Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«**Пермский национальный исследовательский политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**ОТЧЕТ**

Дисциплина: «Информатика»

Тема: «Деревья»

Семестр 2

Выполнил работу

Студент группы РИС-22-1Б

Баженов Т.И.

Проверил

Доцент кафедры ИТАС

Полякова О.А.

Г. Пермь-2023

**I.Введение**

В современном мире информационных технологи все стремится к автоматизации. Это упрощает человеку решение повседневных задач, ведь большинство из них можно доверить компьютеру. Для этого существуют различные структуры данных, которые помогают структурировать хранение данных.

Целью работы является разработка программы, которая представляет из себя структуру данных граф, а также визуализацию данной структуры.

Для достижения поставленной цели, необходимо решить следующие задачи:

1. Построить модель программы
2. Реализовать технологию разработки программы
3. Протестировать программу

**II.Анализ предметной области**

*Формулировка задачи*

Требуется создать программу, которая реализовывает структуру данных граф. В данном дереве реализовать алгоритмы обхода, поиска, вставки и удаления узла, а также алгоритм дейкстры.

Для решения данной задачи необходимо реализовать класс, представляющий структуру графа. Далее нужно продумать логику для реализации методов и отрисовки графа.

**Построение модели программы.**

Для того чтобы реализовать программу необходимо:

1. Реализовать класс граф.
2. Реализовать методы выполняющие поставленные алгоритмы.
3. Отрисовать граф после всех преобразований.

**Приложения**.

Код программы:

Main.cpp

#include "header.h"

int main()

{

system("chcp 1251 >> null");

bool upravlenie = true, dejkstra = false, salesman = false, close = false;

list<int> lst;

graph<string> gr;

gr.addVertex("a");

gr.addVertex("b");

gr.addVertex("c");

gr.addVertex("d");

gr.addVertex("e");

gr.addVertex("f");

gr.addVertex("g");

gr.addEdge("a", "b", 7);

gr.addDirectEdge("b", "c", 10);

gr.addDirectEdge("c", "g", 4);

gr.addDirectEdge("e","a", 13);

gr.addEdge("b", "f", 17);

gr.addEdge("c", "d", 8);

gr.addDirectEdge("b", "f", 13);

gr.addEdge("f", "g", 9);

gr.addEdge("d", "a", 25);

gr.addEdge("f", "d", 4);

gr.addEdge("g", "e", 11);

gr.addEdge("e", "d", 9);

gr.addEdge("g", "d", 22);

gr.addDirectEdge("b", "d", 13);

vector<sf::CircleShape> v(gr.size());

vector<sf::RectangleShape> v\_edges;

vector<sf::Text> v\_text;

vector<sf::Text> v\_weight;

vector<sf::Text> v\_dejkstra;

vector<int> for\_dejk;

//вершина/шрифт

sf::Font font;

font.loadFromFile("arial.ttf");

sf::Text text;

text.setFont(font);

text.setCharacterSize(18);

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

text.setStyle(sf::Text::Bold);

text.setOutlineColor(sf::Color::Green);

text.setOutlineThickness(2);

//ребро/шрифт

sf::Font font\_weight;

font\_weight.loadFromFile("arial.ttf");

sf::Text text\_weight;

text\_weight.setFont(font\_weight);

text\_weight.setCharacterSize(22);

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

text\_weight.setStyle(sf::Text::Bold);

text\_weight.setOutlineColor(sf::Color::White);

text\_weight.setOutlineThickness(4);

vector<sf::ConvexShape> v\_direction;

cordCalculate(v, gr.size());

gr.calcCordEdge(v\_edges, v\_direction,v);

gr.calcCordText(v\_text, v, text);

gr.calcCordWeight(v\_weight, v, text\_weight);

