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

**ОТЧËТ**  
по лабораторной работе № 9  
по дисциплине: «Поиск расстояний в графе»

Выполнили студенты группы 22ВВВ2:  
Беляев Д.

Приняли:  
Акифьев И. В.

Митрохин М. А.

Пенза 2023

**Название**

Поиск расстояний в графе

**Цель работы**

Разработать и реализовать алгоритм нахождения 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.

**Листинг**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "list.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, int in, int out);

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

void DFSadj(int\*\* matrix, int matrixSize, int needPrint, int in, int out);

void DFSlist(DataType\*\* list, int listSize, int in, int out, int needPrint);

void AutoTest();

int main()

{

srand(time(NULL));

const int size = 7;

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

printf("Adjacency matrix:\n");

PrintMatrix(matrix, size, size);

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

printf("List matrix:\n");

PrintListMatrix(list, size);

printf("\nInsert in and out vertex: ");

int in = 0, out = 0;

scanf("%d %d", &in, &out);

printf("\nAdj matrix, find by BFS: ");

BFS(matrix, size, 1, in, out);

printf("\List matrix, find by BFS: ");

BFSList(list, size, 1, in, out);

printf("\nAdj matrix, find by DFS: ");

DFSadj(matrix, size, 1, in, out);

printf("\List matrix, find by DFS: ");

DFSlist(list, size, in, out, 1);

printf("\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]);

}

}

#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 FirstNumber

DataType\* BFSLogic(int\*\* matrix, int size, int in, int out, int\* used, int\* pathSize)

{

std::queue<int> q;

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

memset(dst, -1, size \* sizeof(int));

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

memset(pr, -1, size \* sizeof(int));

q.push(in);

used[in] = 1;

dst[in] = 0;

while (!q.empty())

{

int cur = q.front();

q.pop();

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

{

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

{

q.push(i);

used[i] = 1;

dst[i] = dst[cur] + 1;

pr[i] = cur;

}

}

}

int cur = out;

DataType\* path = CreateDataType(cur);

while (pr[cur] != -1)

{

cur = pr[cur];

Add(cur, path);

}

int\* s = GetLength(path);

if (s == NULL)

\*pathSize = 0;

else

\*pathSize = \*s;

free(dst);

free(pr);

return path;

}

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

{

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

int count = 0;

DataType\* path = BFSLogic(matrix, size, in, out, used, &count);

if (needPrint)

{

if (count == 1)

printf("There is no path\n");

else

{

for (size\_t i = 0, j = count - 1; i < count; i++, j--)

{

printf("%d ", \*TryCheckElementAt(j, path));

}

printf("\n");

}

}

free(used);

path = Clear(path);

free(path);

}

DataType\* BFSLogicList(DataType\*\* list, int size, int in, int out, int\* used, int\* pathSize)

{

std::queue<int> q;

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

memset(dst, -1, size \* sizeof(int));

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

memset(pr, -1, size \* sizeof(int));

q.push(in);

used[in] = 1;

dst[in] = 0;

while (!q.empty())

{

int cur = q.front();

q.pop();

int i = 1;

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

while (v != NULL)

{

if (v == NULL)

break;

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

{

q.push(\*v);

used[\*v] = 1;

dst[\*v] = dst[cur] + 1;

pr[\*v] = cur;

}

i++;

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

}

}

int cur = out;

DataType\* path = CreateDataType(cur);

while (pr[cur] != -1)

{

cur = pr[cur];

Add(cur, path);

}

int\* s = GetLength(path);

if (s == NULL)

\*pathSize = 0;

else

\*pathSize = \*s;

free(dst);

free(pr);

return path;

}

void BFSList(DataType\*\* list, int size, int needPrint, int in, int out)

{

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

int count = 0;

DataType\* path = BFSLogicList(list, size, in, out, used, &count);

if (needPrint)

{

if (count == 1)

printf("There is no path\n");

else

{

for (size\_t i = 0, j = count - 1; i < count; i++, j--)

{

printf("%d ", \*TryCheckElementAt(j, path));

}

printf("\n");

