex + 1);

Q.pop();

for (int i = 0; i < graph->MatrixOrder; i++) {

if (graph->Matrix[vertex][i] == 1 && visitedArray[i] == 0) {

Q.push(i);

visitedArray[i] = 1;

}

}

}

end = clock();

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

}

int\* create\_visitedArray(int MatrixOrder) {

int\* Array = (int\*)malloc(sizeof(int) \* MatrixOrder);

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

Array[i] = 0;

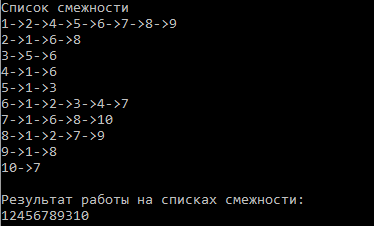
}

return Array;

}

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

Суть обхода на спискам смежности почти та же, как и при обходе матрицы.



**Листинг(Задание 1.3):**

typedef struct Node {

int vertex;

Node\* next;

};

typedef struct Lists {

int numVertexes;

Node\*\* head;

};

Node\* NodeCreate(int vertex) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

Lists\* ListsCreate(int numVertexes) {

Lists\* list = (Lists\*)malloc(sizeof(struct Lists));

list->numVertexes = numVertexes;

list->head = (Node \* \*)malloc(sizeof(struct Node\*) \* numVertexes);

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

list->head[i] = NULL;

}

return list;

}

void AddLast(Node\*\* head, Node\* newNode) {

Node\* current = \*head;

if (\*head == NULL) {

newNode->next = \*head;

\*head = newNode;

return;

}

while (current->next) {

current = current->next;

}

newNode->next = current->next;

current->next = newNode;

}

void TransMatrixInList(Graph\* graph, Lists\* list) {

int MatrixOrder = graph->MatrixOrder;

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

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

if (graph->Matrix[i][j] == 1) {

Node\* newNode = NodeCreate(j);

AddLast(&list->head[i], newNode);

}

}

}

}

void PrintList(Lists\* list) {

printf("\n");

printf("\nСписок смежности");

printf("\n");

Node\* current;

for (int i = 0; i < list->numVertexes; i++) {

printf("%d", i + 1);

current = list->head[i];

while (current != NULL) {

printf("->%d", current->vertex + 1);

current = current->next;

}

printf("\n");

}

}

void BFSinLists(Lists\* list, int vertex, int\* visitedArray) {

std::queue <int> Q;

Q.push(vertex);

visitedArray[vertex] = 1;

while (!Q.empty()) {

vertex = Q.front();

printf("%d", vertex + 1);

Q.pop();

Node\* current = list->head[vertex];

while (current) {

if (visitedArray[current->vertex] == 0) {

Q.push(current->vertex);

visitedArray[current->vertex] = 1;

}

current = current->next;

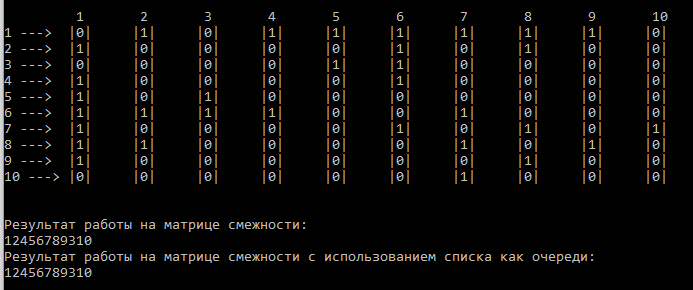
}

}

}

**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



Вместо стандартной библиотеки <queue> С++, мной был создан список, который представляет собой подобие очереди.

**Листинг(Задание 2.1):**

void AddLastQ(int vertex);

void PopNodeQ();

Node\* head = NULL;

Node\* tail = NULL;

void AddLastQ(int vertex) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->vertex = vertex;

newNode->next = NULL;

if (head == NULL) {

head = newNode;

tail = newNode;

return;

}

tail->next = newNode;

tail = newNode;

}

void PopNodeQ() {

Node\* headOld = head;

head = head->next;

free(headOld);

}

double BFSwithoutQ(Graph\* graph, int vertex, int\* visitedArray) {

clock\_t start, end;

start = clock();

AddLastQ(vertex);

visitedArray[vertex] = 1;

while (head != NULL) {

vertex = head->vertex;

printf("%d", vertex + 1);

PopNodeQ();

for (int i = 0; i < graph->MatrixOrder; i++) {

if (graph->Matrix[vertex][i] == 1 && visitedArray[i] == 0) {

AddLastQ(i);

visitedArray[i] = 1;

}

}

}

end = clock();

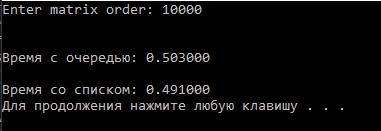
return (double)(end - start) / CLOCKS\_PER\_SEC;

}

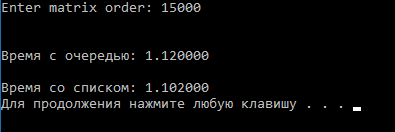
**2.** Оцените время работы двух реализаций алгоритмов обхода в ширину (использующего стандартный класс **queue** и использующего очередь, реализованную самостоятельно) для графов разных порядков.

