Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

Отчёт

*по лабораторной работе №4*

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

*На тему: «Обход графа в глубину»*

**Выполнил студенты группы 19ВВ3:**

Ерёмин А.А

**Принял:**

Митрохин М.А

Пенза 2020

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

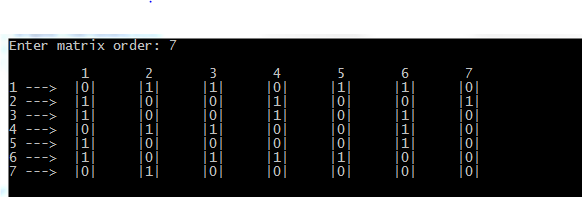
Обход графа – одна из наиболее распространенных операций с графами. Задачей обхода является прохождение всех вершин в графе. Обходы применяются для поиска информации, хранящейся в узлах графа, нахождения связей между вершинами или группами вершин и т.д

Одним из способов обхода графов является поиск в глубину. Идея такого обхода состоит в том, чтобы начав обход из какой-либо вершины всегда переходить по первой встречающейся в процессе обхода связи в следующую вершину, пока существует такая возможно. Как только такая возможность исчерпается вернутся на шаг назад и найти следующий вариант прохождения.

**Задание 1.**

**Задание 1.1**

Сгенерируйте две матрицы смежности для неориентированного графа G. Выведите сгенерированные матрицы на экран.



**Листинг(Задание 1.1) .**

int\*\* matrix\_malloc(int MatrixOrder) { // Выделение памяти под матрицу.

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) { // Выделение памяти под номера вершин

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) { // Создание структуры графа

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

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

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

pGraph->MatrixOrder = MatrixOrder;

return pGraph;

}

int randFunc() {

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

return 1;

}

else {

return 0;

}

}

void graph\_random(Graph\* GraphG) { // Функция случайного заполнения матрицы смежности

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");

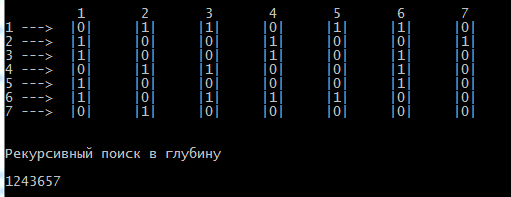
}

printf("\n");

}

**Задание 1.2**

Для сгенерированного графа осуществите процедуру обхода в глубину.



Начальная вершина 1, смежная с вершинами 2, 3, 5, 6.

В соответствии с алгоритмом, при прохождении по первой строке матрицы, первым встретится ребро с вершиной 2, алгоритм переходит к первой связанной вершине(2), которая не посещена, помечает её как посещенную и начинает обход с этой вершины. В вершине 2 первой не посещенной смежной вершиной является вершина 4. Алгоритм помечает вершину 4 и продолжает обход с неё. Дальше все происходит по этой же схеме. Если возможность прохода по какой-то вершины заканчивается, алгоритм возвращается к 1 вершине и, если следующая смежная с 1 вершина не была посещена(в данном случае 3), алгоритм продолжает свою работу с неё.

Как видно из скриншота, в результате обхода в глубину мы прошли по всем вершинам графа и вывели результат обхода в консоль.

**Листинг(Задание 1.2)**

void DFS(int Vertex, Graph\* graph, int\* visitedArray) {

printf("%d", Vertex + 1); // Вывели посещенную вершину

visitedArray[Vertex] = 1; // Пометили как посещенную

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

if (graph->Matrix[Vertex][vertexNext] && visitedArray[vertexNext] != 1) {

DFS(vertexNext, graph, visitedArray);

}

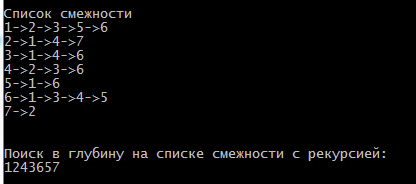
}

}

Конкретно этот алгоритм выполняется рекурсивно, при встрече с не посещенной смежной вершиной, снова вызывается функция обхода, только уже в качестве начальной вершины обхода передаётся не посещенная.