}

}

free(used);

path = Clear(path);

free(path);

}

#pragma endregion

#pragma region Second Number

int DFSCuroadj(int in, int out, int\* visited, int\*\* matrix, int matrixSize, std::vector<int>\* path)

{

visited[in] = 1;

int end = -1;

for (size\_t r = 0; r <= matrixSize; r++)

{

if ((matrix[in][r] != 0) && (visited[r] == 0))

{

visited[in] = 1;

(\*path).push\_back(in);

if (r == out)

{

(\*path).push\_back(r);

return 1;

}

end = DFSCuroadj(r, out, visited, matrix, matrixSize, path);

if (end == 1)

return 1;

}

}

if (path->size() - 1 <= 0)

return 0;

visited[(\*path)[(\*path).size() - 1]] = 0;

path->erase(path->begin() + (path->size() - 1));

path->shrink\_to\_fit();

return 0;

}

void DFSadj(int\*\* matrix, int matrixSize, int needPrint, int in, int out)

{

int\* vis = (int\*)calloc(matrixSize, sizeof(int));

std::vector<int> path;

DFSCuroadj(in, out, vis, matrix, matrixSize, &path);

if (needPrint)

{

if (path.empty() || path.size() == 1)

printf("There is no way\n");

else

{

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

{

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

}

printf("\n");

}

}

free(vis);

path.clear();

path.shrink\_to\_fit();

}

int DFSCurolist(int in, int out, int\* visited, DataType\*\* list, int listSize, std::vector<int>\* path)

{

visited[in] = 1;

int i = 1;

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

while (v != NULL)

{

if (v == NULL)

break;

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

{

visited[in] = 1;

(\*path).push\_back(in);

if (\*v == out)

{

(\*path).push\_back(\*v);

return 1;

}

int end = DFSCurolist(\*v, out, visited, list, listSize, path);

if (end == 1)

return 1;

}

i++;

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

}

if (path->size() <= 0)

return 0;

visited[(\*path)[(\*path).size() - 1]] = 0;

path->erase(path->begin() + (path->size() - 1));

return 0;

}

void DFSlist(DataType\*\* list, int listSize, int in, int out, int needPrint)

{

int\* vis = (int\*)calloc(listSize, sizeof(int));

std::vector<int> path;

DFSCurolist(in, out, vis, list, listSize, &path);

if (needPrint)

{

if (path.empty() || path.size() == 1)

printf("There is no way\n");

else

{

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

{

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

}

printf("\n");

}

}

free(vis);

}

#pragma endregion

#pragma region AutoTest

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

{

double time\_spent = 0.0;

clock\_t begin = clock();

func(matrix, size, 0, 0, size -1);

clock\_t end = clock();

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

return time\_spent;

}

void AutoTest()

{

int in, out;

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

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

for (int i = 500; i <= 7000; i += 500)

{

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

double bfs = CalculateSpeed(BFS, matrix, i - 1);

double dfs = CalculateSpeed(DFSadj, matrix, i - 1);

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

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

ClearMatrix(matrix, i);

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

}

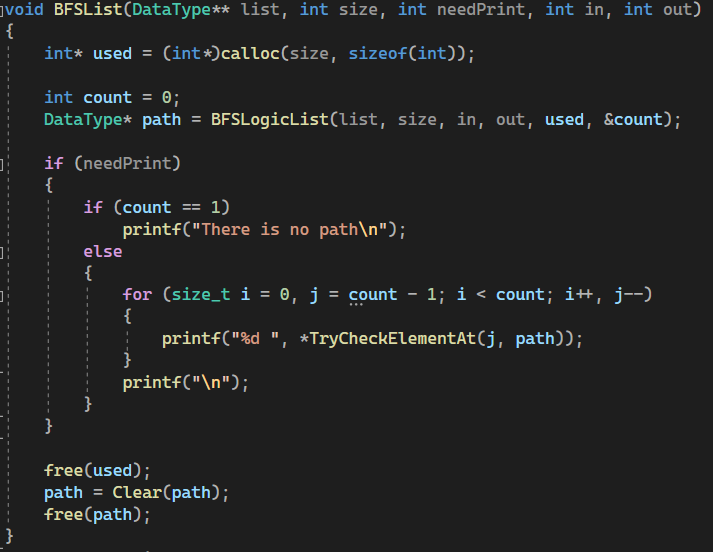
printf("--------------------------------------\n");

}

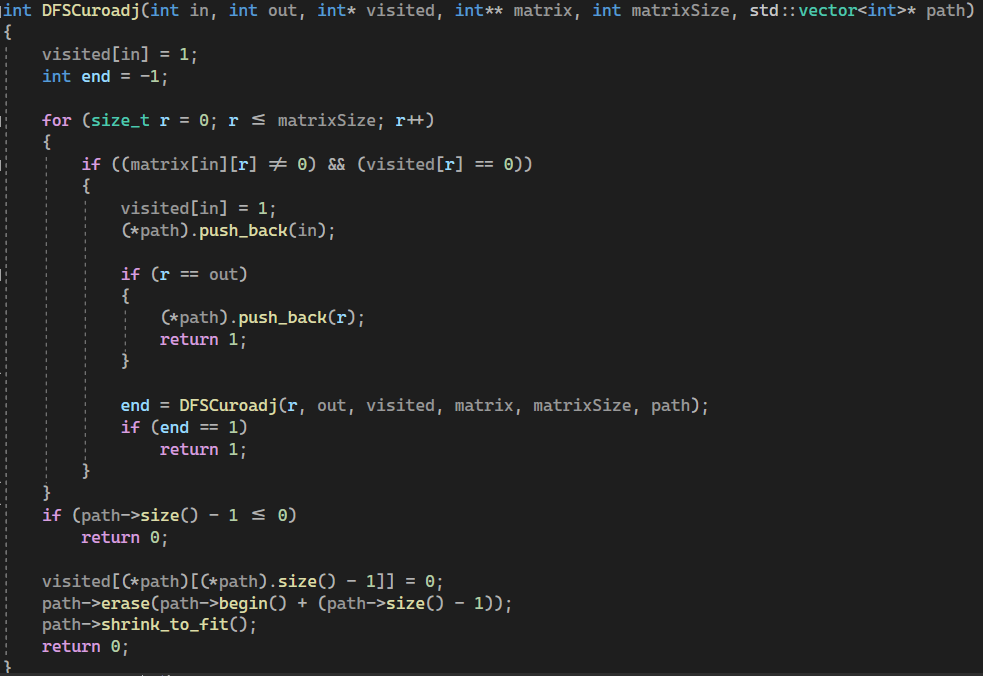
#pragma endregion

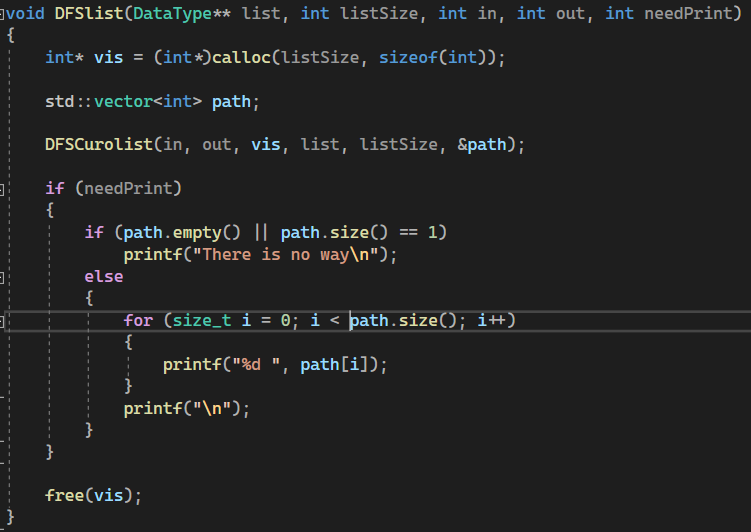
**Задания**

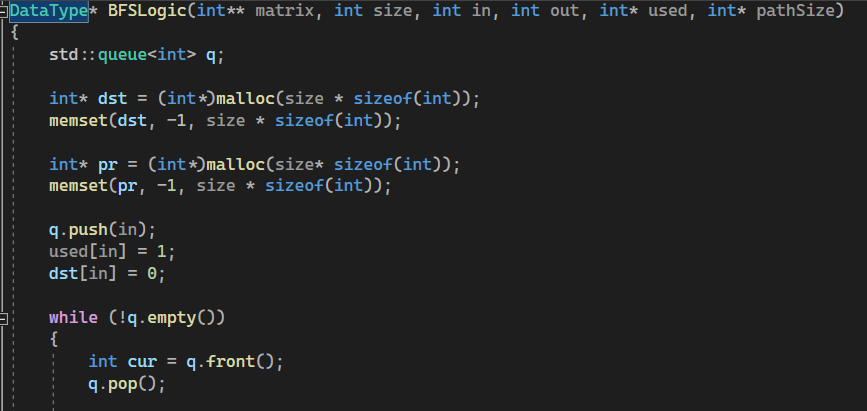
Задание 1

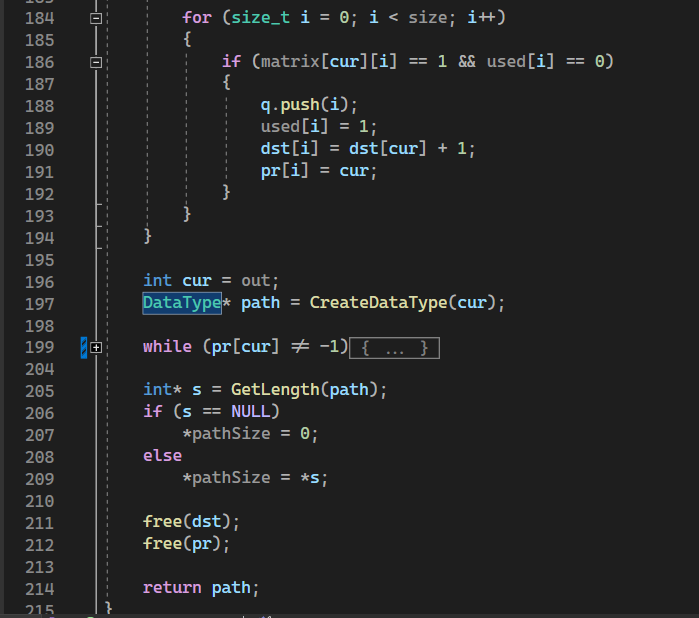


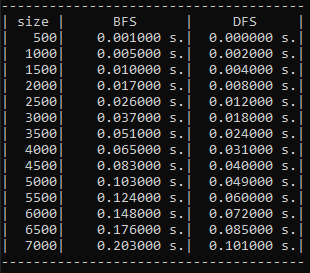
Задание 2





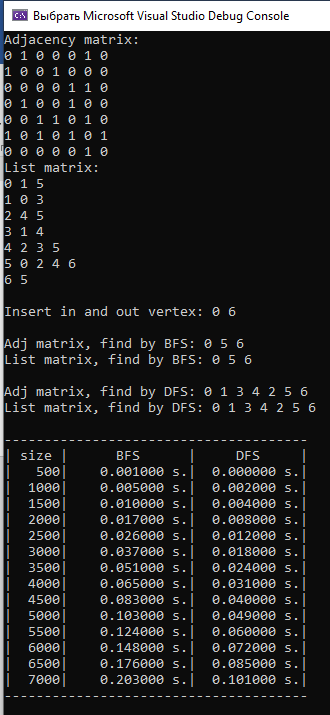
****

****

****

Проведя тесты, пришел к выводу, что скорость выполнения поиска пути через обход в ширину более медленная, но алгоритм дает кратчайший путь, а поиск через обход в глубину напротив более быстрый, но выдает не всегда оптимальный маршрут.

**Результат работы программы**

****

**Вывод**

Разработал и реализовал алгоритм нахождения расстояний в графе