Министерство высшего образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«Пермский национальный исследовательский политехнический университет» (ПНИПУ)**

Электротехнический факультет

Кафедра «Информационные технологии и автоматизированные системы»

ОТЧЁТ

по лабораторной работе

Тема: «Графы»

Выполнил

Студент группы РИС-22-2б

Зубов Р. А.

Проверил доц. Кафедры ИТАС

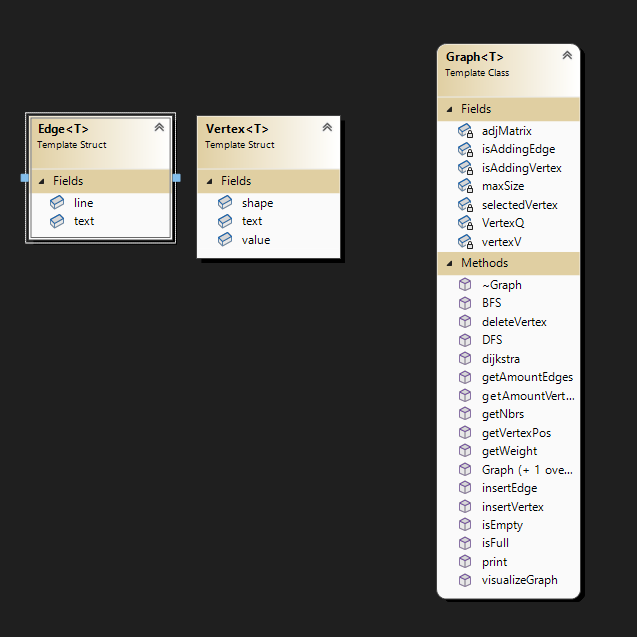
Полякова О.А.

Пермь 2023

# Постановка задачи

Реализовать алгоритмы для собственного варианта графа, имеющего не менее 6 вершин.  
  
Алгоритмы:  
1. Обход в ширину.  
2. Обход в глубину.  
3. Алгоритм Дейкстры.  
Требования:  
1. Пользовательский интерфейс на усмотрение разработчика с условием кроссплатформенности (поощряется использование Qt или иных фреймворков)  
2. Визуализация графа с использованием любой доступной графической библиотеки (SFML, SDL, OpenGL и подобных)  
3. Реализованные алгоритмы должны справляться как с графом, представленным в задании варианта, так и с другими на усмотрение проверяющего.  
4. Необходимо реализовать функции для редактирования графа:  
- Создание новой вершины.  
- Удаление вершины.  
- Добавление и удаление ребра.  
- Редактирование весов ребер.  
- Редактирование матрицы смежности (или инцидентности в зависимости от реализации).  
5. Выполнить отчет.

UML – диаграмма



Код программы

**main.cpp**

#include <SFML/Graphics.hpp>

#include <SFML/Window.hpp>

#include <vector>

#include <cmath>

#include <algorithm>

#include "Graph.h"

template<typename T>

void drawGraph(sf::RenderWindow& window, const Graph<T>& graph, const std::vector<Vertex<T>>& vertices, const std::vector<Edge<T>>& edges) {

for (const auto& edge : edges) {

window.draw(edge.line, 2, sf::Lines);

window.draw(edge.text);

}

for (const auto& vertex : vertices) {

window.draw(vertex.shape);

window.draw(vertex.text);

}

window.display();

}

int main() {

setlocale(LC\_ALL, "rus");

int amountVerts, amountEdges, weightEdge;

int vertex, sourceVertex, targetVertex;

bool\* visitedVertes = new bool[50];

std::fill(visitedVertes, visitedVertes + 50, false);

std::cout << "Enter num of vertex: "; std::cin >> amountVerts; std::cout << std::endl;

std::cout << "Enter num of edges: "; std::cin >> amountEdges; std::cout << std::endl;

Graph<int> graph(amountVerts \* amountEdges);

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

{

std::cout << "Vertex: "; std::cin >> vertex;

graph.insertVertex(vertex);

std::cout << std::endl;