**Убедившись в правильности работы алгоритма, стоит убрать вывод пройденных вершин, из-за достаточного большого их количества.**

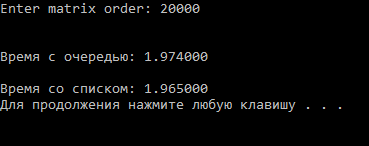
10000.



15000.



20000.



Таким образом, алгоритм использующий собственные списки немного быстрее(правда, скорее в пределах погрешности). Также, можно сделать вывод, что суть написанного списка, очень похожа на библиотечную.

**Листинг (Задание 2.2):**

double BFSwithoutQ(Graph\* graph, int vertex, int\* visitedArray) {

clock\_t start, end;

start = clock();

AddLastQ(vertex);

visitedArray[vertex] = 1;

while (head != NULL) {

vertex = head->vertex;

PopNodeQ();

for (int i = 0; i < graph->MatrixOrder; i++) {

if (graph->Matrix[vertex][i] == 1 && visitedArray[i] == 0) {

AddLastQ(i);

visitedArray[i] = 1;

}

}

}

end = clock();

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

}

double BFS(Graph\* graph, int vertex, int\* visitedArray) {

clock\_t start, end;

std::queue <int> Q;

start = clock();

Q.push(vertex);

visitedArray[vertex] = 1;

while (!Q.empty()) {

vertex = Q.front();

Q.pop();

for (int i = 0; i < graph->MatrixOrder; i++) {

if (graph->Matrix[vertex][i] == 1 && visitedArray[i] == 0) {

Q.push(i);

visitedArray[i] = 1;

}

}

}

end = clock();

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

}

TimeQue = BFS(graph, 0, visitedArray);

TimeList = BFSwithoutQ(graph, 0, visitedArray);

**Вывод:** в ходе выполнения лабораторных заданий ознакомился с алгоритмом обхода графа в ширину, а также реализовал его на матрицах и списках смежности на языке Си.

**Листинг(All):**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Algorithms.h"

typedef struct Graph {

int\*\* Matrix;

int\* Vertexes;

int MatrixOrder;

};

typedef struct Node {

int vertex;

Node\* next;

};

typedef struct Lists {

int numVertexes;

Node\*\* head;

};

Node\* NodeCreate(int vertex);

void AddLastQ(int vertex);

void PopNodeQ();

Node\* head = NULL;

Node\* tail = NULL;

int\*\* matrix\_malloc(int MatrixOrder) {

int\*\* Matrix = (int\*\*)malloc(sizeof(int) \* MatrixOrder);

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

Matrix[i] = (int\*)malloc(sizeof(int) \* MatrixOrder);

}

return Matrix;

}

int\* vertexes\_malloc(int MatrixOrder) {

int\* Vertexes = (int\*)malloc(sizeof(int) \* MatrixOrder);

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

Vertexes[i] = i + 1;

}

return Vertexes;

}

Graph\* graph\_create(int MatrixOrder) {

Graph\* pGraph = (Graph\*)malloc(sizeof(Graph));

pGraph->Matrix = matrix\_malloc(MatrixOrder);

pGraph->Vertexes = vertexes\_malloc(MatrixOrder);

pGraph->MatrixOrder = MatrixOrder;

return pGraph;

}

int\* create\_visitedArray(int MatrixOrder) {

int\* Array = (int\*)malloc(sizeof(int) \* MatrixOrder);

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

Array[i] = 0;

}

return Array;

}

void graph\_free(Graph\* GraphG) {

int MatrixOrder = GraphG->MatrixOrder;

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

free(GraphG->Matrix[i]);

}

free(GraphG->Matrix);

free(GraphG->Vertexes);

free(GraphG);

}

int randFunc() {

if (rand() % 101 <= 30) {

return 1;

}

else {

return 0;

}

}

void graph\_random(Graph\* GraphG) {

int MatrixOrder = GraphG->MatrixOrder;

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

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

if (i == j) {

GraphG->Matrix[i][j] = 0;

}

else {

GraphG->Matrix[i][j] = GraphG->Matrix[j][i] = randFunc();

}

}

}

}

void clear\_visited(int\* visitedArray, int size) {

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

visitedArray[i] = 0;

}

}

void matrix\_print(Graph\* GraphG) {

printf("\n");

int MatrixOrder = GraphG->MatrixOrder;

printf("\t");

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

printf(" %d\t", GraphG->Vertexes[i]);

}

printf("\n");

