|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 6** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Основные алгоритмы работы с графами»** | |
|  | |
| Выполнил студент группы ИКБО-07-21 | Дамарад Д.В. |
| Принял преподаватель | Скворцова Л.А. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

**СОДЕРЖАНИЕ**

[**ЦЕЛЬ РАБОТЫ** 3](#_Toc122021628)

[**ПОСТАНОВКА ЗАДАЧИ** 3](#_Toc122021629)

[**РЕШЕНИЕ** 4](#_Toc122021630)

[**ТЕСТИРОВАНИЕ** 11](#_Toc122021631)

[**ПОЛНЫЙ КОД ПРОГРАММЫ** 12](#_Toc122021632)

# **ЦЕЛЬ РАБОТЫ**

Получение практических навыков по выполнению операций над структурой данных граф.

# **ПОСТАНОВКА ЗАДАЧИ**

1. Разработать класс «Граф», обеспечивающий хранение и работу со структурой данных «граф», в соответствии с вариантом индивидуального задания. Реализовать метод ввода графа с клавиатуры, наполнение графа осуществлять с помощью метода добавления одного ребра. Реализовать метод вывода графа и методы, выполняющие задачи, определенные вариантом индивидуального задания.
2. Разработать программу, демонстрирующую работу всех методов класса.
3. Произвести тестирование программы на предоставляемом графе.
4. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

|  |  |  |
| --- | --- | --- |
| Вариант | Представление графа в памяти | Задачи |
| 6 | Список смежных вершин | Вывести все цепочки в графе, используя метод поиска в ширину.  Составить программу нахождения кратчайших путей упорядочив граф методом «Топологическая сортировка». Вывести путь, полученный методом. |

# **РЕШЕНИЕ**

Структура данных граф может быть использована для представления данных, моделей процессов, в программировании приложений. Многие приложения используют не только набор элементов данных, но также и набор соединений между парами этих элементов. Отношения, которые вытекают из этих соединений, позволяют ответить на вопросы:

1. Существует ли путь от одного такого элемента к другому, вдоль этого соединения?
2. В какие еще элементы можно перейти из заданного элемента?
3. Какой путь от одного заданного элемента к другому наилучший?

Примеры задач, данные которых представляют граф:

1. Определить населенный пункт, в котором разместить больницу, чтобы расстояние ото всех населенных пунктов было примерно одинаковым. В этой задаче, при выборе пути от одного населенного пункта к другому, необходимо 236 реализовать географическую карту местности. Населенные пункты соединены дорогами. Получаем множество попарно соединенных данных.
2. Планирование строительных работ при постройке жилого дома или производственных зданий. В этой задаче много различных видов работ, выполняемых в некоторой последовательности, выполнение одного вида работ позволяет выполнить множество следующих работ. Получаем множество попарно соединенных работ.
3. Организация поисковой системы интернета. При поиске в интернете необходимой информации мы сталкиваемся в документах со ссылками на другие документы. Множество соединенных документов образует граф.
4. Составление расписания (например, занятий) – производственный процесс требует решения задач при этом должно учитываться множество ограничений, по условиям которых не может быть начато решение одной задачи, пока не будет завершено решение другой задачи. Например, при составлении расписания конкретного преподавателя надо учитывать, день недели, номер пары, группу (группы в потоке), аудиторию, при этом: преподаватель на этой паре не занят, аудитория свободна и у группы в это время нет других занятий и возможно есть другие ограничения, которые должны быть согласованы.
5. Модель facebook – это граф. На facebook элементы данных представляют вершину графа, сюда входят: пользователь, фотография, альбом, событие, группа, страница, комментарий, история, видео, ссылка, примечание и т.д. является узлом. Связь (отношение) одного элемента с другим реализуется — это ребро от однй вершины к другой. Публикуете ли вы фотографию, присоединяетесь ли к группе, например, к странице и т. д. Для этих отношений создается новое ребро.

Основные понятия:

**Граф** - конечное не пустое множество вершин (V) и конечное множество упорядоченных или неупорядоченных пар вершин (ребер Е). Обозначим граф так: G={{V},{E}}. Вершины соединены линиями — это ребра графа. Ребро представляет соединение пары вершин.

**Ребро** – неупорядоченная пара вершин. Обозначение ребра (u, v) – ребро соединяет вершины u и v.

**Неориентированный граф** - содержит только неупорядоченные ребра. В таком графе есть ребро (u,v) и ребро (v,u).

**Ориентированный граф** – содержит только дуги.

**Дуга** – упорядоченная пара вершин. Вершины, соединенные ребром, называются смежными. Ребра, имеющие общую вершину, так же называются смежными. Ребро и любая из его вершин называются инцидентными.

**Степень вершины** – это число ребер, инцидентных этой вершине. В ориентированном графе степень определяется по количеству дуг, исходящих из вершины.

**Маршрут** — последовательность рёбер (в неориентированном графе), такая, что сток одного ребра является истоком другого ребра.

**Сток** – вершина, в которую входит ребро, так если ребро указано как (u, v), то v – сток, а u исток.

**Простой путь** – это путь, в котором все дуги различны. Другими словами, простой путь проходит через вершину только один раз.

**Полный граф** – это граф, у которого каждая вершина смежная со всеми остальными вершинами. Полный граф, состоящий из N вершин, содержит не более N(N-1) /2 ребер.

**Связный граф** — это граф, содержащий ровно одну компоненту связности. Это означает, что между любой парой вершин этого графа существует как минимум один путь. Несвязный граф состоит из некоторого множества связных компонентов, которые представляют собой максимальные связные подграфы.

**Взвешенный граф** – это граф, ребра которого несут информацию (говорят имеют вес) - некоторое значение, которое может определять: расстояние между населенными пунктами (вершинами); время проезда от одного населенного пункта до другого; длина кабеля, соединяющего компьютеры компьютерной сети и т.д.

**Длина пути** – это число ребер от вершины v к вершине u. Длина может быть различной для пути между двумя вершинами. Расстояние между вершинами v и u - r (v; u) – это путь с минимальным количеством ребер от вершины v к вершине u. Если между вершинами не существует никакого пути, то r между вершинами приравнивается к ∞. Диаметром d(G) графа G называется максимальное расстояние между его вершинами: d(G) = max d(vi; vj).

**Цикл (контур)** – путь, начинающийся и заканчивающийся в одной и той же вершине. Простой цикл – соответствующий ему путь простой.

**Ациклический граф** – это граф без циклов.

**Деревом** называется ациклический связный граф.

**Остовное дерево связного графа** – это подграф связного графа, который одержит все вершины этого графа и представляет единое дерево.

**Лес** - множество деревьев графа.

**Эксцентриситет для фиксированной вершины графа** – это расстояние до наиболее удаленной от нее вершины (с максимальным удалением).

Способы представления графа в оперативной памяти.

Для идентификации вершин графа будем использовать натуральные числа – номера вершин. Структура данных, как мы знаем, предназначена хранить элементы данных и отношения между ними. Элементы графа – вершины связаны отношением соединения пар. Следовательно, надо хранить бинарные отношения. Для хранения таких отношений удобно использовать матрицу, которую в программе реализуем на двумерном массиве или на структуре из линейных динамических списков:

1. Матрица смежности
2. Списки смежных вершин
3. Матрица инцидентности
4. Перечень ребер
5. Матрица весов взвешенного графа

Структура узла графа и способ его хранения:

struct Edge {

int src = -1;

int dest = -1;

int weight = -1;

};

vector<vector<Edge>> EdgeList;

Конструктор Graph изменяет размер вектора хранящий граф:

Graph::Graph(int n) {

EdgeList.resize(n);

}

Ввод графа осуществляется с помощью метода add, который принимает начальную и конечную вершину, а также вес смежного к этим вершинам ребра. Для начала проверяется возможность вставки узла в граф, если такая возможность есть, то инициализируется новое ребро которое будет вставлено в вектор графа, далее значение начальной вершины уменьшается на 1, для корректной работки с вектором данных и с помощью метода pushback() ребро добавляется в вектор:

bool Graph::add(int src, int dest, int weight) {

if ((src == 0 || dest == 0)) {

cout << "One of the vertices is 0" << endl;

return false;

}

if ((src > EdgeList.size() || dest > EdgeList.size()) && EdgeList.size() != 0) {

cout << "Not enough space in graph" << endl;

return false;

}

Edge edge = { src, dest , weight };

src--;

EdgeList[src].push\_back(edge);

return true;

}

Метод graphsize() возвращает размер вектора, хранящий граф:

int Graph::graphSize() {

return EdgeList.size();

}

Метод graphresize() увеличивает размер вектора, хранящий граф:

void Graph::graphResize(int size) {

if (size > EdgeList.size()) {

EdgeList.resize(size);

return;

}

else {

cout << "Error..." << endl;

}

}

Метод graphClear() очищает вектор, хранящий граф:

void Graph::graphClear() {

EdgeList.clear();

cout << "Graph has been cleared" << endl;

}

Метод printGraph() выводит граф в виде “вершина” -> “вершина” (“вес ребра”):

void Graph::printGraph() {

for (int i = 0; i < EdgeList.size(); i++) {

for (Edge v : EdgeList[i]) {

cout << v.src << " -> " << v.dest << " " << "(" << v.weight << ") " << "| ";

}

cout << endl;