//окно

size\_t temp\_size = gr.size();

sf::Color color(255, 255, 255);

sf::RenderWindow window(sf::VideoMode(1000, 1000), "Graph");

window.setFramerateLimit(60);

thread t([&upravlenie, &gr, &dejkstra, &for\_dejk, &window, &salesman, &lst, & close](){ //параллельный поток для управления графом

auto vector\_my = gr.getVertexData();

string menu;

string data, data2;

int menu\_control;

int weight;

vector<string>::iterator it;

menu = "1. Добавить вершину\n2. Удалить вершину\n3. Добавить ненаправленное ребро\n"

"4. Добавить направленное ребро\n5. Удалить ребро\n6. Решение задачи коммивояжера\n7. Алгоритм Дейкстры\n8. Обход в ширину\n9. Обход в глубину\n0. Завершение работы программы";

while (true){

cout << menu << "\n";

cin >> menu\_control;

if (!window.isOpen()) {

return;

}

switch (menu\_control){

case 1:

cout << "Введите название вершины: " << "\n"; cin >> data;

gr.addVertex(data);

upravlenie = true;

break;

case 2:

cout << "Введите название вершины: " << "\n"; cin >> data;

gr.removeVertex(data);

upravlenie = true;

break;

case 3:

cout << "Введите две связные вершины: " << "\n"; cin >> data; cin >> data2;

cout << "Введите вес ребра: " << "\n"; cin >> weight;

gr.addEdge(data, data2, weight);

upravlenie = true;

break;

case 4:

cout << "Введите две связные вершины: " << "\n"; cin >> data; cin >> data2;

cout << "Введите вес ребра: " << "\n"; cin >> weight;

gr.addDirectEdge(data, data2, weight);

upravlenie = true;

break;

case 5:

cout << "Введите две связные вершины, ребро между которыми необходимо удалить: " << "\n"; cin >> data; cin >> data2;

gr.removeEdge(data, data2);

upravlenie = true;

break;

case 6:

cout << "Введите стартовую вершину: " << "\n"; cin >> data;

vector\_my = gr.getVertexData();

it = find(vector\_my.begin(), vector\_my.end(), data);

if (it == vector\_my.end()){

cout << "\n" << "\n" << "Вершина отсутствует." << "\n";

system("pause");

break;

}

gr.SalesmanTravel(distance(vector\_my.begin(), it));

lst = gr.getPath();

if(lst.size() != vector\_my.size()){

cout << "\n" << "\n" << "Решение задачи коммивояжера для данного графа не существует." << "\n";

system("pause");

}

else{

cout << "\n" << "Путь:" << "\n";

for (auto& i : lst){

cout << vector\_my[i] << " -> ";

}

cout << vector\_my[lst.front()] << "\n";

cout << "Расстояние: " << gr.getPathWeight() << "\n" << "\n";

salesman = true;

system("pause");

salesman = false;

}

break;

case 7:

vector\_my = gr.getVertexData();

cout << "Введите стартовую вершину: " << endl << "> "; cin >> data;

it = find(vector\_my.begin(), vector\_my.end(), data);

if (it == vector\_my.end())

{

cout << endl << endl << "Данной вершины не существует." << endl;

system("pause");

}

else

{

for\_dejk = gr.dijkstra(size\_t(distance(vector\_my.begin(), it)));

dejkstra = true;

system("pause");

dejkstra = false;

}

break;

case 8:

cout << "Введите стартовую вершину: " << endl << "> "; cin >> data;

cout << "Обход в ширину:" << endl;

if (!gr.BFSHelper(data))

{

system("cls");

cout << "Данной вершины не существует." << endl;

system("pause");

break;

}

system("pause");

break;

case 9:

cout << "Введите стартовую вершину: " << endl << "> "; cin >> data;

cout << "Обход в глубину:" << endl;

if (!gr.DFSHelper(data))

{

system("cls");

cout << "Данной вершины не существует." << endl;

system("pause");

break;

}

system("pause");

break;

default:

menu\_control = 0;

break;

}

system("cls");

string fl;

}

});

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event))

if (event.type == sf::Event::Closed || !t.joinable()) {

window.close();

t.join();