**Задание 1.3**

Реализуйте процедуру обхода в глубину для графа, представленного списками смежности.



Сутью алгоритма обхода в глубину для списка смежности является похожая с матрицей схема, только вместо прохода по определенной строке матрицы, происходит проход по односвязному списку, начало которого берётся из массива с индексом заданной изначально вершины.

**Листинг(Задание 1.3)**

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

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

visitedArray[vertex] = 1;

Node\* current = list->head[vertex]; // Начинаем с первого узла, на который указывает элемент массива с индексом заданной вершины(который является вершиной).

while (current) { // Просматриваем все смежные вершины, пока не найдём не посещенную

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

DFSinLists(list, current->vertex, visitedArray);

}

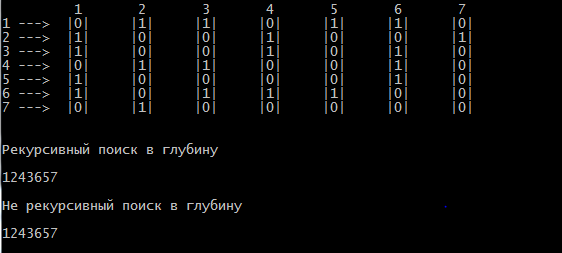
current = current->next;

}

}

**Задание 2\***

Для матричной формы представления графов выполните преобразование рекурсивной реализации обхода графа к не рекурсивной.



Рекурсивный алгоритм обхода в глубину очень прост в реализации, однако у него есть существенный недостаток, рано или поздно, при большом количестве вершин и связей между ними, может произойти переполнение стека.

Эту проблему решает не рекурсивный алгоритм обхода в глубину, строящийся на основе собственно выделенного для хранения вершин стека.

Вместо того, чтобы снова и снова вызывать функцию обхода, не посещенные вершины кладутся в стек.

**Листинг(Задание 2\*)**

void DFSnR(int Vertex, Graph\* graph, int\* visitedArray) {

std::stack<int>myStack;

myStack.push(Vertex); // Пушим первую вершину в стек

while (!myStack.empty()) {

Vertex = myStack.top(); // Используем номер вершины, по которой будет осуществляться обход из вершины стека.

myStack.pop(); // Достаём вершину из стека

if (visitedArray[Vertex] == 0) { // Проверяем, была ли вершина посещена.

visitedArray[Vertex] = 1; // Помечаем, если нет.

printf("%d", Vertex + 1); // Выводим на экран.

for (int i = graph->MatrixOrder - 1; i >= 0; i--) { // Обходим строку матрицы в обратном порядке(для получения такого же порядка как и при рекурсивном обходе)

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

myStack.push(i); // Если есть связь и вершина не посещена, пушим в стек.

}

}

}

}

}

**Вывод: В ходе выполнения лабораторных заданий ознакомился с алгоритмом обхода графа в глубину, реализовал данный алгоритм на матрице и списке смежности. Преобразовал рекурсивную реализацию алгоритма к не рекурсивной.**

**Листинг(All).**

#include "stdafx.h"

#include "Algorithms.h"

#include "Structs.h"

#include <stack>

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");

}

void DFS(int Vertex, Graph\* graph, int\* visitedArray) {

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

visitedArray[Vertex] = 1;

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

if (graph->Matrix[Vertex][vertexNext] && visitedArray[vertexNext] != 1) {

DFS(vertexNext, graph, visitedArray);

}

}

}

void DFSnR(int Vertex, Graph\* graph, int\* visitedArray) {

std::stack<int>myStack;

myStack.push(Vertex);

while (!myStack.empty()) {

Vertex = myStack.top();

myStack.pop();

if (visitedArray[Vertex] == 0) {

visitedArray[Vertex] = 1;

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

for (int i = graph->MatrixOrder - 1; i >= 0; i--) {

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

myStack.push(i);

}

}

}

}

}

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 AddFirst(Node\*\* head, Node\* newNode) {

newNode->next = \*head;

\*head = newNode;

}

void AddAfter(Node\* afterNode, Node\* newNode) {

newNode->next = afterNode->next;

afterNode->next = newNode;

}

void AddBefore(Node\*\* head, Node\* beforeNode, Node\* newNode) {

Node\* current = \*head;

if (\*head == beforeNode) {

AddFirst(head, newNode);

return;

}

while (current && current->next != beforeNode)

current = current->next;

if (current)

AddAfter(current, newNode);

}

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

Node\* current = \*head;

if (\*head == NULL) {

AddFirst(head, newNode);

return;

}

while (current->next)

current = current->next;

AddAfter(current, newNode);

}

void PrintList(Lists\* list) {

printf("\n");

printf("Список 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 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[i]);

}

free(list->head);

free(list);

}

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

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

visitedArray[vertex] = 1;

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

while (current) {

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

DFSinLists(list, current->vertex, visitedArray);

}

current = current->next;

}

}

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

std::stack<int>stackVertexes;

stackVertexes.push(vertex);

while (!stackVertexes.empty()) {

vertex = stackVertexes.top();

stackVertexes.pop();

if (visitedArray[vertex] == 0) {

visitedArray[vertex] = 1;

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

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

while (current) {

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

stackVertexes.push(current->vertex);

}

current = current->next;

}

}

}

}

void TransMatrixInListRev(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);

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

}

}

}

}

int main(void) {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

srand(time(NULL));

int MatrixOrder;

printf("Enter matrix order: ");

scanf("%d", &MatrixOrder);

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

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

graph\_random(graph);

matrix\_print(graph);

printf("\nРекурсивный поиск в глубину\n");

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

if (visitedArray[vertex] == 0) {

printf("\n");

DFS(vertex, graph, visitedArray);

}

}

clear\_visited(visitedArray, MatrixOrder);

printf("\n\nНе рекурсивный поиск в глубину\n");

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

if (visitedArray[vertex] == 0) {

printf("\n");

DFSnR(vertex, graph, visitedArray);

}

}

printf("\n\n");

clear\_visited(visitedArray, MatrixOrder);

Lists\* list = ListsCreate(MatrixOrder);

TransMatrixInList(graph, list);

PrintList(list);

printf("\n\nПоиск в глубину на списке смежности с рекурсией: ");

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

if (visitedArray[vertex] == 0) {

printf("\n");

DFSinLists(list, vertex, visitedArray);

}

}

printf("\n");

clear\_visited(visitedArray, MatrixOrder);

Lists\* listReverse = ListsCreate(MatrixOrder);

TransMatrixInListRev(graph, listReverse);

PrintList(listReverse);

printf("\n\nПоиск в глубину на списке смежности без рекурсией: ");

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

if (visitedArray[vertex] == 0) {

printf("\n");

DFSinListsNR(listReverse, vertex, visitedArray);

}

}

clear\_visited(visitedArray, MatrixOrder);

PrintList(list);

printf("\n\nПоиск в глубину на списке смежности без рекурсией: ");

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

if (visitedArray[vertex] == 0) {

printf("\n");

DFSinListsNR(list, vertex, visitedArray);

}

}

printf("\n");

graph\_free(graph);

ListsDelete(list);

ListsDelete(listReverse);

printf("\n");

system("Pause");

}

**«Struct.h»**

#pragma once

typedef struct Graph {

int\*\* Matrix;

int\* Vertexes;

int MatrixOrder;

};

typedef struct Node {

int vertex;

Node\* next;

};

typedef struct Lists {

int numVertexes;

struct Node\*\* head;

};