}

}

Метод BFS() выводит ищет цепочку, начиная с некоторой вершины с помощью поиска в ширину:

void Graph::BFS(int src) {

map<int, bool> checkedNodes; //Проверенные узлы

queue<int> queue;

src--;

queue.push(src);

while (true) {

int currentNode = queue.front(); //Берем узел

queue.pop(); //Удаляем из очереди его

cout << currentNode + 1; //Вывод узла

for (int i = 0; i < EdgeList[currentNode].size(); i++) //Пробегаемся по связанным с ним узлами

{

if (checkedNodes.count(EdgeList[currentNode][i].dest) == 0) //Если это узел новый, то добавляем его в список проверенных и в очередь

{

checkedNodes[EdgeList[currentNode][i].dest] = true;

queue.push(EdgeList[currentNode][i].dest - 1);

}

}

if (queue.empty()) break;

else cout << "->";

}

}

Метод printChains() выводит все цепочки в графе:

void Graph::printChains() {

for (int i = 1; i < this->graphSize(); i++) {

this->BFS(i);

cout << endl;

}

}

Интерфейс программы (Рисунок 1) представляет из себя меню, состоящее из 5 пунктов, во всех пунктам, кроме второго, пользователю необходимо ввести вершину(ы), а также в первом пункте необходимо ввести расстояние между вводимыми вершинами. При введении числа не из промежутка [0,4] или некорректного символа программа попросит пользователя повторить ввод.

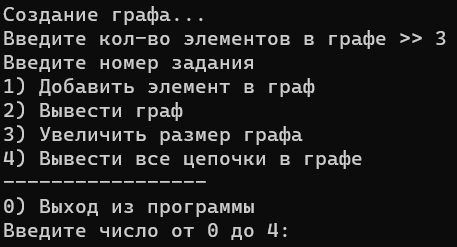


Рисунок 1 - Интерфейс программы

# **ТЕСТИРОВАНИЕ**

Протестируем создание графа, а также его вывод с помощью графа (Рисунок 3), приложенного к номеру варианта (Рисунок 2).

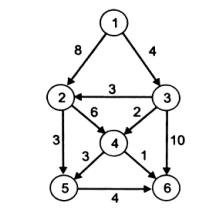


Рисунок 2 - Рисунок графа, приложенный к номеру варианта

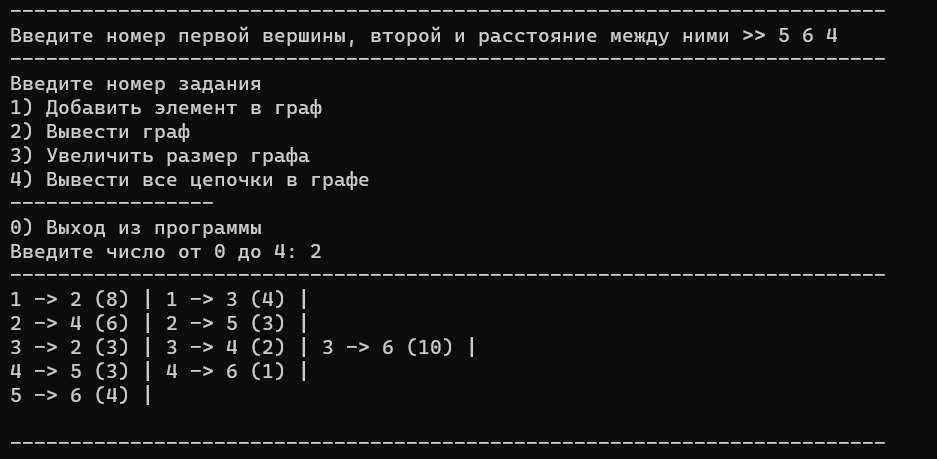


Рисунок 3 - Результат тестирования добавления ввода графа и его вывода

Протестируем вывод всех цепочек методомо поиска в ширину (Рисунок 4).

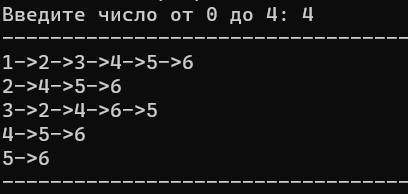


Рисунок 4 - Тестирование вывода всех цепочек

Из результатов программы видно, что все функции задания работают корректно.

# **ПОЛНЫЙ КОД ПРОГРАММЫ**

Файл Graph.h

#pragma once

#include <iostream>

#include <vector>

#include <queue>

#include <map>

using namespace std;

class Graph {

private:

struct Edge {

int src = -1;

int dest = -1;

int weight = -1;

};

vector<vector<Edge>> EdgeList;

void BFS(int src);

public:

Graph(int n);

bool add(int src, int dest, int weight);

int graphSize();

void graphResize(int size);

void graphClear();

void printGraph();

void printChains();

};

Файл Graph.cpp

#include "Graph.h"

Graph::Graph(int n) {

EdgeList.resize(n);

}

bool Graph::add(int src, int dest, int weight) {

if ((src == 0 || dest == 0)) {

cout << "One of the vertices is 0" << endl;

return false;

}

if ((src > EdgeList.size() || dest > EdgeList.size()) && EdgeList.size() != 0) {

cout << "Not enough space in graph" << endl;

return false;

}

Edge edge = { src, dest , weight };

src--;

EdgeList[src].push\_back(edge);

return true;

}

int Graph::graphSize() {

return EdgeList.size();

}

void Graph::graphResize(int size) {

if (size > EdgeList.size()) {

EdgeList.resize(size);

return;

}

else {

cout << "Error..." << endl;

}

}

void Graph::graphClear() {

EdgeList.clear();

cout << "Graph has been cleared" << endl;

}

void Graph::printGraph() {

for (int i = 0; i < EdgeList.size(); i++) {

for (Edge v : EdgeList[i]) {

cout << v.src << " -> " << v.dest << " " << "(" << v.weight << ") " << "| ";

}

cout << endl;

}

}

void Graph::BFS(int src) {

map<int, bool> checkedNodes; //Проверенные узлы

queue<int> queue;

src--;

queue.push(src);

while (true) {

int currentNode = queue.front(); //Берем узел

queue.pop(); //Удаляем из очереди его

cout << currentNode + 1; //Вывод узла

for (int i = 0; i < EdgeList[currentNode].size(); i++) //Пробегаемся по связанным с ним узлами

{

if (checkedNodes.count(EdgeList[currentNode][i].dest) == 0) //Если это узел новый, то добавляем его в список проверенных и в очередь

{

checkedNodes[EdgeList[currentNode][i].dest] = true;

queue.push(EdgeList[currentNode][i].dest - 1);

}

}

if (queue.empty()) break;

else cout << "->";

}

}

void Graph::printChains() {

for (int i = 1; i < this->graphSize(); i++) {

this->BFS(i);

cout << endl;

}

}

Файл main.cpp

#include "Graph.h"

int inputMenu(int& number);

int main() {

system("chcp 1251 > null");

int menu;

int n;

cout << "Создание графа...";

cout << endl << "Введите кол-во элементов в графе >> ";

cin >> n;

Graph graph(n);

while (true) {

inputMenu(menu);

if (menu == 0) {

graph.graphClear();

cout << "Задание не выбрано... Выход из программы" << endl;

break;

}

switch (menu) {

case 1: {

cout << "-------------------------------------------------------------------------" << endl;

int first, second, weight;

cout << "Введите номер первой вершины, второй и расстояние между ними >> ";

cin >> first;

cin >> second;

cin >> weight;

if (graph.add(first, second, weight) == false) {

cout << "Error..." << endl;

}

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 2: {

cout << "-------------------------------------------------------------------------" << endl;

graph.printGraph();

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 3: {

cout << "-------------------------------------------------------------------------" << endl;

cout << "Текущий размер графа = " << graph.graphSize() << endl;

int size;

cout << "Введите новый размер графа >> ";

cin >> size;

graph.graphResize(size);

cout << "Новый размер графа = " << graph.graphSize() << endl;

cout << "-------------------------------------------------------------------------" << endl;

break;

}

case 4: {

cout << "-------------------------------------------------------------------------" << endl;

graph.printChains();

cout << "-------------------------------------------------------------------------" << endl;

break;

}

default: {

break;

}

}

}

return 0;

}

int inputMenu(int& number) {

cout << "Введите номер задания" << endl;

cout << "1) Добавить элемент в граф" << endl;

cout << "2) Вывести граф" << endl;

cout << "3) Увеличить размер графа" << endl;

cout << "4) Вывести все цепочки в графе" << endl;

cout << "-----------------" << endl;

cout << "0) Выход из программы" << endl;

while (1) {

cout << "Введите число от 0 до 4: ";

if ((cin >> number).good() && (number >= 0) && (number <= 4)) {

return number;

}

if (cin.fail()) {

cin.clear();

cout << endl << "Неверный ввод, повторите." << endl << endl;

}

else {

cout << endl << "Число вне допустимого диапазона значений. Повторите ввод." << endl << endl;

}

cin.ignore(100, '\n');

}

}

//Тестовые данные

//1

//1 2 8

//1

//1 3 4

//1

//2 4 6

//1

//2 5 3

//1

//3 2 3

//1

//3 4 2

//1

//3 6 10

//1

//4 5 3

//1

//4 6 1

//1

//5 6 4