}

if (upravlenie){

upravlenie = false;

temp\_size = gr.size();

cordCalculate(v, temp\_size);

gr.calcCordEdge(v\_edges, v\_direction, v);

gr.calcCordText(v\_text, v, text);

gr.calcCordWeight(v\_weight, v, text\_weight);

}

if (salesman){

vector<sf::CircleShape> v\_point;

vector<sf::CircleShape> v\_path;

v\_path.reserve(800);

sf::CircleShape trailShape(5.0f);

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

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

sf::Image image;

image.loadFromFile("images/triangle.png");

image.createMaskFromColor(sf::Color(255, 255, 255));

sf::Texture triangle;

triangle.loadFromImage(image);

sf::Sprite triangle\_sp;

triangle\_sp.setTexture(triangle);

triangle\_sp.setScale(0.07f, 0.07f);

triangle\_sp.setOrigin(triangle\_sp.getLocalBounds().width / 2.0f, triangle\_sp.getLocalBounds().height / 2.0f);

const float animationSpeed = 700.0f;

sf::Clock clock;

float elapsedTime = 0.0f;

int current\_local, next\_local;

sf::Vector2f temp;

auto lst\_temp = gr.getPath();

int size\_temp = lst\_temp.size();

next\_local = lst\_temp.front();

for (size\_t i = 1; i < size\_temp; ++i){

elapsedTime = 0.0f;

current\_local = next\_local;

lst\_temp.pop\_front();

next\_local = lst\_temp.front();

v\_point.push\_back(v[current\_local]);

v\_point[i - 1].setOutlineColor(sf::Color::Black);

v\_point[i - 1].setOutlineThickness(9);

//установка спрайта

temp = v[current\_local].getPosition() - v[next\_local].getPosition();

triangle\_sp.setPosition(v[current\_local].getPosition());

triangle\_sp.setRotation((atan2(temp.y, temp.x) \* 180 / 3.14159) - 90);

sf::Vector2f startPosition = v[current\_local].getPosition();

sf::Vector2f targetPosition = v[next\_local].getPosition();

sf::Vector2f distance = targetPosition - startPosition;

float totalDistance = std::sqrt(distance.x \* distance.x + distance.y \* distance.y);

float travelTime = totalDistance / animationSpeed;

if (!salesman) break;

while (elapsedTime < travelTime) {

clock.restart();

window.clear(color);

if (!salesman) break;

for (int j = 0; j < v\_point.size(); j++){

window.draw(v\_point[j]);

}

for (int j = 0; j < v\_edges.size(); j++){

window.draw(v\_edges[j]);

window.draw(v\_direction[j]);

}

for (int j = 0; j < v\_path.size(); j++){

window.draw(v\_path[j]);

}

for (int j = 0; j < v.size(); j++){

window.draw(v[j]);

window.draw(v\_text[j]);

}

for (int j = 0; j < v\_weight.size(); j++){

window.draw(v\_weight[j]);

}

window.draw(triangle\_sp);

window.display();

elapsedTime += clock.getElapsedTime().asSeconds();

float t = elapsedTime / travelTime;

sf::Vector2f currentPosition = startPosition + distance \* t;

trailShape.setPosition(triangle\_sp.getPosition());

v\_path.push\_back(trailShape);

triangle\_sp.setPosition(currentPosition);

travelTime = totalDistance / animationSpeed;

sf::sleep(sf::milliseconds(10));

}

}

v\_point.push\_back(v[next\_local]);

v\_point.back().setOutlineColor(sf::Color::Black);

v\_point.back().setOutlineThickness(9);

if (salesman){

elapsedTime = 0.0f;

lst\_temp = gr.getPath();

temp = v[lst\_temp.back()].getPosition() - v[lst\_temp.front()].getPosition();

triangle\_sp.setPosition(v[lst\_temp.back()].getPosition());

triangle\_sp.setRotation((atan2(temp.y, temp.x) \* 180 / 3.14159) - 90);

sf::Vector2f startPosition = v[lst\_temp.back()].getPosition();

sf::Vector2f targetPosition = v[lst\_temp.front()].getPosition();

sf::Vector2f distance = targetPosition - startPosition;

float totalDistance = std::sqrt(distance.x \* distance.x + distance.y \* distance.y);

float travelTime = totalDistance / animationSpeed;

if (!salesman) {

break;