}

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

{

std::cout << "Source vert: "; std::cin >> sourceVertex; std::cout << std::endl;

std::cout << "End vert: "; std::cin >> targetVertex; std::cout << std::endl;

std::cout << "Weight of edge: "; std::cin >> weightEdge; std::cout << std::endl;

graph.insertEdge(sourceVertex, targetVertex, weightEdge);

}

std::cout << std::endl;

graph.print();

graph.dijkstra(graph, 2);

int sVert = 1;

graph.DFS(sVert, visitedVertes);

std::fill(visitedVertes, visitedVertes + 50, false);

graph.BFS(sVert, visitedVertes);

graph.visualizeGraph(graph);

return 0;

}

**Graph.h**

#pragma once

#include <vector>

#include <iostream>

#include <limits>

#include <SFML/Graphics.hpp>

#include <SFML/Window.hpp>

#include <cmath>

#include <queue>

#ifndef M\_PI

#define M\_PI 3.14/\*159265358979323846\*/

#endif

template<typename T>

struct Vertex {

sf::CircleShape shape;

T value;

sf::Text text;

};

template<typename T>

struct Edge {

sf::Vertex line[2];

sf::Text text;

};

template<typename T>

class Graph

{

public:

Graph() {}

Graph(const int& size);

~Graph(){}

void insertEdge(const T& vert\_1, const T& vert\_2, int weight);

void insertVertex(const T& vert);

void deleteVertex(){ this->vertList.pop\_back(); }

int getAmountEdges();

int getAmountVertexes() { return this->vertexV.size(); }

bool isFull() { return this->vertexV.size() == this->maxSize; }

bool isEmpty() { return this->vertexV.size() == 0; }

int getVertexPos(const T& vert);

int getWeight(const T& vert\_1, const T& vert\_2);

std::vector<T> getNbrs(const T& vert);

void print();

void dijkstra(const Graph<T>& graph, const T& source);

void DFS(T& startVertex, bool\* visitedVertes);

void BFS(T& startVertex, bool\* visitedVertes);

void visualizeGraph(Graph<T>& graph);

private:

std::vector<T> vertexV;

std::vector<std::vector<int>> adjMatrix;

int maxSize;

bool isAddingVertex = false; // Флаг для добавления вершины

bool isAddingEdge = false; // Флаг для добавления ребра

T selectedVertex; // Выбранная вершина для добавления ребра

std::queue<T> VertexQ;

};

template<typename T>

Graph<T>::Graph(const int& size)

{

this->maxSize = size;

this->adjMatrix = std::vector<std::vector<T>>(size, std::vector<T>(size));

for (int i = 0; i < this->maxSize; i++)

{

for (int j = 0; j < this->maxSize; j++)

this->adjMatrix[i][j] = 0;

}

}

template<typename T>

void Graph<T>::insertEdge(const T & vert\_1, const T & vert\_2, int weight)

{

if (getVertexPos(vert\_1) != -1 && this->getVertexPos(vert\_2) != -1)

{

int vertPos\_1 = getVertexPos(vert\_1);

int vertPos\_2 = getVertexPos(vert\_2);

if (this->adjMatrix[vertPos\_1][vertPos\_2] != 0

&& this->adjMatrix[vertPos\_2][vertPos\_1] != 0)

{

std::cout << "The Edge between verts already exist" << std::endl;

return;

}

else

{

adjMatrix[vertPos\_1][vertPos\_2] = weight;

adjMatrix[vertPos\_2][vertPos\_1] = weight;

}

}

else

{

std::cout << "Two(or one of them) verts doesnt exist in graph" << std::endl;

return;

}

}

template<typename T>

void Graph<T>::insertVertex(const T& vert)

{

if (this->isFull())

{

std::cout << "Cant add vert" << std::endl;

return;

}

this->vertexV.push\_back(vert);

}

template<typename T>

int Graph<T>::getAmountEdges()

{

int amount = 0;

if (!this->isEmpty())

{

for (int i = 0; i < this->vertexV.size(); i++)

{

for (int j = 0; j < this->vertexV.size(); j++)

if (this->adjMatrix[i][j] != 0) amount++;

}

}

return amount / 2;

}

template<typename T>

int Graph<T>::getVertexPos(const T& vert)

{

for (int i = 0; i < this->vertexV.size(); i++)

{

if (this->vertexV[i] == vert) return i;

}

return -1;

}

template<typename T>

int Graph<T>::getWeight(const T& vert\_1, const T& vert\_2)

{

if (this->isEmpty()) return 0;

int vert\_1Pos = this->getVertexPos(vert\_1);

int vert\_2Pos = this->getVertexPos(vert\_2);

if (vert\_1Pos == -1 || vert\_2Pos == -1)

{

std::cout << "One of Node doesnt exist" << std::endl;

return 0;

}

return this->adjMatrix[vert\_1Pos][vert\_2Pos];

}

template<typename T>

std::vector<T> Graph<T>::getNbrs(const T& vert)

{

std::vector<T> NbrsVect;

int pos = this->getVertexPos(vert);

if (pos != -1)

{

for (int i = 0; i < this->vertexV.size(); i++)

if (this->adjMatrix[pos][i] != 0)

NbrsVect.push\_back(this->vertexV[i]);

}

return NbrsVect;

}

template<typename T>

void Graph<T>::print()

{

if (!this->isEmpty())

{

std::cout << "Adjustment Matrix: " << std::endl;

std::cout << " ";

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

std::cout << vertexV[i] << " ";

std::cout << std::endl;

for (int i = 0; i < this->vertexV.size(); i++)

{

std::cout << this->vertexV[i] << " ";

for (int j = 0; j < this->vertexV.size(); j++)

std::cout << " " << this->adjMatrix[i][j] << " ";

std::cout << std::endl;

}

}

else std::cout << "Graph is empty" << std::endl;

}

template<typename T>

void Graph<T>::dijkstra(const Graph<T>& graph, const T& source) {

int n = getAmountVertexes();

std::vector<int> distances(n, std::numeric\_limits<int>::max());

distances[getVertexPos(source)] = 0;

std::vector<bool> visited(n, false);

for (int i = 0; i < n - 1; i++) {

int minDistance = std::numeric\_limits<int>::max();

int minIndex = -1;

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

if (!visited[j] && distances[j] < minDistance) {

minDistance = distances[j];

minIndex = j;

}

}

if (minIndex == -1)

break;

visited[minIndex] = true;

std::vector<T> neighbors = getNbrs(graph.vertexV[minIndex]);

for (const T& neighbor : neighbors) {

int neighborPos = getVertexPos(neighbor);

int weight = getWeight(graph.vertexV[minIndex], neighbor);

if (!visited[neighborPos] && distances[minIndex] != std::numeric\_limits<int>::max()

&& distances[minIndex] + weight < distances[neighborPos]) {

distances[neighborPos] = distances[minIndex] + weight;

}

}

}

// Вывод кратчайших расстояний от исходной вершины до всех остальных вершин

std::cout << "Кратчайшие расстояния от вершины " << source << ":\n";

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

if (distances[i] == std::numeric\_limits<int>::max()) {

std::cout << "Вершина " << graph.vertexV[i] << " недостижима" << std::endl;

}

else {

std::cout << "Расстояние до вершины " << graph.vertexV[i] << ": " << distances[i] << std::endl;

}

}

}

template<typename T>

void Graph<T>::DFS(T& startVertex, bool\* visitedVertes)

{

std::cout << "Vertex " << startVertex << " is visited\n";

visitedVertes[this->getVertexPos(startVertex)] = true;

std::vector<T> neighbors = this->getNbrs(startVertex);

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

{

if (!visitedVertes[this->getVertexPos(neighbors[i])])

this->DFS(neighbors[i], visitedVertes);

}

}

template<typename T>

void Graph<T>::BFS(T& startVertex, bool\* visitedVertes)

{

if (visitedVertes[this->getVertexPos(startVertex)] == false)

{

this->VertexQ.push(startVertex);

std::cout << "Vertex " << startVertex << " is complete\n";

visitedVertes[this->getVertexPos(startVertex)] = true;

}

std::vector<T> neighbors = this->getNbrs(startVertex);

this->VertexQ.pop();

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

{

if (!visitedVertes[this->getVertexPos(neighbors[i])])

{

this->VertexQ.push(neighbors[i]);

visitedVertes[this->getVertexPos(neighbors[i])] = true;

std::cout << "Vertex " << neighbors[i] << " is complete\n";

}

}

if (this->VertexQ.empty()) return;

BFS(VertexQ.front(), visitedVertes);

}

template<typename T>

void Graph<T>::visualizeGraph(Graph<T>& graph) {

sf::RenderWindow window(sf::VideoMode(800, 600), "Graph Visualization");

window.setFramerateLimit(60);

// Создание вершин

std::vector<Vertex<T>> vertices;

int numVertices = graph.getAmountVertexes();

float radius = 200.0f;

float centerX = 400.0f;

float centerY = 300.0f;

sf::Font font;

font.loadFromFile("arial.ttf");

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

float angle = static\_cast<float>(i) \* (2.0f \* M\_PI / numVertices);

float x = centerX + radius \* std::cos(angle);

float y = centerY + radius \* std::sin(angle);

//Задание вершины

Vertex<T> vertex;

vertex.shape.setPosition(x, y);

vertex.shape.setRadius(30.0f);

vertex.shape.setFillColor(sf::Color::White);

vertex.shape.setOutlineThickness(2.0f);

vertex.shape.setOutlineColor(sf::Color::Black);

vertex.value = graph.vertexV[i];

//Добавление номера вершины в круг

sf::Text text(std::to\_string(vertex.value), font, 20);

text.setFillColor(sf::Color::Black);

text.setOrigin(text.getLocalBounds().width / 2, text.getLocalBounds().height / 2);

text.setPosition(vertex.shape.getPosition() + sf::Vector2f(vertex.shape.getRadius(), vertex.shape.getRadius()));

vertex.text = text;

vertices.push\_back(vertex);

}

// Создание ребер

std::vector<Edge<T>> edges;

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

T vertexValue = graph.vertexV[i];

std::vector<T> neighbors = graph.getNbrs(vertexValue);

for (const T& neighbor : neighbors) {

Edge<T> edge;

int vertexIndex = graph.getVertexPos(vertexValue);

int neighborIndex = graph.getVertexPos(neighbor);

sf::Vector2f vertexCenter = vertices[vertexIndex].shape.getPosition() + sf::Vector2f(vertices[vertexIndex].shape.getRadius(),

vertices[vertexIndex].shape.getRadius());

sf::Vector2f neighborCenter = vertices[neighborIndex].shape.getPosition() + sf::Vector2f(vertices[neighborIndex].shape.getRadius(),

vertices[neighborIndex].shape.getRadius());

edge.line[0].position = vertexCenter;

edge.line[0].color = sf::Color::Black;

edge.line[1].position = neighborCenter;

edge.line[1].color = sf::Color::Black;

// Определение позиции для отображения текста

sf::Vector2f startPos = edge.line[0].position;

sf::Vector2f endPos = edge.line[1].position;

sf::Vector2f Move(0, 15);

sf::Vector2f textPos = Move + startPos + (endPos - startPos) / 2.0f;

// Отображение текста с весом ребра над линией

sf::Text text(std::to\_string(graph.getWeight(vertexIndex + 1, neighborIndex + 1)), font, 16);

text.setFillColor(sf::Color::Black);

text.setOrigin(text.getLocalBounds().width / 2, text.getLocalBounds().height / 2);

text.setPosition(textPos);

edge.text = text;

edges.push\_back(edge);

}

}

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

if (event.type == sf::Event::Closed)

window.close();

}

window.clear(sf::Color::White);

drawGraph(window, graph, vertices, edges);

}

}

