Министерство образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

Отчёт

*по лабораторной работе №1*

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

*На тему: «Поиск расстояний в графе»*

**Выполнил студенты группы 19ВВ3:**

Ерёмин А.А

**Принял:**

Митрохин М.А

Пенза 2020

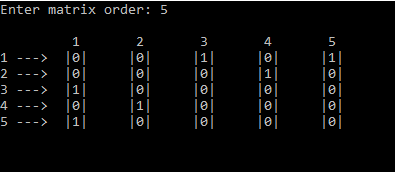
**Цель:** реализовать алгоритм поиска расстояний в графе.

**Ход работы:**

**Задание 1.**

**1.** Сгенерируйте (используя генератор случайных чисел) матрицу

смежности для неориентированного графа G. Выведите матрицу на экран.

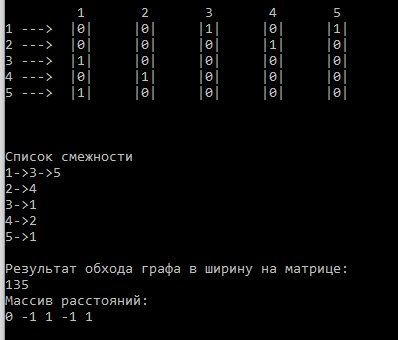


**2.** Для сгенерированного графа осуществите процедуру поиска

расстояний, реализованную в соответствии с приведенным выше описанием.

При реализации алгоритма в качестве очереди используйте класс queue из

стандартной библиотеки С++.



**Листинг(1.2):**

void BFSD(Graph\* graph ,int\* dist, int start\_vertex) {

std::queue<int>Q;

int num\_vertexes = graph->matrix\_order;

Q.push(start\_vertex);

dist[start\_vertex] = 0;

int current\_vertex;

while (!Q.empty()) {

current\_vertex = Q.front();

Q.pop();

printf("%d", current\_vertex + 1);

for (int i = 0; i < num\_vertexes; i++) {

if (graph->matrix[current\_vertex][i] && dist[i] == -1) {

Q.push(i);

dist[i] = dist[current\_vertex] + 1;

}

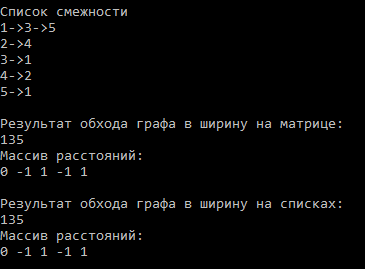
}

}

}

Суть алгоритма проста. Используется стандартный обход в ширину, с тем условий, что массив посещенных вершин становится массивом расстояний (в котором не посещённая вершина помечена как -1, 0 – расстояние вершины до самой себя, >0 расстояние до других вершин). А действие пометки не посещенной вершины заменяется на сумму – расстояние до предыдущей вершины + 1.

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



**Листинг (1.3):**

void BFSDLists(Lists\* list, int\* dist, int start\_vertex) {

std::queue<int>Q;

int num\_vertexes = list->num\_vertexes;

Q.push(start\_vertex);

dist[start\_vertex] = 0;

int current\_vertex;

while (!Q.empty()) {

current\_vertex = Q.front();

Q.pop();

printf("%d", current\_vertex + 1);

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

while(currentNode) {

if (dist[currentNode->vertex] == -1) {

Q.push(currentNode->vertex);

dist[currentNode->vertex] = dist[current\_vertex] + 1;

}

currentNode = currentNode->next;

}

}

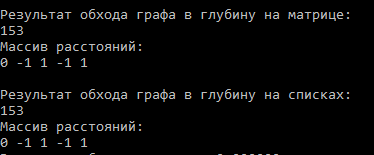
}

Разница лишь в условии прохода по графу.

**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.



**Листинг (2.1):**

void DFSD(Graph\* graph, int\* dist, int start\_vertex) {

std::stack<int>S;

int num\_vertexes = graph->matrix\_order;

S.push(start\_vertex);

dist[start\_vertex] = 0;

int current\_vertex;

while (!S.empty()) {

current\_vertex = S.top();

S.pop();

printf("%d", current\_vertex + 1);

for (int i = 0; i < num\_vertexes; i++) {

if (graph->matrix[current\_vertex][i] && dist[i] == -1) {

S.push(i);

dist[i] = dist[current\_vertex] + 1;

}

}

}

}

Суть такая же, только обход в ширину заменяется обходом в глубину. Однако, стоит учитывать важную деталь – если в вершину есть несколько путей разные по дальности, то алгоритм выведет тот путь, который встретит первым. Алгоритм можно немного видоизменить и тогда он будет осуществлять поиск кратчайших расстояний. (Для этого достаточно заменить условие dist[i] == -1 на dist[i] > dist[v] + 1)