}

while (elapsedTime < travelTime){

clock.restart();

window.clear(color);

if (!salesman) break;

for (int j = 0; j < v\_point.size(); j++){

window.draw(v\_point[j]);

}

for (int j = 0; j < v\_edges.size(); j++){

window.draw(v\_edges[j]);

window.draw(v\_direction[j]);

}

for (int j = 0; j < v\_path.size(); j++){

window.draw(v\_path[j]);

}

for (int j = 0; j < v.size(); j++){

window.draw(v[j]);

window.draw(v\_text[j]);

}

for (int j = 0; j < v\_weight.size(); j++){

window.draw(v\_weight[j]);

}

window.draw(triangle\_sp);

window.display();

elapsedTime += clock.getElapsedTime().asSeconds();

float t = elapsedTime / travelTime;

sf::Vector2f currentPosition = startPosition + distance \* t;

trailShape.setPosition(triangle\_sp.getPosition());

v\_path.push\_back(trailShape);

triangle\_sp.setPosition(currentPosition);

travelTime = totalDistance / animationSpeed;

sf::sleep(sf::milliseconds(10));

}

while (salesman){

sf::sleep(sf::milliseconds(5));

}

}

}

else{

if (dejkstra){

gr.calcCordDijkstra(v\_dejkstra, v, for\_dejk, text);

}

window.clear(color);

for (int i = 0; i < v\_edges.size(); i++){

window.draw(v\_edges[i]);

window.draw(v\_direction[i]);

}

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

window.draw(v[i]);

}

for (int i = 0; i < v\_text.size(); i++){

window.draw(v\_text[i]);

}

for (int i = 0; i < v\_weight.size(); i++){

window.draw(v\_weight[i]);

}

if (dejkstra)

{

for (int i = 0; i < v\_dejkstra.size(); i++)

window.draw(v\_dejkstra[i]);

}

}

window.display();

sf::sleep(sf::milliseconds(5));

}

return 0;

}

//расстановка вершин

void cordCalculate(vector <sf::CircleShape>& vec, size\_t new\_size){

if (new\_size == 0){

vec.clear();

return;

}

float width, height;

vec.clear();

sf::CircleShape circle(40);

circle.setFillColor(sf::Color::Green);

circle.setOutlineThickness(3);

circle.setOutlineColor(sf::Color::Magenta);

float radius = 350.f;

sf::Vector2f center = { 500.f, 500.f };

float currentAngle = 0.f;

float angleStep = 360.f / new\_size;

for (int i = 0; i < new\_size; i++){

circle.setPosition(center.x + radius \* std::cos(currentAngle \* 3.14159 / 180.f), center.y + radius \* std::sin(currentAngle \* 3.14159 / 180.f));

currentAngle += angleStep;

width = circle.getLocalBounds().width;

height = circle.getLocalBounds().height;

circle.setOrigin(width / 2.0f, height / 2.0f);

vec.push\_back(circle);

}

}

#pragma once

#include "header.h"

template <typename T>

class graph{

public:

graph();

~graph();

size\_t size() {

return this->\_size;

}

void addVertex(T data);

void addEdge(T data\_v1, T data\_v2, int weight);

void addEdge(size\_t v1, size\_t v2, int weight);

void addDirectEdge(T v\_start, T v\_finish, int weight);

void addDirectEdge(size\_t start, size\_t finish, int weight);

void removeVertex(T data);

void removeEdge(T data\_v1, T data\_v2);

void removeEdge(size\_t v1, size\_t v2);

vector<T> getVertexData() {

return this->data;

