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

**ОТЧËТ**  
по лабораторной работе №8  
по дисциплине: «Обход графа в ширину»

Выполнили студенты группы 22ВВВ2:  
Беляев Д.

Приняли:  
Акифьев И. В.

Митрохин М. А.

Пенза 2023

**Название**

Обход графа в ширину

**Цель работы**

Разработать и реализовать алгоритм обхода 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.

**Листинг**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "list.h"

#include "queue.h"

#include <queue>

int\*\* GenerateMatrix(int size\_x, int size\_y);

void ClearMatrix(int\*\* matrix, int size);

void PrintMatrix(int\*\* matrix, int size\_x, int size\_y);

void PrintListMatrix(DataType\*\* matrix, int size);

int\*\* GenerateAdjacencyMatrix(int size);

DataType\*\* GenerateListMatrix(int\*\* matrix, int size);

void BFS(int\*\* matrix, int size, int needPrint);

void BFSList(DataType\*\* list, int size);

void BFSLocal(int\*\* matrix, int size, int needPrint);

void AutoTest();

double CalculateSpeed(void (\*func)(int\*\*, int, int), int\*\* matrix, int size)

{

double time\_spent = 0.0;

clock\_t begin = clock();

func(matrix, size, 0);

clock\_t end = clock();

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

return time\_spent;

}

int main()

{

srand(time(NULL));

const int size = 4;

int\*\* matrix = GenerateAdjacencyMatrix(size);

printf("Adjacency matrix:\n");

PrintMatrix(matrix, size, size);

printf("---- START BFS RUN ----\n");

BFS(matrix, size, 1);

printf("\n");

DataType\*\* list = GenerateListMatrix(matrix, size);

printf("List matrix:\n");

PrintListMatrix(list, size);

printf("---- START BFS RUN ----\n");

BFSList(list, size);

printf("\n");

printf("---- START BFS LOCAL RUN ----\n");

BFSLocal(matrix, size, 1);

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

AutoTest();

return 0;

}

#pragma region Matrix work

int\*\* GenerateMatrix(int size\_x, int size\_y)

{

int\*\* matrix = (int\*\*)malloc(size\_x \* sizeof(int\*));

if (matrix == NULL)

return NULL;

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

{

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

if (matrix[i] == NULL)

return NULL;

}

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

{

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

{

matrix[i][j] = 0;

}

}

return matrix;

}

void ClearMatrix(int\*\* matrix, int size)

{

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

{

free(matrix[i]);

}

free(matrix);

}

void PrintMatrix(int\*\* matrix, int size\_x, int size\_y)

{

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

{

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

{

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

}

printf("\n");

}

}

void PrintListMatrix(DataType\*\* matrix, int size)

{

for (size\_t i = 0; i < size; i++)

{

PrintDataType(matrix[i]);

}

}

int\*\* RemoveVertex(int\*\* matrix, int\* size, int r)

{

int s = \*size;

int\*\* nMatrix = GenerateMatrix(s - 1, s - 1);

int di = 0, dj = 0;

for (size\_t i = 0; i < s; i++)

{

if (i == r)

i++;

if (i >= s)

break;

for (size\_t j = 0; j < s; j++)

{

if (j == r)

j++;

if (j >= s)

break;

nMatrix[di][dj] = matrix[i][j];

dj++;

}

di++;

dj = 0;

}

\*size = s - 1;

return nMatrix;

}

#pragma endregion

#pragma region MatrixGenerate

int\*\* GenerateAdjacencyMatrix(int size)

{

int\*\* matrix = GenerateMatrix(size, size);

if (matrix == NULL)

return NULL;

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

{

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

{

if (i == j)

continue;

int rnd = rand() % 2;

matrix[i][j] = rnd;

if (rnd == 1)

matrix[j][i] = 1;

}

}

return matrix;

}

DataType\*\* GenerateListMatrix(int\*\* matrix, int size)

{

DataType\*\* matrixL = (DataType\*\*)malloc(sizeof(DataType\*) \* size);

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

matrixL[i] = CreateDataType(i);

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

{

for (size\_t j = 0; j < size; j++)

{

if (matrix[i][j] != 1)

continue;

Add(j, matrixL[i]);

}

}

return matrixL;

}

#pragma endregion

#pragma region First Number

void BFSLogic(int\*\* matrix, int size, int vertex, int\* used, int needPrint)

{

std::queue<int> q;

q.push(vertex);

used[vertex] = 1;

while (!q.empty())

{

int cur = q.front();

q.pop();

if(needPrint == 1)

printf("%d ", cur);

for (size\_t i = 0; i < size; i++)

{

if (matrix[vertex][i] == 1 && used[i] == 0)

{

q.push(i);

used[i] = 1;

}

}

}

}

void BFS(int\*\* matrix, int size, int needPrint)

{

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

BFSLogic(matrix, size, 0, used , needPrint);

for (size\_t i = 0; i < size; i++)

{

if (used[i] == 0)

{

if(needPrint == 1)

printf("\n");

BFSLogic(matrix, size, i, used, needPrint);

}

}

if (needPrint == 1)

printf("\n");

free(used);

}

void BFSLogicList(DataType\*\* list, int size, int vertex, int\* used)

{

std::queue<int> q;

q.push(vertex);

used[vertex] = 1;

while (!q.empty())

{

int cur = q.front();

q.pop();

printf("%d ", cur);

int i = 1;

int\* v = TryCheckElementAt(i, list[vertex]);

while (v != NULL)

{

if (v == NULL)

break;

if (used[\*v] == 0)

{

q.push(\*v);

used[\*v] = 1;

}

i++;

v = TryCheckElementAt(i, list[vertex]);

}

}

}

void BFSList(DataType\*\* list, int size)

{

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

BFSLogicList(list, size, 0, used);

for (size\_t i = 0; i < size; i++)

{

if (used[i] == 0)

{

printf("\n");

BFSLogicList(list, size, i, used);

}

}

printf("\n");

free(used);

}

#pragma endregion

#pragma region Second Number

void BFSLogicLocal(int\*\* matrix, int size, int vertex, int\* used, int needPrint)

{

DataType\* q = CreateQueue(vertex);

used[vertex] = 1;

while (IsEmpty(q) == 0)

{

int cur;

q = Dequeue(q, &cur);

if(needPrint == 1)

printf("%d ", cur);

for (size\_t i = 0; i < size; i++)

{

if (matrix[vertex][i] == 1 && used[i] == 0)

{

q = Enqueue(i, q);

used[i] = 1;

}

}

}

}

void BFSLocal(int\*\* matrix, int size, int needPrint)

{

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

BFSLogicLocal(matrix, size, 0, used, needPrint);

for (size\_t i = 0; i < size; i++)

{

if (used[i] == 0)

{

if (needPrint == 1)

printf("\n");

BFSLogicLocal(matrix, size, i, used, needPrint);

}

}

if (needPrint == 1)

printf("\n");

free(used);

}

#pragma endregion

#pragma region AutoTest

void AutoTest()

{

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

printf("| size |time with class|time no class|\n");

for (size\_t i = 2000; i <= 20000; i+= 2000)

{

int\*\* matrix = GenerateAdjacencyMatrix(i);

double def = CalculateSpeed(BFS, matrix, i);

double own = CalculateSpeed(BFSLocal, matrix, i);

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

printf("%12f s.", def);

ClearMatrix(matrix, i);

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

}

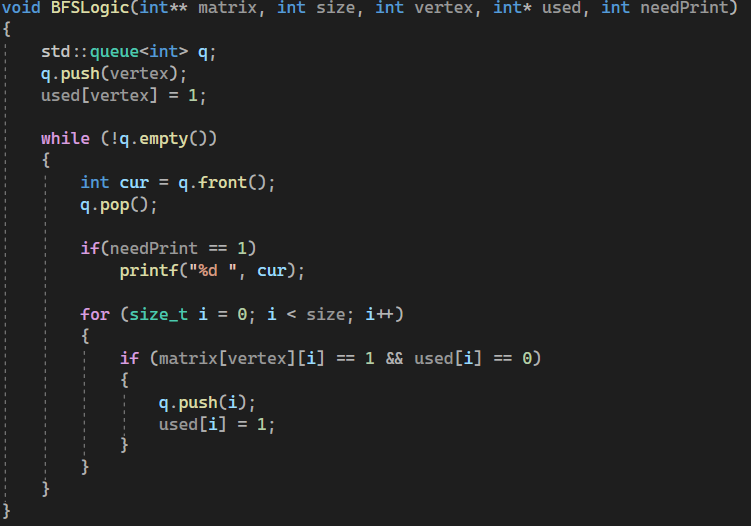
printf("--------------------------------------\n");

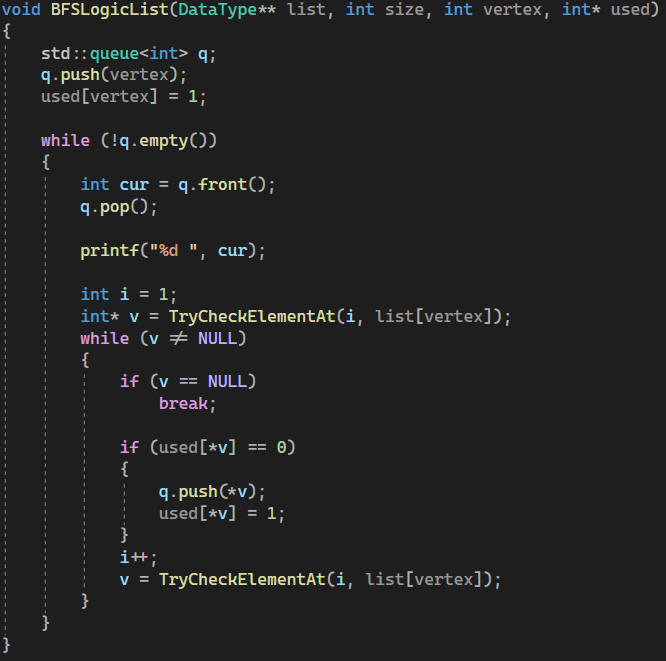
}

#pragma endregion

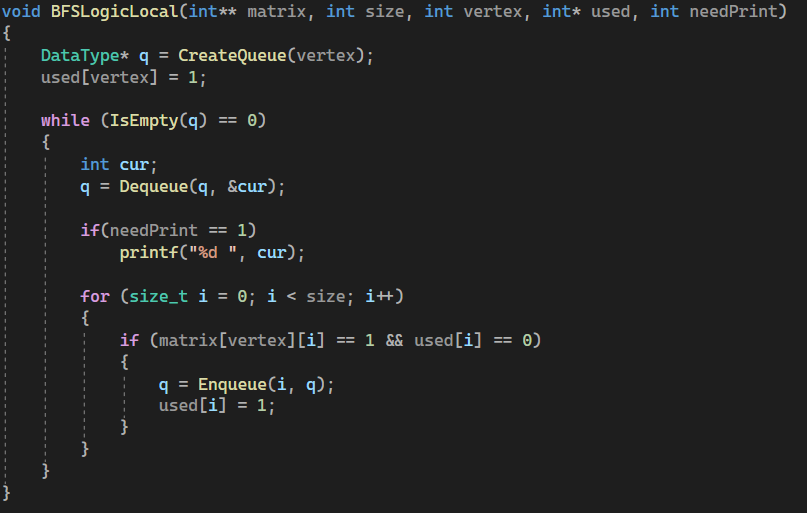
**Задания**

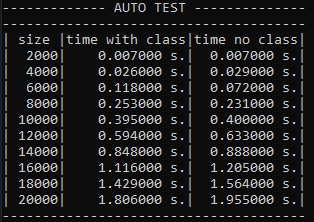
Задание 1





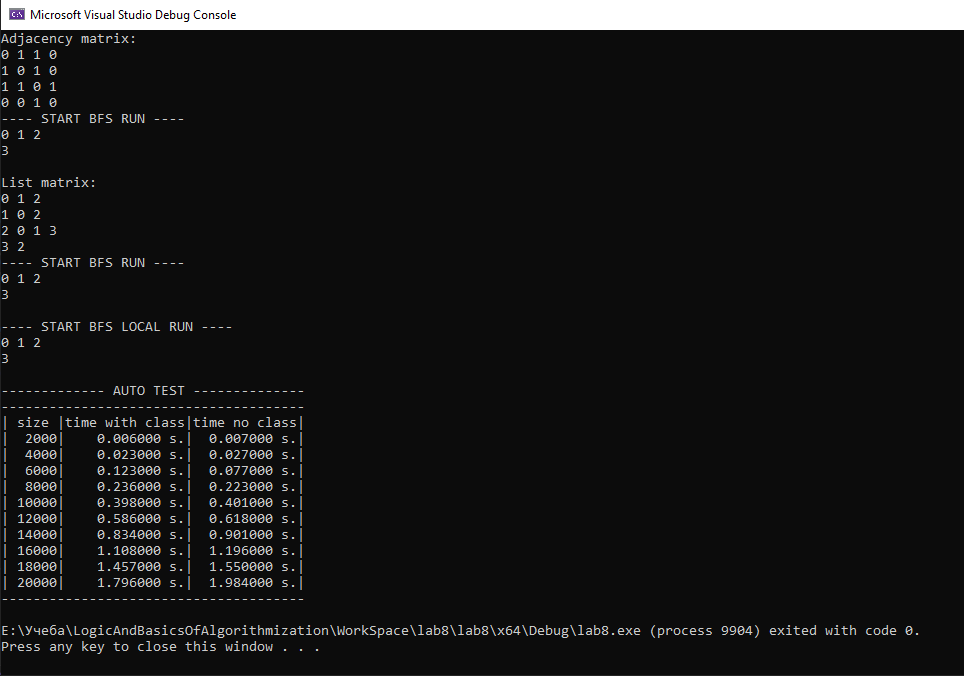
Задание 2





Проведя тесты, я пришел к выводу, что на малом размере матрицы встроенная в C++ очередь проигрывает самописной, но с повышением размеров эффективность алгоритма с встроенной очередью повышается, заметно превосходя аналогичный с самописной.

**Результат работы программы**

****

**Вывод**

Разработал и реализовал алгоритм обхода графа в ширину