**Листинг (2.2):**

void DFSDLists(Lists\* list, int\* dist, int start\_vertex) {

std::stack<int>S;

int num\_vertexes = list->num\_vertexes;

S.push(start\_vertex);

dist[start\_vertex] = 0;

int current\_vertex;

while (!S.empty()) {

current\_vertex = S.top();

S.pop();

printf("%d", current\_vertex + 1);

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

while(currentNode) {

if (dist[currentNode->vertex] == -1) {

S.push(currentNode->vertex);

dist[currentNode->vertex] = dist[current\_vertex] + 1;

}

currentNode = currentNode->next;

}

}

}

**3.** Оцените время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.

Graph order = 1000



2000



3000



4000



5000



Исходя из полученных данных, можно сделать вывод, что алгоритм поиска расстояний на основе обхода в глубину все же немного медленнее.

**Вывод:** в ходе выполнения лабораторных заданий были поняты и реализованы алгоритмы поиска расстояний в графе на основе двух обходов, оценена скорость их работы.

**Листинг (all):**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "Algorithms.h"

typedef struct Graph {

int\*\* matrix;

int\* vertexes;

int matrix\_order;

};

typedef struct Node {

int vertex;

Node\* next;

};

typedef struct Lists {

int num\_vertexes;

Node\*\* head;

};

int\*\* MatrixMalloc(int matrix\_order) {

int\*\* Matrix = (int\*\*)malloc(sizeof(int) \* matrix\_order);

for (int i = 0; i < matrix\_order; i++) {

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

}

return Matrix;

}

int\* VertexesMalloc(int matrix\_order) {

int\* Vertexes = (int\*)malloc(sizeof(int) \* matrix\_order);

for (int i = 0; i < matrix\_order; i++) {

Vertexes[i] = i + 1;

}

return Vertexes;

}

Graph\* GraphCreate(int matrix\_order) {

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

pGraph->matrix = MatrixMalloc(matrix\_order);

pGraph->vertexes = VertexesMalloc(matrix\_order);

pGraph->matrix\_order = matrix\_order;

return pGraph;

}

int\* ArrayCreate(int num\_vertexes) {

int\* Array = (int\*)malloc(sizeof(int) \* num\_vertexes);

return Array;

}

void DistanceArrayRefresh(int\* distance\_array, int num\_vertexes) {

for (int i = 0; i < num\_vertexes; i++) {

distance\_array[i] = -1;

}

}

void VisitedArrayRefresh(int\* visited\_array, int num\_vertexes) {

for (int i = 0; i < num\_vertexes; i++) {

visited\_array[i] = 0;

}

}

void GraphFree(Graph\* GraphG) {

int MatrixOrder = GraphG->matrix\_order;

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

free(GraphG->matrix[i]);

}

free(GraphG->matrix);

free(GraphG->vertexes);

free(GraphG);

}

int \_Rand() {

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

return 1;

}

else {

return 0;

}

}

void GraphRandom(Graph\* GraphG) {

int matrix\_order = GraphG->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = i; j < matrix\_order; j++) {

if (i == j) {

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

}

else {

GraphG->matrix[i][j] = GraphG->matrix[j][i] = \_Rand();

}

}

}

}

void MatrixPrint(Graph\* GraphG) {

printf("\n");

int matrix\_order = GraphG->matrix\_order;

printf("\t");

for (int i = 0; i < matrix\_order; i++) {

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

}

printf("\n");

for (int i = 0; i < matrix\_order; i++) {

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

for (int j = 0; j < matrix\_order; 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 num\_vertexes) {

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

list->num\_vertexes = num\_vertexes;

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

for (int i = 0; i < num\_vertexes; i++) {

list->head[i] = NULL;

}

return list;

}

void ListsDelete(Lists\* list) {

for (int i = 0; i < list->num\_vertexes; 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);

}

void AddFirst(Lists\* list, Node\* newNode, int vertex) {

newNode->next = list->head[vertex];

list->head[vertex] = newNode;

}

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

newNode->next = prevNode->next;

prevNode->next = newNode;

}

void AddLast(Lists\* list, Node\* newNode, int vertex) {

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

if (current == NULL) {

AddFirst(list, newNode, vertex);

return;

}

while (current->next) {

current = current->next;

}

AddAfter(current, newNode);

}

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

int matrix\_order = graph->matrix\_order;

for (int i = 0; i < matrix\_order; i++) {

for (int j = 0; j < matrix\_order; j++) {

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

Node\* newNode = NodeCreate(j);

AddLast(list, newNode, i);

}

}

}

}

void ListsPrint(Lists\* list) {

printf("\n");

printf("\nСписок смежности");

printf("\n");

Node\* current;

for (int i = 0; i < list->num\_vertexes; i++) {

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

current = list->head[i];

while (current != NULL) {

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

current = current->next;

}

printf("\n");

}

}

void ArrayPrint(int\* Array, int num\_vertexes) {

printf("\n");

for (int i = 0; i < num\_vertexes; i++) {

printf("%d ", Array[i]);

}

}

void BFSD(Graph\* graph ,int\* dist, int start\_vertex) {

std::queue<int>Q;

int num\_vertexes = graph->matrix\_order;

Q.push(start\_vertex);

dist[start\_vertex] = 0;

int current\_vertex;

while (!Q.empty()) {

current\_vertex = Q.front();

Q.pop();

printf("%d", current\_vertex + 1);

for (int i = 0; i < num\_vertexes; i++) {

if (graph->matrix[current\_vertex][i] && dist[i] == -1) {

Q.push(i);

dist[i] = dist[current\_vertex] + 1;

}

}

}

}

void BFSDLists(Lists\* list, int\* dist, int start\_vertex) {

std::queue<int>Q;

int num\_vertexes = list->num\_vertexes;

Q.push(start\_vertex);

dist[start\_vertex] = 0;

int current\_vertex;

while (!Q.empty()) {

current\_vertex = Q.front();

Q.pop();

printf("%d", current\_vertex + 1);

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

while(currentNode) {

if (dist[currentNode->vertex] == -1) {

Q.push(currentNode->vertex);

dist[currentNode->vertex] = dist[current\_vertex] + 1;

}

currentNode = currentNode->next;

}

}

}

void DFSD(Graph\* graph, int\* dist, int start\_vertex) {

std::stack<int>S;

int num\_vertexes = graph->matrix\_order;

S.push(start\_vertex);

dist[start\_vertex] = 0;

int current\_vertex;

while (!S.empty()) {

current\_vertex = S.top();

S.pop();

printf("%d", current\_vertex + 1);

for (int i = 0; i < num\_vertexes; i++) {

if (graph->matrix[current\_vertex][i] && dist[i] == -1) {

S.push(i);

dist[i] = dist[current\_vertex] + 1;

}

}

}

}

void DFSDLists(Lists\* list, int\* dist, int start\_vertex) {

std::stack<int>S;

int num\_vertexes = list->num\_vertexes;

S.push(start\_vertex);

dist[start\_vertex] = 0;

int current\_vertex;

while (!S.empty()) {

current\_vertex = S.top();

S.pop();

printf("%d", current\_vertex + 1);

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

while(currentNode) {

if (dist[currentNode->vertex] == -1) {

S.push(currentNode->vertex);

dist[currentNode->vertex] = dist[current\_vertex] + 1;

}

currentNode = currentNode->next;

}

}

}

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

srand(time(NULL));

int matrix\_order;

double start[2];

double end[2];

printf("Enter matrix order: ");

scanf("%d", &matrix\_order);

Graph\* graph = GraphCreate(matrix\_order);

GraphRandom(graph);

MatrixPrint(graph);

Lists\* list = ListsCreate(matrix\_order);

TransMatrixInList(graph, list);

ListsPrint(list);

int\* distance\_array = ArrayCreate(matrix\_order);

DistanceArrayRefresh(distance\_array, matrix\_order);

printf("\nРезультат обхода графа в ширину на матрице:\n");

start[0] = clock();

BFSD(graph, distance\_array, 0);

end[0] = clock();

printf("\nМассив расстояний:");

ArrayPrint(distance\_array, matrix\_order);

DistanceArrayRefresh(distance\_array, matrix\_order);

printf("\n\nРезультат обхода графа в ширину на списках:\n");

BFSDLists(list, distance\_array, 0);

printf("\nМассив расстояний:");

ArrayPrint(distance\_array, matrix\_order);

DistanceArrayRefresh(distance\_array, matrix\_order);

printf("\n\nРезультат обхода графа в глубину на матрице:\n");

start[1] = clock();

DFSD(graph, distance\_array, 0);

end[1] = clock();

printf("\nМассив расстояний:");

ArrayPrint(distance\_array, matrix\_order);

DistanceArrayRefresh(distance\_array, matrix\_order);

printf("\n\nРезультат обхода графа в глубину на списках:\n");

DFSDLists(list, distance\_array, 0);

printf("\nМассив расстояний:");

ArrayPrint(distance\_array, matrix\_order);

printf("\nВремя при обходе в ширину: ");

printf("%lf", (end[0] - start[0]) / CLOCKS\_PER\_SEC);

printf("\nВремя при обходе в глубину: ");

printf("%lf", (end[1] - start[1]) / CLOCKS\_PER\_SEC);

printf("\n\n");

system("PAUSE");

}