}

void calcCordEdge(vector<sf::RectangleShape>& vec, vector<sf::ConvexShape>& v\_direction, const vector<sf::CircleShape>& vert);

void calcCordText(vector<sf::Text>& v\_text, const vector<sf::CircleShape>& vert, sf::Text& text);

void calcCordWeight(vector<sf::Text>& v\_weight, const vector<sf::CircleShape>& vert, sf::Text& text);

void calcCordDijkstra(vector<sf::Text>& v\_text, const vector<sf::CircleShape>& vert, const vector<int>& v\_dijkstra, sf::Text text);

vector<int> dijkstra(size\_t start);

void SalesmanTravel(const int& start);

list<int> getPath() {

return this->pathh;

}

int getPathWeight() {

return this->pathWeight;

}

void clearPath() {

this->pathh.clear();

}

void BFS(const int& v\_start);

void DFS(const int& v\_start);

bool BFSHelper(T data);

bool DFSHelper(T data);

private:

bool AllVisited(std::vector<bool>& visitedVerts);

std::vector<int> getNeighbours(const int& vert);

void mainSalesman(const int& start, const int& current, std::list<int>& path, std::vector<bool>& visitedVerts);

int pathWeight;

size\_t \_size;

vector<T> data;

vector<vector <int>> matrix;

list<int> pathh;

};

template<typename T>

inline bool graph<T>::BFSHelper(T data)

{

auto it = find(this->data.begin(), this->data.end(), data);

if (it == this->data.end())

return false;

BFS(distance(this->data.begin(), it));

return true;

}

template<typename T>

inline bool graph<T>::DFSHelper(T data)

{

auto it = find(this->data.begin(), this->data.end(), data);

if (it == this->data.end())

return false;

DFS(distance(this->data.begin(), it));

return true;

}

template <typename T>

vector<int> graph<T>::dijkstra(size\_t start) {

const int INF = INT\_MAX;

vector<int> distance(this->\_size, INF);

vector<bool> visited(this->\_size, false);

distance[start] = 0;

size\_t current;

int minDistance = INF;

// выбор непосещенной вершины с минимальным расстоянием

for (size\_t i = 0; i < this->\_size - 1; ++i) {

minDistance = INF;

current = -1;

for (size\_t j = 0; j < this->\_size; ++j) {

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

current = j;

minDistance = distance[j];

}

}

// если вершина не найдена, то алгоритм закончил работу

if (current == -1) break;

visited[current] = true;

// обновление расстояний до соседей текущей вершины

for (size\_t j = 0; j < this->\_size; ++j) {

if (this->matrix[current][j] != INF && this->matrix[current][j] != 0) { //проверяем вершину на существование

int pathLength = this->matrix[current][j] + minDistance;

if (pathLength < distance[j]) {

distance[j] = pathLength;

}

}

}

}

for (auto& i : distance) //если пути нет, то заменяем на несуществующее растояние

if (i == INF)

i = -1;

return distance;

}

template <typename T>

void graph<T>::DFS(const int& v\_start)

{

size\_t i = 0;

vector<bool> visited(\_size, false);

stack<int> st;

visited[v\_start] = true;

st.push(v\_start);

while (!st.empty())

{

auto current = st.top();

st.pop();

cout << ++i << ". " << this->data[current] << endl;

vector<int> neighbours = getNeighbours(current);

for (const auto& neighbour : neighbours)

{

if (!visited[neighbour])

{

visited[neighbour] = true;

st.push(neighbour);

}

}

}

}

template <typename T>

void graph<T>::BFS(const int& v\_start)

{

size\_t i = 0;

vector<bool> visited(\_size, false);

queue<int> queue;

visited[v\_start] = true;

queue.push(v\_start);

while (!queue.empty())

{

auto current = queue.front();

queue.pop();

cout << ++i << ". " << this->data[current] << endl;

vector<int> neighbours = getNeighbours(current);

for (const auto& neighbour : neighbours)