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

printf("%d --->\t", GraphG->Vertexes[i]);

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

printf("|%d|\t", GraphG->Matrix[i][j]);

}

printf("\n");

}

printf("\n");

}

Node\* NodeCreate(int vertex) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->vertex = vertex;

newNode->next = NULL;

return newNode;

}

Lists\* ListsCreate(int numVertexes) {

Lists\* list = (Lists\*)malloc(sizeof(struct Lists));

list->numVertexes = numVertexes;

list->head = (Node \* \*)malloc(sizeof(struct Node\*) \* numVertexes);

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

list->head[i] = NULL;

}

return list;

}

void AddLastQ(int vertex) {

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->vertex = vertex;

newNode->next = NULL;

if (head == NULL) {

head = newNode;

tail = newNode;

return;

}

tail->next = newNode;

tail = newNode;

}

void AddLast(Node\*\* head, Node\* newNode) {

Node\* current = \*head;

if (\*head == NULL) {

newNode->next = \*head;

\*head = newNode;

return;

}

while (current->next) {

current = current->next;

}

newNode->next = current->next;

current->next = newNode;

}

void PrintList(Lists\* list) {

printf("\n");

printf("\nСписок смежности");

printf("\n");

Node\* current;

for (int i = 0; i < list->numVertexes; i++) {

printf("%d", i + 1);

current = list->head[i];

while (current != NULL) {

printf("->%d", current->vertex + 1);

current = current->next;

}

printf("\n");

}

}

void TransMatrixInList(Graph\* graph, Lists\* list) {

int MatrixOrder = graph->MatrixOrder;

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

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

if (graph->Matrix[i][j] == 1) {

Node\* newNode = NodeCreate(j);

AddLast(&list->head[i], newNode);

}

}

}

}

void PopNodeQ() {

Node\* headOld = head;

head = head->next;

free(headOld);

}

double BFSwithoutQ(Graph\* graph, int vertex, int\* visitedArray) {

clock\_t start, end;

start = clock();

AddLastQ(vertex);

visitedArray[vertex] = 1;

while (head != NULL) {

vertex = head->vertex;

PopNodeQ();

for (int i = 0; i < graph->MatrixOrder; i++) {

if (graph->Matrix[vertex][i] == 1 && visitedArray[i] == 0) {

AddLastQ(i);

visitedArray[i] = 1;

}

}

}

end = clock();

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

}

double BFS(Graph\* graph, int vertex, int\* visitedArray) {

clock\_t start, end;

std::queue <int> Q;

start = clock();

Q.push(vertex);

visitedArray[vertex] = 1;

while (!Q.empty()) {

vertex = Q.front();

Q.pop();

for (int i = 0; i < graph->MatrixOrder; i++) {

if (graph->Matrix[vertex][i] == 1 && visitedArray[i] == 0) {

Q.push(i);

visitedArray[i] = 1;

}

}

}

end = clock();

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

}

void BFSinLists(Lists\* list, int vertex, int\* visitedArray) {

std::queue <int> Q;

Q.push(vertex);

visitedArray[vertex] = 1;

while (!Q.empty()) {

vertex = Q.front();

printf("%d", vertex + 1);

Q.pop();

Node\* current = list->head[vertex];

while (current) {

if (visitedArray[current->vertex] == 0) {

Q.push(current->vertex);

visitedArray[current->vertex] = 1;

}

current = current->next;

}

}

}

void ListsDelete(Lists\* list) {

for (int i = 0; i < list->numVertexes; i++) {

while (list->head[i] != NULL) {

Node\* prevNode = list->head[i];

list->head[i] = list->head[i]->next;

free(prevNode);

}

}

free(list->head);

free(list);

}

int main(void) {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

srand(time(NULL));

int MatrixOrder;

double TimeQue = 0, TimeList = 0;

printf("Enter matrix order: ");

scanf("%d", &MatrixOrder);

Graph\* graph = graph\_create(MatrixOrder);

int\* visitedArray = create\_visitedArray(MatrixOrder);

graph\_random(graph);

matrix\_print(graph);

printf("\nРезультат работы на матрице смежности: \n");

TimeQue = BFS(graph, 0, visitedArray);

clear\_visited(visitedArray, MatrixOrder);

printf("\nРезультат работы на матрице смежности с использованием списка как очереди: \n");

TimeList = BFSwithoutQ(graph, 0, visitedArray);

printf("\n\n");

printf("Время с очередью: %lf", TimeQue);

printf("\n\n");

printf("Время со списком: %lf\n", TimeList);

Lists\* list = ListsCreate(MatrixOrder);

TransMatrixInList(graph, list);

PrintList(list);

printf("\nРезультат работы на списках смежности: \n");

clear\_visited(visitedArray, MatrixOrder);

BFSinLists(list, 0, visitedArray);

printf("\n");

graph\_free(graph);

ListsDelete(list);

system("Pause");

}