Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра “Вычислительная техника”

**ОТЧЕТ**

по лабораторной работе №8

по дисциплине «Логика и основы алгоритмизации в инженерных задачах»

# на тему: «Обход графа в ширину»

Выполнили студенты группы 22ВВВ3:

Куракин Н.Н.

Майоров Н.А.

Матюшин К. М.

Приняли:

Юрова О.В.

Акифьев И.В.

Пенза 2023

**Название**

Обход графа в ширину

**Цель работы**

Научиться использовать «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.

**Задание 1:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include <stdbool.h>

#include <locale.h>

#include <queue>

struct node {

int numNode;

struct node\* nextNode;

};

int\* CreateGraph(int vertexes) {

int\* Mtrx = (int\*)malloc(sizeof(int) \* vertexes \* vertexes);

if (!Mtrx) return 0;

for (int i = 0; i != vertexes; i++) {

\*(Mtrx + i \* vertexes + i) = 0;

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

\*(Mtrx + i \* vertexes + j) = rand() % 2;

\*(Mtrx + j \* vertexes + i) = \*(Mtrx + i \* vertexes + j);

}

}

return Mtrx;

}

int ModifyGraph(struct node\* list, int\* graph, int vertexes) {

if (!list || !graph) return 0;

for (int i = 0; i != vertexes; i++) {

struct node\* current = list + i;

current->numNode = i + 1;

for (int j = 0; j != vertexes; j++) {

if (\*(graph + i \* vertexes + j)) {

current->nextNode = (struct node\*)malloc(sizeof(struct node));

current = current->nextNode;

current->numNode = j + 1;

}

}

current->nextNode = NULL;

}

return 1;

}

void breadthFirstSearchAdjacencyMatrix(int\* matrix, int startNode, int numNodes) {

int\* visited = (int\*)malloc(sizeof(int) \* numNodes);

if (!visited || numNodes <= 0) return;

std::queue<int>queue;

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

visited[i] = 0;

}

visited[startNode] = 1;

queue.push(startNode);

while (!queue.empty()) {

int currentNode = queue.front();

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

queue.pop();

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

if (\*(matrix + currentNode \* numNodes + i) && !visited[i]) {

visited[i] = 1;

queue.push(i);

}

}

}

}

void breadthFirstSearchAdjacencyList(struct node\* adjacencyList, int startNode, int numNodes) {

int\* visited = (int\*)malloc(sizeof(int) \* numNodes);

if (!visited || numNodes<=0) return;

std::queue<int>queue;

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

visited[i] = 0;

}

visited[startNode] = 1;

queue.push(startNode);

while (!queue.empty()) {

int currentNode = queue.front();

struct node\* current = adjacencyList + currentNode;

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

queue.pop();

do {

if (!visited[(current)->numNode - 1]) {

visited[(current)->numNode - 1] = 1;

queue.push((current)->numNode - 1);

}

current = current->nextNode;

} while (current);

}

}

void ShowGraph(int\* graph, int vertexes) {

if (!graph) return;

for (int i = 0; i != vertexes; i++) {

for (int j = 0; j != vertexes; j++) {

printf("%i ", \*(graph + i \* vertexes + j));

}

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

printf("\n");

}

}

void ShowList(struct node\* list, int vertexes) {

if (!list) return;

printf("\n");

for (int i = 0; i != vertexes; i++) {

struct node\* current = list + i;

//printf("|%i\t|->", current->numNode);

while (current) {

printf("|%i|\t->", current->numNode);

current = current->nextNode;

}

printf("\n");

}

}

int main() {

srand(time(NULL));

setlocale(LC\_ALL, "RUS");

int vertexNum = 0, vertexStart = 0;

printf("Колво вершин: ");

scanf("%d", &vertexNum);

printf("Начальная точка: ");

if (!scanf("%d", &vertexStart) || vertexStart > vertexNum || vertexStart < 0) {

printf("Нет такой вершины");

return 0;

}

int\* adjacencyMatrix = CreateGraph(vertexNum);

ShowGraph(adjacencyMatrix, vertexNum);

struct node\* adjacencyList = (struct node\*)malloc(sizeof(struct node) \* vertexNum);

breadthFirstSearchAdjacencyMatrix(adjacencyMatrix, vertexStart - 1, vertexNum);

printf("\n");

if (!ModifyGraph(adjacencyList, adjacencyMatrix, vertexNum)) return 1;

ShowList(adjacencyList, vertexNum);

breadthFirstSearchAdjacencyList(adjacencyList, vertexStart - 1, vertexNum);

return 0;

}

**Задание 2:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include <stdbool.h>

#include <locale.h>

#include <queue>

struct node {

int numNode;

struct node\* nextNode;

};

struct QueueNode {

int data;

struct QueueNode\* next;

};

struct Queue {

struct QueueNode\* front, \* rear;

};

int\* CreateGraph(int vertexes) {

int\* Mtrx = (int\*)malloc(sizeof(int) \* vertexes \* vertexes);

if (!Mtrx) return 0;

for (int i = 0; i != vertexes; i++) {

\*(Mtrx + i \* vertexes + i) = 0;

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

\*(Mtrx + i \* vertexes + j) = rand() % 2;

\*(Mtrx + j \* vertexes + i) = \*(Mtrx + i \* vertexes + j);

}

}

return Mtrx;

}

void breadthFirstSearchAdjacencyMatrix(int\* matrix, int startNode, int numNodes) {

int\* visited = (int\*)malloc(sizeof(int) \* numNodes);

if (!visited) return;

std::queue<int>queue;

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

visited[i] = 0;

}

visited[startNode] = 1;

queue.push(startNode);

while (!queue.empty()) {

int currentNode = queue.front();

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

queue.pop();

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

if (\*(matrix + currentNode \* numNodes + i) && !visited[i]) {

visited[i] = 1;

queue.push(i);

}

}

}

}

void ShowGraph(int\* graph, int vertexes) {

if (!graph) return;

for (int i = 0; i != vertexes; i++) {

for (int j = 0; j != vertexes; j++) {

printf("%i ", \*(graph + i \* vertexes + j));

}

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

printf("\n");

}

}

struct QueueNode\* newQueueNode(int data) {

struct QueueNode\* temp = (struct QueueNode\*)malloc(sizeof(struct QueueNode));

temp->data = data;

temp->next = NULL;

return temp;

}

struct Queue\* createQueue() {

struct Queue\* q = (struct Queue\*)malloc(sizeof(struct Queue));

q->front = q->rear = NULL;

return q;

}

bool isEmpty(struct Queue\* q) {

return (q->front == NULL);

}

void enQueue(struct Queue\* q, int data) {

struct QueueNode\* temp = newQueueNode(data);

if (q->rear == NULL) {

q->front = q->rear = temp;

return;

}

q->rear->next = temp;

q->rear = temp;

}

void deQueue(struct Queue\* q) {

if (isEmpty(q)) {

return;

}

struct QueueNode\* temp = q->front;

q->front = q->front->next;

if (q->front == NULL) {

q->rear = NULL;

}

free(temp);

}

void breadthFirstSearchCustomQueue(int\* graph, int numNodes, int startNode) {

bool\* visited = (bool\*)malloc(numNodes \* sizeof(bool)); // Массивдляотслеживанияпосещенныхвершин

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

visited[i] = false; // Инициализируем все вершины как непосещенные

}

struct Queue\* q = createQueue(); // Инициализируем очередь

visited[startNode] = true; // Помечаем начальную вершину как посещенную

enQueue(q, startNode); // Добавляем начальную вершину в очередь

while (!isEmpty(q)) {

int currentNode = q->front->data;

printf("%d ", currentNode + 1); // Выводим вершину

deQueue(q); // Удаляем вершину из очереди

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

if (\*(graph + currentNode \* numNodes + i) && !visited[i]) {

visited[i] = true; // Помечаем вершину как посещенную

enQueue(q, i); // Добавляем вершину в очередь для дальнейшего обхода

}

}

}

free(visited);

}

int main() {

srand(time(NULL));

setlocale(LC\_ALL, "RUS");

int vertexNum = 0;

printf("Колво вершин: ");

scanf("%d", &vertexNum);

int\* adjacencyMatrix = CreateGraph(vertexNum);

ShowGraph(adjacencyMatrix, vertexNum);

clock\_t start, end;

double cpu\_time\_used;

start = clock();

breadthFirstSearchAdjacencyMatrix(adjacencyMatrix, 0, vertexNum);

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Время выполнения обхода в ширину с использованием стандартной очереди: %f секунд\n", cpu\_time\_used);

start = clock();

// Выполняем обход в ширину с использованием очереди, реализованной самостоятельно

breadthFirstSearchCustomQueue(adjacencyMatrix, vertexNum, 0);

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

printf("Время выполнения обхода в ширину с использованием самостоятельно реализованной очереди: %f секунд\n", cpu\_time\_used);

return 0;

}

|  |  |  |
| --- | --- | --- |
| Кол-во вершин | Время обхода графа в ширину с помощью библиотечной очереди | Время обхода в ширину с помощью нашего метода |
| 6000^2 | 0,242 | 0,221 |
| 8000^2 | 0,424 | 0,397 |
| 10000^2 | 0,643 | 0,609 |
| 12000^2 | 0,921 | 0,874 |
| 14000^2 | 1,338 | 1,2 |

Вывод к заданию 2: Наша программа работает быстрее.

Вывод: Мы изучили и научились пользоваться методом «Обхода графа в ширину»