{

if (!visited[neighbour])

{

visited[neighbour] = true;

queue.push(neighbour);

}

}

}

}

template<typename T>

inline std::vector<int> graph<T>::getNeighbours(const int& vert){

std::list<int> result;

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

if (this->matrix[vert][j] != 0 && this->matrix[vert][j] != INT\_MAX){

result.push\_back(j);

}

}

return std::vector<int>(result.begin(), result.end());

}

template<typename T>

inline void graph<T>::SalesmanTravel(const int& start){

this->pathh.clear();

this->pathWeight = INT\_MAX;

std::list<int> path;

path.clear();

path.push\_back(start);

std::vector<bool> vis(this->data.size(), false);

vis[start] = true;

this->mainSalesman(start, start, path, vis);

}

template<typename T>

inline bool graph<T>::AllVisited(std::vector<bool>& visitedVerts){

bool flag = true;

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

if (visitedVerts[i] != true) {

flag = false;

}

}

return flag;

};

template<typename T>

inline void graph<T>::mainSalesman(const int& start, const int& current, std::list<int>& path, std::vector<bool>& visitedVerts){

if (this->AllVisited(visitedVerts)){

int min = 0;

if (!this->matrix[current][start]){

return;

}

std::vector<int> p(path.begin(), path.end());

for (int i = 0; i < p.size() - 1; ++i){

min += this->matrix[p[i]][p[i + 1]];

}

min += this->matrix[current][start];

if (this->pathWeight >= min){

this->pathWeight = min;

this->pathh = path;

}

}

std::vector<int> nbrs = this->getNeighbours(current);

for (int& i : nbrs){

if (!visitedVerts[i]){

std::vector<bool> cpy\_vis(visitedVerts.begin(), visitedVerts.end());

std::list<int> path\_cpy(path.begin(), path.end());

cpy\_vis[i] = true;

path\_cpy.push\_back(i);

mainSalesman(start, i, path\_cpy, cpy\_vis);

}

};

}

template<typename T>

inline graph<T>::graph(){

matrix.reserve(10);

data.reserve(10);

\_size = 0;

pathWeight = INT\_MAX;

}

template<typename T>

inline void graph<T>::addVertex(T data){

this->data.push\_back(data);

++this->\_size;

vector<int> vec(this->data.size(), 0);

vec[\_size - 1] = INT\_MAX;

for (auto& i : this->matrix) {

i.push\_back(0);

}

this->matrix.push\_back(vec);

}

template<typename T>

inline void graph<T>::addEdge(T data\_v1, T data\_v2, int weight){

auto it1 = find(this->data.begin(), this->data.end(), data\_v1);

auto it2 = find(this->data.begin(), this->data.end(), data\_v2);

if (it1 == this->data.end() || it2 == this->data.end()) return;

addEdge(size\_t(distance(this->data.begin(), it1)), size\_t(distance(this->data.begin(), it2)), weight);

}

template<typename T>

void graph<T>::addEdge(size\_t v1, size\_t v2, int weight){

if (v1 == v2) return;

this->matrix[v1][v2] = weight;

this->matrix[v2][v1] = weight;

}

template<typename T>

inline void graph<T>::addDirectEdge(T v\_start, T v\_finish, int weight){

auto it1 = find(this->data.begin(), this->data.end(), v\_start);

auto it2 = find(this->data.begin(), this->data.end(), v\_finish);

if (it1 == this->data.end() || it2 == this->data.end()) return;

addDirectEdge(size\_t(distance(this->data.begin(), it1)), size\_t(distance(this->data.begin(), it2)), weight);

}

template<typename T>

inline void graph<T>::addDirectEdge(size\_t start, size\_t finish, int weight){

if (start == finish) return;

this->matrix[start][finish] = weight;

}

template<typename T>

inline void graph<T>::removeVertex(T data){

auto it1 = find(this->data.begin(), this->data.end(), data);

if (it1 == this->data.end()) return;

this->matrix.erase(matrix.begin() + distance(this->data.begin(), it1));

