Министерство науки и высшего образования Российской Федерации  
Пензенский государственный университет  
Кафедра вычислительная техника

**ОТЧËТ**  
по лабораторной работе №9  
по курсу «Логика и основы алгоритмизации в инженерных задачах»  
на тему «Поиск расстояний в графе»

Выполнили студенты группы 22ВВВ2:  
Хоссейни Нежад С. А. С. М.

Приняли:  
Акифьев И. В.  
Митрохин М. А.

Пенза 2023

**Название**

Поиск расстояний в графе

**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.

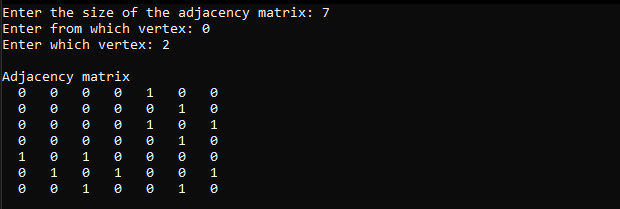
**Задание 2\***

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.

**Ход работы**

**Задание 1**

1. Сгенерировал (используя генератор случайных чисел) матрицу смежности для неориентированного графа G. Вывел матрицу на экран.



1. Для сгенерированного графа осуществил процедуру поиска расстояний. При  реализации алгоритма в качестве очереди использовал класс queue из стандартной библиотеки С++.



1. **\***Реализовал процедуру поиска расстояний для графа, представленного списками смежности.





**Задание 2\***

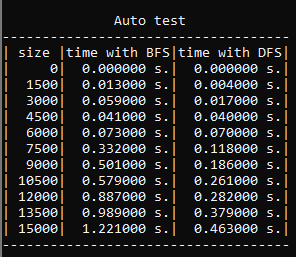
1. Реализовал процедуру поиска расстояний на основе обхода в глубину.



1. Реализовал процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.



1. Оценил время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.



На всех размерах по времени быстрее всего работает реализация поиска расстояния с обходом в глубину.

**Листинг**

* Header.h

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

void fill\_arr(int\*\* Arr, int size);

void print\_arr(int\*\* Arr, int size);

int find\_arr(int\* Arr, int size, int element);

struct node\* createNode(int v);

struct Graph\* createGraph(int vertices);

void addEdge(struct Graph\* graph, int src, int dest);

void printGraph(struct Graph\* graph);

struct Graph\* transformGraph(int\*\* Arr, int size);

int BfsList(struct Graph\* graph, int size, int vertex, int end);

int Bfs(int\*\* Matrix, int size, int vertex, int end, int shutdown);

void autoTest(int startVertex, int endVertex);

int\* fill\_visited(int\* visited, int size);

int DFS(int\*\* Matrix, int size, int start, int end, int shutdown);

void Dfs(int\*\* Matrix, int size, int vertex, int\* visited, int end, int\* c, int shutdown);

int DFSList(struct Graph\* graph, int size, int start, int end);

void DfsList(struct Graph\* graph, int vertex, int\* visited, int end, int\* c);

* Lab9.cpp

#include "Header.h"

#include <queue>

#include <iostream>

using namespace std;

int main(void)

{

srand(time(NULL));

int sizeMatrix, start = 0, end = 0;

int\*\* Arr;

printf("Enter the size of the adjacency matrix: ");

scanf("%d", &sizeMatrix);

printf("Enter from which vertex: ");

scanf("%d", &start);

printf("Enter which vertex: ");

scanf("%d", &end);

Arr = (int\*\*)malloc(sizeMatrix \* sizeof(int\*));

for (int i = 0; i < sizeMatrix; ++i)

{

Arr[i] = (int\*)malloc(sizeMatrix \* sizeof(int));

}

printf("\nAdjacency matrix\n");

fill\_arr(Arr, sizeMatrix);

print\_arr(Arr, sizeMatrix);

struct Graph\* graph = transformGraph(Arr, sizeMatrix);

printGraph(graph);

printf("\nBFS in the adjacency matrix\n");

printf("\ndistance: %d ", Bfs(Arr, sizeMatrix, start, end, 0));

printf("\n\nBFS in the adjacency list\n");

printf("\ndistance: %d ", BfsList(graph, sizeMatrix, start, end));

printf("\n\n\nDFS in the adjacency matrix\n");

printf("way: ");

printf("\ndistance: %d ", DFS(Arr, sizeMatrix, start, end, 0));

printf("\n\nDFS in the adjacency list\n");

printf("way: ");

printf("\ndistance: %d \n", DFSList(graph, sizeMatrix, start, end));

for (int i = 0; i < sizeMatrix; i++)

free(Arr[i]);

free(Arr);

autoTest(start, end);

return 0;

}

struct node

{

int vertex;

struct node\* next;

};

struct Graph

{

int numVertices;

struct node\*\* adjLists;

};

int Bfs(int\*\* Matrix, int size, int vertex, int end, int shutdown)

{

std::queue<int> queue;

queue.push(vertex);

int\* visited = (int\*)calloc(size, size \* sizeof(int));

visited = fill\_visited(visited, size);

visited[vertex] = 0;

if (!shutdown)

printf("way: ");

while (!queue.empty())

{

vertex = queue.front();

queue.pop();

if (!shutdown)

printf("%d ", vertex);

if (vertex == end)

break;

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

{

if (Matrix[vertex][i] == 1 && visited[i] == -1)

{

queue.push(i);

visited[i] = visited[vertex] + 1;

}

}

}

int data = visited[end];

if(size != 0)

free(visited);

return data;

}

int BfsList(struct Graph\* graph, int size, int vertex, int end)

{

std::queue<int> queue;

queue.push(vertex);

int\* visited = (int\*)calloc(size, size \* sizeof(int));

visited = fill\_visited(visited, size);

visited[vertex] = 0;

printf("way: ");

while (!queue.empty())

{

vertex = queue.front();

queue.pop();

printf("%d ", vertex);

if (vertex == end)

break;

struct node\* temp = graph->adjLists[vertex];

while (temp)

{

int i = temp->vertex;

if (visited[i] == -1)

{

queue.push(i);

visited[i] = visited[vertex] + 1;

}

temp = temp->next;

}

}

int data = visited[end];

free(visited);

return data;

}

int DFS(int\*\* Matrix, int size, int start, int end, int shutdown)

{

int\* visited = (int\*)calloc(size, size \* sizeof(int));

int count = 0;

Dfs(Matrix, size, 0, visited, end, &count, shutdown);

free(visited);

return count - 1;

}

void Dfs(int\*\* Matrix, int size, int vertex, int\* visited, int end, int\* c, int shutdown)

{

int count = \*c;

if (visited[end] != 0)

return;

visited[vertex] = 1;

if (!shutdown)

printf("%d ", vertex);

\*c = count + 1;

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

{

if (Matrix[vertex][i] == 1 && visited[i] == 0)

{

Dfs(Matrix, size, i, visited, end, c, shutdown);

}

}

}

int DFSList(struct Graph\* graph, int size, int start, int end)

{

int\* visited = (int\*)calloc(size, size \* sizeof(int));

int count = 0;

DfsList(graph, start, visited, end, &count);

free(visited);

return count - 1;

}

void DfsList(struct Graph\* graph, int vertex, int\* visited, int end, int\* c)

{

int count = \*c;

if (visited[end] != 0)

return;

visited[vertex] = 1;

printf("%d ", vertex);

\*c = count + 1;

struct node\* temp = graph->adjLists[vertex];

while (temp)

{

int i = temp->vertex;

if (visited[i] == 0)

DfsList(graph, i, visited, end, c);

temp = temp->next;

}

}

void print\_arr(int\*\* Arr, int size)

{

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

{

for (int j = 0; j < size; j++)

{

printf("%3d ", Arr[i][j]);

}

printf("\n");

}

}

void fill\_arr(int\*\* Arr, int size)

{

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

{

for (int j = 0; j < size; j++)

{

if (i == j)

{

Arr[i][j] = 0;

continue;

}

if (rand() % 3 == 1)

{

Arr[i][j] = 1;

Arr[j][i] = 1;

}

else

{

Arr[i][j] = 0;

Arr[j][i] = 0;

}

}

}

}

int\* fill\_visited(int\* visited, int size)

{

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

{

visited[i] = -1;

}

return visited;

}

void autoTest(int startVertex, int endVertex)

{

printf("\n\n Auto test");

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

printf("| size |time with BFS|time with DFS|\n");

for (int size = 0; size <= 15000; size += 1500)

{

printf("|%6d|", size);

int\*\* Arr = (int\*\*)malloc(size \* sizeof(int\*));

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

{

Arr[i] = (int\*)malloc(size \* sizeof(int));

}

fill\_arr(Arr, size);

clock\_t start = clock();

int a = Bfs(Arr, size, startVertex, endVertex, 1);

clock\_t end = clock();

double time\_spent = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("%10f s.", time\_spent);

start = clock();

a = DFS(Arr, size, startVertex, endVertex, 1);

end = clock();

time\_spent = (double)(end - start) / CLOCKS\_PER\_SEC;

printf("|%10f s.|\n", time\_spent);

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

free(Arr[i]);

free(Arr);

}

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

}

struct node\* createNode(int v)

{

struct node\* newNode = (struct node\*)malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices)

{

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (struct node\*\*)malloc(vertices \* sizeof(struct node\*));

for (int i = 0; i < vertices; i++)

graph->adjLists[i] = NULL;

return graph;

}

struct Graph\* transformGraph(int\*\* Arr, int size)

{

struct Graph\* graph = createGraph(size);

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

{

for (int j = i; j < size; j++)

{

if (Arr[i][j] == 1)

{

addEdge(graph, i, j);

}

}

}

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest)

{

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

newNode = createNode(src);

newNode->next = graph->adjLists[dest];

graph->adjLists[dest] = newNode;

}

void printGraph(struct Graph\* graph)

{

printf("\nAdjancecy list");

for (int v = 0; v < graph->numVertices; v++)

{

struct node\* temp = graph->adjLists[v];

printf("\n|%d|", v);

while (temp)

{

printf(" -> %d", temp->vertex);

temp = temp->next;

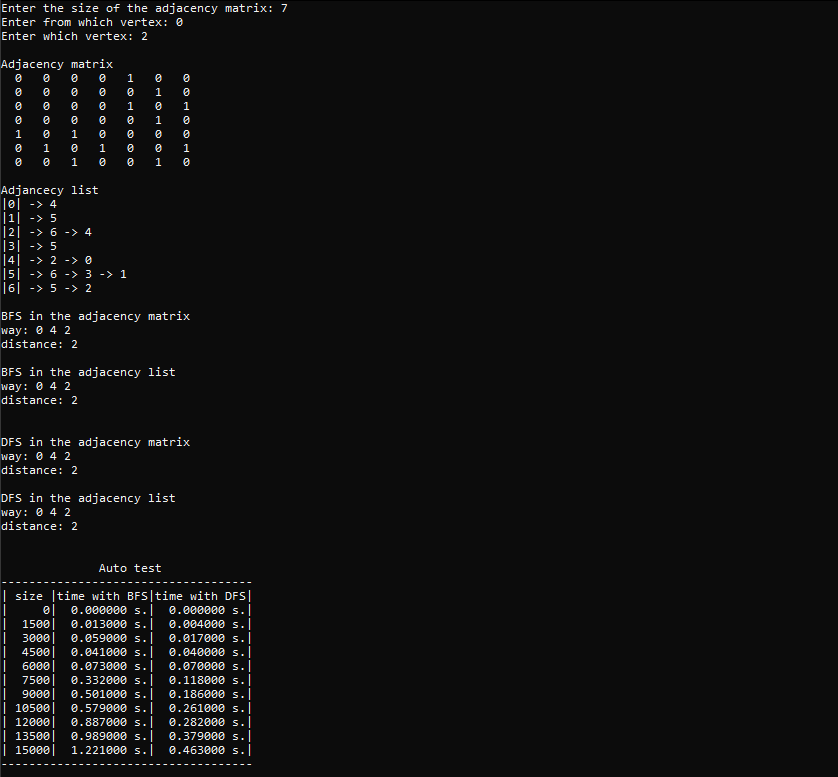
}

}

printf("\n");

}

**Результат работы программы**

****

**Вывод**

В результате выполнения лабораторной работы я научился выполнять поиск расстояний в графе.