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

this->matrix[i].erase(this->matrix[i].begin() + distance(this->data.begin(), it1));

this->data.erase(it1);

--this->\_size;

}

template<typename T>

inline void graph<T>::removeEdge(T data\_v1, T data\_v2){

auto it1 = find(this->data.begin(), this->data.end(), data\_v1);

auto it2 = find(this->data.begin(), this->data.end(), data\_v2);

if (it1 == this->data.end() || it2 == this->data.end()) return;

removeEdge(size\_t(distance(this->data.begin(), it1)), size\_t(distance(this->data.begin(), it2)));

}

template<typename T>

inline void graph<T>::removeEdge(size\_t v1, size\_t v2){

if (v1 == v2) return;

this->matrix[v1][v2] = 0;

this->matrix[v2][v1] = 0;

}

template<typename T>

inline void graph<T>::calcCordEdge(vector<sf::RectangleShape>& edges, vector<sf::ConvexShape>& v\_direction, const vector<sf::CircleShape>& vert){

edges.clear();

v\_direction.clear();

if (this->matrix.size() == 0) return;

float length, angle;

float width, height;

sf::ConvexShape triangle\_trash(3);

triangle\_trash.setPoint(0, sf::Vector2f(0, 0));

triangle\_trash.setPoint(1, sf::Vector2f(0, 0));

triangle\_trash.setPoint(2, sf::Vector2f(0, 0));

sf::ConvexShape triangle(3);

triangle.setPoint(0, sf::Vector2f(0, 0));

triangle.setPoint(1, sf::Vector2f(25, 0));

triangle.setPoint(2, sf::Vector2f(14, 25));

triangle.setFillColor(sf::Color::Magenta);

triangle.setOutlineThickness(0);

float x, y, k, b, t;

sf::Vector2f center1;

sf::Vector2f center2;

sf::Vector2f temp;

size\_t row = 0, column = 1;

while (row < this->\_size - 1) {

while (column < this->\_size){

if (this->matrix[row][column] == 0 && this->matrix[column][row] == 0)

++column;

else if (this->matrix[column][row] != 0){

temp = vert[column].getPosition() - vert[row].getPosition();

length = sqrt(pow(temp.x, 2) + pow(temp.y, 2));

angle = atan2(temp.y, temp.x) \* 180 / 3.14159;

sf::RectangleShape line(sf::Vector2f(length, 3.f));

line.setPosition(vert[row].getPosition());

line.setRotation(angle);

line.setFillColor(sf::Color::Green);

edges.push\_back(line);

if (this->matrix[row][column] == 0){

center1 = vert[column].getPosition();

center2 = vert[row].getPosition();

k = (center2.y - center1.y) / (center2.x - center1.x);

b = center1.y - k \* center1.x;

t = 1 - (55 / length);

x = center1.x + t \* (center2.x - center1.x);

y = center1.y + t \* (center2.y - center1.y);

width = triangle.getLocalBounds().width;

height = triangle.getLocalBounds().height;

triangle.setOrigin(width / 2.0f, height / 2.0f);

triangle.setRotation(angle + 90);

triangle.setPosition(x, y);

v\_direction.push\_back(triangle);

}

else{

v\_direction.push\_back(triangle\_trash);

}

++column;

}

else{

temp = vert[row].getPosition() - vert[column].getPosition();

length = sqrt(pow(temp.x, 2) + pow(temp.y, 2));

angle = atan2(temp.y, temp.x) \* 180 / 3.14159;

sf::RectangleShape line(sf::Vector2f(length, 3.f));

line.setPosition(vert[column].getPosition());

line.setRotation(angle);

line.setFillColor(sf::Color::Green);

edges.push\_back(line);

if (this->matrix[column][row] == 0){

center1 = vert[row].getPosition();

center2 = vert[column].getPosition();

k = (center2.y - center1.y) / (center2.x - center1.x);

b = center1.y - k \* center1.x;

t = 1 - (50 / length);

x = center1.x + t \* (center2.x - center1.x);

y = center1.y + t \* (center2.y - center1.y);

width = triangle.getLocalBounds().width;

height = triangle.getLocalBounds().height;

triangle.setOrigin(width / 2.0f, height / 2.0f);

triangle.setRotation(angle + 90);

triangle.setPosition(x, y);

v\_direction.push\_back(triangle);

}

else{

v\_direction.push\_back(triangle\_trash);

}

++column;

}

}

++row;

column = row + 1;

}

}

template<typename T>

inline void graph<T>::calcCordText(vector<sf::Text>& v\_text, const vector<sf::CircleShape>& vert, sf::Text& text){

float width, height;

v\_text.clear();

string str;

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

str = this->data[i];

text.setString(str);

width = text.getLocalBounds().width;

height = text.getLocalBounds().height;

text.setOrigin(width / 2.0f, height / 2.0f);

text.setPosition(vert[i].getPosition().x, vert[i].getPosition().y - 5);

v\_text.push\_back(text);

}

}

template<typename T>

inline void graph<T>::calcCordWeight(vector<sf::Text>& v\_weight, const vector<sf::CircleShape>& vert, sf::Text& text){

if (vert.size() == 0) return;

v\_weight.clear();

string str;

float x1, y1, x2, y2, midX, midY;

float width = 0, height = 0;

size\_t row = 0, column = 1;

while (row < this->\_size - 1){

while (column < this->\_size){

if (this->matrix[row][column] == 0 && this->matrix[column][row] == 0) {

++column;

}

else if (this->matrix[column][row] != 0){

str = to\_string(this->matrix[column][row]);

str = str.substr(0, str.length() - 7);

text.setString(str);

x1 = vert[column].getPosition().x;

y1 = vert[column].getPosition().y;

x2 = vert[row].getPosition().x;

y2 = vert[row].getPosition().y;

midX = (x1 + x2) / 2;

midY = (y1 + y2) / 2;

sf::FloatRect textRect = text.getLocalBounds();

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

text.setPosition(midX, midY);

v\_weight.push\_back(text);

++column;

}

else{

str = to\_string(this->matrix[row][column]);

str = str.substr(0, str.length() - 7);

text.setString(str);

x1 = vert[row].getPosition().x;

y1 = vert[row].getPosition().y;

x2 = vert[column].getPosition().x;

y2 = vert[column].getPosition().y;

midX = (x1 + x2) / 2;

midY = (y1 + y2) / 2;

sf::FloatRect textRect = text.getLocalBounds();

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

text.setPosition(midX, midY);

v\_weight.push\_back(text);

++column;

}

}

++row;

column = row + 1;

}

}

template<typename T>

inline void graph<T>::calcCordDijkstra(vector<sf::Text>& v\_text, const vector<sf::CircleShape>& vert, const vector<int>& v\_dijkstra, sf::Text text)

{

float width, height;

v\_text.clear();

string str;

text.setOutlineThickness(3);

for (int i = 0; i < v\_dijkstra.size(); i++)

{

str = to\_string(v\_dijkstra[i]);

text.setString(str);

width = text.getLocalBounds().width;

height = text.getLocalBounds().height;

text.setOrigin(width / 2.0f, height / 2.0f);

text.setPosition(vert[i].getPosition().x, vert[i].getPosition().y - 60); //

v\_text.push\_back(text);

}

}

template<typename T>

graph<T>::~graph(){

this->\_size = 0;

this->data.clear();

this->matrix.clear();

}

#pragma once

#include <iostream>

#include <fstream>

#include <vector>

#include <queue>

#include <stack>

#include <algorithm>

#include <cmath>

#include <unordered\_set>

#include <SFML/Graphics.hpp>

#include <thread>

using namespace std;

#include "Graph.h"

void cordCalculate(vector <sf::CircleShape>& vec, size\_t new\_size);

UML диаграмма классов:

