Министерство образования и науки Российской Федерации

Федеральное государственное образовательное автономное учреждение высшего образования

"Пермский национальный исследовательский политехнический университет"

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ

Дисциплина: Основы алгоритмизации

Тема: Граф

|  |  |
| --- | --- |
| Выполнил работу | |
| Студент группы РИС-22-1б | |
| Деревнин И.В. | |
|  | |
| Проверил работу | |
| Доцент кафедры ИТАС | |
| Полякова О.А. | |
|  | |

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.

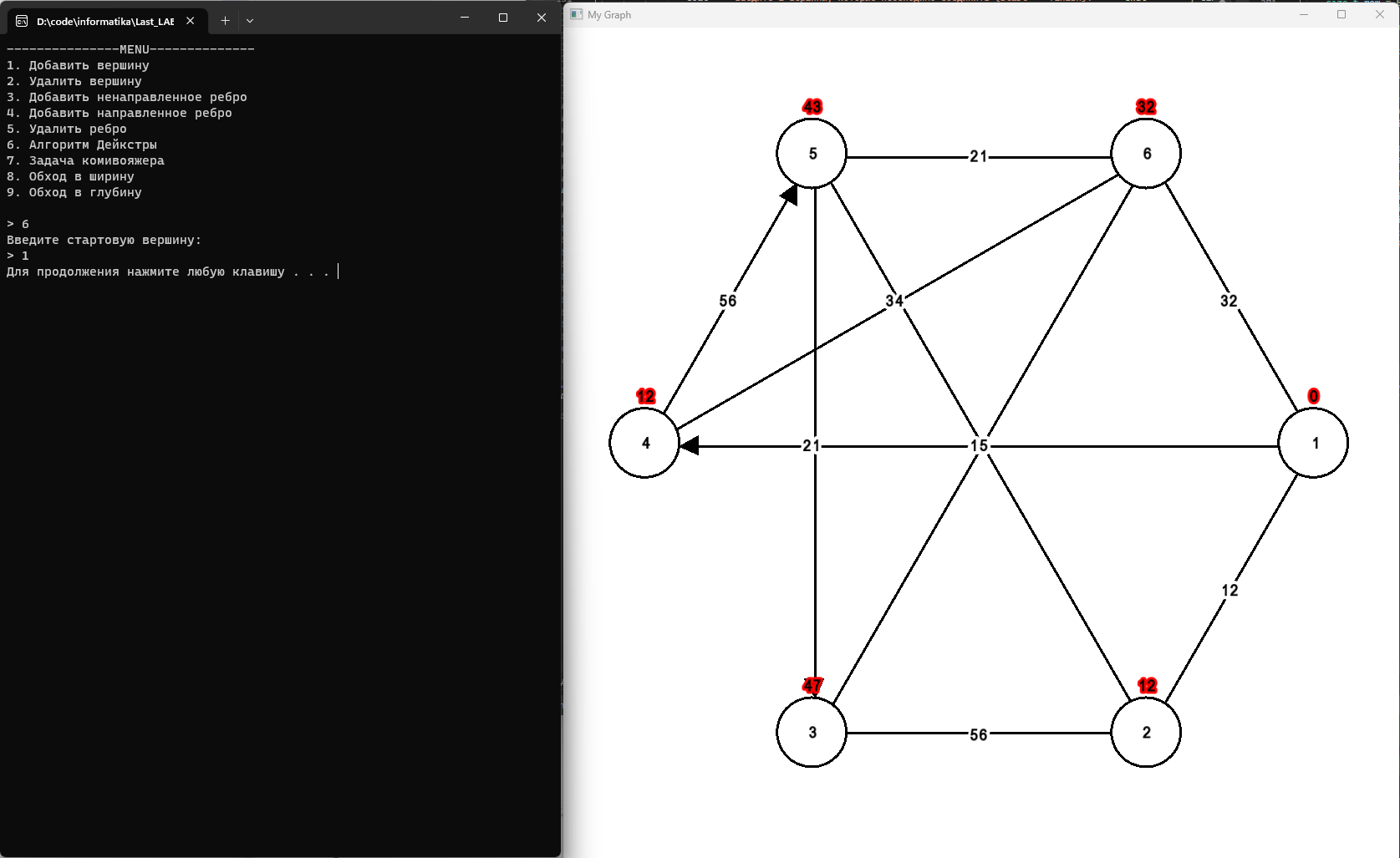
**Построение модели программы.**

Для того чтобы реализовать программу необходимо:

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

**III.Тестирования программы**

Тестирование программы:

*Рис. 1 – Тестирование программы.*

**Заключение**

Была разработана программа, которая представляет из себя структуру данных граф, а также отрисовывает данную структуру. Полученная программа может в дальнейшем может модифицироваться или использоваться в более сложных проектах, как отдельный компонент.

**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("чер"); //0

//gr.addVertex("12.2"); //1

//gr.addVertex("13.3"); //2

//gr.addVertex("14.4"); //3

//gr.addVertex("15.5"); //4

//gr.addVertex("17.7"); //5

//gr.addVertex("10.01"); //6

//gr.addEdge("12.2", "13.3", 23);

//gr.addDirectEdge("17.7", "13.3", 1);

//gr.addDirectEdge("14.4", "17.7", 2);

//gr.addDirectEdge("10.01","чер", 23);

//gr.addEdge("чер", "14.4", 25);

//gr.addEdge("10.01", "14.4", 23);

//gr.addDirectEdge("15.5", "12.2", 13);

//gr.addEdge("12.2", "17.7", 35);

//gr.addEdge("10.01", "13.3", 25);

//gr.addEdge("10.01", "12.2", 12);

//gr.addEdge("12.2", "14.4", 12);

//gr.addEdge("14.4", "13.3", 1);

//gr.addEdge("14.4", "15.5", 53);

//gr.addEdge("17.7", "15.5", 42);

//gr.addEdge("17.7", "10.01", 15);

//gr.BFSHelper("12.2");

//cout << endl << endl;

//gr.DFSHelper("12.2");

//создание вектора для координат вершины и заполнение его координатами

vector<sf::CircleShape> v(gr.size()); //создание вектора для координат вершин

vector<sf::RectangleShape> v\_edges; //создание вектора для координат ребер

vector<sf::Text> v\_text; //создание вектора для координат текста

vector<sf::Text> v\_weight; //создание вектора для координат веса ребер

vector<sf::Text> v\_dejkstra; //создание вектора для коориднат текста алгоритма Дейкстры

vector<int> for\_dejk;

//создание шрифта для данных вершин

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::White);

text.setOutlineThickness(2);

//создание шрифта для веса ребра

sf::Font font\_weight;

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

sf::Text text\_weight;

text\_weight.setFont(font\_weight);

text\_weight.setCharacterSize(18);

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;

//вызов функций, для первичного расчета постановок

cordCalculate(v, gr.size()); //функция, которая выполняет расчет расположения вершин

gr.calcCordEdge(v\_edges, v\_direction,v); //метод, который выполняет расчет координат для ребер

gr.calcCordText(v\_text, v, text); //метода, расчитывающий координаты для даннных вершины

gr.calcCordWeight(v\_weight, v, text\_weight); //метод, рассчитывающий координаты для веса ребра

size\_t temp\_size = gr.size(); //переменная, которая отвечает за перерасчет расположения ребер и вершин

sf::Color color(255, 255, 255); //цвет для окна

sf::RenderWindow window(sf::VideoMode(1000, 1000), "My Graph"); //создание окна, в котором будет отрисовываться граф.

window.setFramerateLimit(30); // Ограничение на 30 fps

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 = "---------------MENU--------------\n1. Добавить вершину\n2. Удалить вершину\n3. Добавить ненаправленное ребро\n4. Добавить направленное ребро\n5. Удалить ребро\n6. Алгоритм Дейкстры\n7. Задача комивояжера\n8. Обход в ширину\n9. Обход в глубину\n";

/\*"---------------MENU--------------\n

1. Добавить вершину\n

2. Удалить вершину\n

3. Добавить ненаправленное ребро\n

4. Добавить направленное ребро\n

5. Удалить ребро\n

6. Алгоритм Дейкстры\n

7. Задача комивояжера\n"

8. Обход в ширину

9. Обход в глубину

0. Выход.\*/

while (true)

{

cout << menu << endl << "> ";

cin >> menu\_control;

if (!window.isOpen())

return;

switch (menu\_control)

{

case 1:

cout << "Введите данные для вершины: " << endl << "> "; cin >> data;

gr.addVertex(data);

upravlenie = true;

break;

case 2:

cout << "Введите данные вершины для удаления: " << endl << "> "; cin >> data;

gr.removeVertex(data);

upravlenie = true;

break;

case 3:

cout << "Введите 2 вершины, которые необходимо соединить: " << endl << "> "; cin >> data;

cout << "> "; cin >> data2;

cout << "Введите вес ребра: " << endl << "> "; cin >> weight;

gr.addEdge(data, data2, weight);

upravlenie = true;

break;

case 4:

cout << "Введите 2 вершины, которые необходимо соединить (start -> finish): " << endl << "> "; cin >> data;

cout << "> "; cin >> data2;

cout << "Введите вес ребра: " << endl << "> "; cin >> weight;

gr.addDirectEdge(data, data2, weight);

upravlenie = true;

break;

case 5:

cout << "Введите 2 вершины, ребро между которыми необходимо удалить: " << endl << "> "; cin >> data;

cout << "> "; cin >> data2;

gr.removeEdge(data, data2);

upravlenie = true;

break;

case 6:

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 7:

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

vector\_my = gr.getVertexData();

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

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

{

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

system("pause");

break;

}

//gr.clearPath();

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

lst = gr.getPath();

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

{

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

cout << "Попробуйте изменить граф." << endl << endl;

system("pause");

}

else

{

cout << endl << "Путь комивояжера:" << endl;

for (auto& i : lst)

{

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

}

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

cout << "Цена пути: " << gr.getPathWeight() << endl << endl;

salesman = true;

system("pause");

salesman = 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;

}

});

//главный цикл отрисовки окна с графом. Пока окно открыто, оно отрисовывается

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; //вектор для отметки пройденных вершин

vector<sf::CircleShape> v\_path; //вектор для отрисовки пройденного пути

v\_path.reserve(800);

sf::CircleShape trailShape(3.0f);

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

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

//создание спрайта (машинки)

sf::Image image;

image.loadFromFile("images/car\_sprite.png");

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

sf::Texture texture\_car;

texture\_car.loadFromImage(image);

sf::Sprite car\_sprite;

car\_sprite.setTexture(texture\_car); //устанавливаем текстуру

car\_sprite.setScale(0.7f, 0.7f); //устанавливаем размер спрайта

car\_sprite.setOrigin(car\_sprite.getLocalBounds().width / 2.0f, car\_sprite.getLocalBounds().height / 2.0f);

//дополнительные переменные для отрисовки

const float animationSpeed = 300.0f; // Скорость перемещения спрайта (пикселей в секунду)

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;

//получение нового пути

current\_local = next\_local;

lst\_temp.pop\_front();

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

//создание текстуры для отмеки пути

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

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

v\_point[i - 1].setOutlineThickness(6);

//установка спрайта

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

car\_sprite.setPosition(v[current\_local].getPosition());

car\_sprite.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;

//отрисовка спрайта и экрана

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(car\_sprite);

window.display();

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

float t = elapsedTime / travelTime;

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

trailShape.setPosition(car\_sprite.getPosition());

v\_path.push\_back(trailShape);

car\_sprite.setPosition(currentPosition);

travelTime = totalDistance / animationSpeed;

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

}

}

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

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

v\_point.back().setOutlineThickness(6);

if (salesman)

{

elapsedTime = 0.0f;

lst\_temp = gr.getPath();

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

car\_sprite.setPosition(v[lst\_temp.back()].getPosition());

car\_sprite.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(car\_sprite);

window.display();

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

float t = elapsedTime / travelTime;

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

trailShape.setPosition(car\_sprite.getPosition());

v\_path.push\_back(trailShape);

car\_sprite.setPosition(currentPosition);

travelTime = totalDistance / animationSpeed;

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

}

while (salesman)

{

sf::sleep(sf::milliseconds(33)); // Задержка на ~30 кадров в секунду

}

}

}

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(33)); // Задержка на ~30 кадров в секунду

}

return 0;

}

//функция рассчиывающая расстановку 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::White);

circle.setOutlineThickness(3);

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

float radius = 400.f; //радиус круга

sf::Vector2f center = { 500.f, 500.f }; //Место где будет находится центр круга

float currentAngle = 0.f; //поворот конкретной элемента

float angleStep = 360.f / new\_size; //щаг поворота относительно круга

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);

}

Graph.h:

#pragma once

#include "header.h"

template <typename T>

class graph

{

public:

graph();

~graph();

size\_t size() { return this->\_size; } //количество вершин в графе

void addVertex(T data); //метод добавляет вершину в граф

void addEdge(T data\_v1, T data\_v2, int weight); //добавление ребра по ключу

void addEdge(size\_t v1, size\_t v2, int weight); //данный метод добавляет ребро между вершинами

void addDirectEdge(T v\_start, T v\_finish, int weight); //добавление направленного ребра, между вершинами

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

void removeVertex(T data); //удаление вершины, по ключу

void removeEdge(T data\_v1, T data\_v2); //удаление ребра по ключу, ключ - значение вершины

void removeEdge(size\_t v1, size\_t v2); //удаление ребра по индексу вершины

vector<T> getVertexData() { return this->data; } //метод возвращает вектор с данными

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

void calcCordText(vector<sf::Text>& v\_text, const vector<sf::CircleShape>& vert, sf::Text& text); //рассчет для отрисовки данных в вершине

void calcCordWeight(vector<sf::Text>& v\_weight, const vector<sf::CircleShape>& vert, sf::Text& text); //расчет для отрисовки веса ребра

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

//решение задачи комивояжера

void SalesmanTravel(const int& start);

list<int> getPath() { return this->pathh; }

int getPathWeight() { return this->pathWeight; }

void clearPath() { this->pathh.clear(); }

vector<int> dijkstra(size\_t start); //реализация алгоритма дейкстры.

void BFS(const int& v\_start); //алгоритм обхода в ширину.

void DFS(const int& v\_start); //алгоритм обхода в глубину.

bool BFSHelper(T data);

bool DFSHelper(T data);

private:

//методы для решению задачи комивояжера

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; //количество верщин в графе

vector<T> data; //массив с данными

vector<vector <int>> matrix; //матрица смежности

list<int> pathh; //список с путем

};

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>

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

{

size\_t i = 0;

vector<bool> visited(\_size, false); // массив посещенных вершин

stack<int> st; // очередь для обхода в ширину

visited[v\_start] = true; // помечаем начальную вершину как посещенную

st.push(v\_start); // добавляем начальную вершину в очередь

while (!st.empty())

{

auto current = st.top(); // берем текущую вершину из очереди

st.pop();

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

// Получаем смежные вершины и добавляем их в очередь, если они еще не посещены

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); // массив посещенных вершин

queue<int> queue; // очередь для обхода в ширину

visited[v\_start] = true; // помечаем начальную вершину как посещенную

queue.push(v\_start); // добавляем начальную вершину в очередь

while (!queue.empty())

{

auto current = queue.front(); // берем текущую вершину из очереди

queue.pop();

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

// Получаем смежные вершины и добавляем их в очередь, если они еще не посещены

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]];

/\*std::cout << this->vertexList[p[i]] << ' ';\*/

}

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>

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>

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); //добавление элемента в массив, где хранятся данные

++this->\_size;

vector<int> vec(this->data.size(), 0); //создание временного массива, для того, чтобы запушить его в матрицу

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

for (auto& i : this->matrix) //добавление к уже имеющимся массивам в матрице еще одну ячейку с нулями, так как

i.push\_back(0); //ребер к новой вершине нет. (добавление в матрицу еще одного столбца)

this->matrix.push\_back(vec); //добавляем массив в матрицу ( строку )

}

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) //параметрами в данном методе являются индексы в массиве с данными

{

if (v1 == v2) return; //не будет создано петли

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; //не будет создано петли

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;

//this->data.erase(remove\_if(this->data.begin(), this->data.end(), [data](T s) {return data == s;}), this->data.end());

}

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; //длина прямоуголника и угол поворота

float width, height;

//создание прототипов треугольника, для стрелки

sf::ConvexShape triangle\_trash(3); // создаем многоугольник с 3 вершинами (треугольник)

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); // создаем многоугольник с 3 вершинами (треугольник)

triangle.setPoint(0, sf::Vector2f(0, 0)); // задаем координаты первой вершины

triangle.setPoint(1, sf::Vector2f(25, 0)); // задаем координаты второй вершины

triangle.setPoint(2, sf::Vector2f(14, 25)); // задаем координаты третьей вершины

triangle.setFillColor(sf::Color::Black); // задаем цвет заливки

triangle.setOutlineThickness(0); // задаем толщину границы

float x, y, k, b, t;

sf::Vector2f center1;

sf::Vector2f center2;

sf::Vector2f temp; //временная пара координат

size\_t row = 0, column = 1; //индексы колонок и строк для итерации по матрице смежности

//короче тут такая байда, что мне необходимо сейчас правильно расположить треугольники, поэтому вся поебота с ними

//надо как то грамотно это реализовать, поэтому нужно заняться этим.

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

{

while (column < this->\_size)

{

if (this->matrix[row][column] == 0 && this->matrix[column][row] == 0) //если ребра между вершинами нет

++column;

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

{

temp = vert[column].getPosition() - vert[row].getPosition(); //вычисляем вектор, направленный от point1 к point2

length = sqrt(pow(temp.x, 2) + pow(temp.y, 2)); //вычисляем длину вектора temp

angle = atan2(temp.y, temp.x) \* 180 / 3.14159; //вычисляем угол между осью X и вектором temp и переводим радианы в градусы

sf::RectangleShape line(sf::Vector2f(length, 3.f)); //создаем прямоуголник шириной 3 пикселя, и вычисленной длиной

line.setPosition(vert[row].getPosition()); // устанавливаем начальную точку линии

line.setRotation(angle); // поворачиваем прямоугольник на нужный угол

line.setFillColor(sf::Color::Black); // задаем цвет линии

edges.push\_back(line); //добавляем ребро в вектор

if (this->matrix[row][column] == 0) //если ребро направленное

{

center1 = vert[column].getPosition(); //берем центры окружностей

center2 = vert[row].getPosition();

k = (center2.y - center1.y) / (center2.x - center1.x); //находим угловой кожфициент данной прямой

b = center1.y - k \* center1.x; //находим свободный коэффициент прямой

t = 1 - (55 / length); //расчитываем коэфициент смещения (подобран опытным путем)

x = center1.x + t \* (center2.x - center1.x); //находим x и y для треуголника (стрелки)

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

width = triangle.getLocalBounds().width;

height = triangle.getLocalBounds().height;

triangle.setOrigin(width / 2.0f, height / 2.0f);//устанавливаем отсчета треугоолника от середины

triangle.setRotation(angle + 90); //поворачиваем треугольник на нужный угол + поправка, так как треуголник отрисовывается изначально перевернутым

triangle.setPosition(x, y); //устанавливаем позицию

v\_direction.push\_back(triangle); //добавляем в вектор для последующей отрисовки

}

else

{

v\_direction.push\_back(triangle\_trash); //если ребро не направленное, то просто добавляем треугольник в конец вектора (для простоты отрисовки)

}

++column;

}

else //здесь все аналогично, только работаем с верхней частью матрицы

{

temp = vert[row].getPosition() - vert[column].getPosition(); //вычисляем вектор, направленный от point1 к point2

length = sqrt(pow(temp.x, 2) + pow(temp.y, 2)); //вычисляем длину вектора temp

angle = atan2(temp.y, temp.x) \* 180 / 3.14159; //вычисляем угол между осью X и вектором temp и переводим радианы в градусы

sf::RectangleShape line(sf::Vector2f(length, 3.f)); //создаем прямоуголник шириной 3 пикселя, и вычисленной длиной

line.setPosition(vert[column].getPosition()); // устанавливаем начальную точку линии

line.setRotation(angle); // поворачиваем прямоугольник на нужный угол

line.setFillColor(sf::Color::Black); // задаем цвет линии

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;

}

}

//Попробовать передавать объект Text по ссылке, оставить только цикл в методе

template<typename T>

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

{

float width, height; //переменные для установки точки отсчета от середины

v\_text.clear();

string str;

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

{

str = this->data[i];

text.setString(str);

//text.setString(sf::String::fromUtf8(str.begin(), str.end())); //устанавливаем текст

width = text.getLocalBounds().width;

height = text.getLocalBounds().height;

text.setOrigin(width / 2.0f, height / 2.0f); //устанавливаем точку отрисовки в центр

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

v\_text.push\_back(text);

}

}

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; //переменные для установки точки отсчета от середины

v\_text.clear();

string str;

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

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); //устанавливаем точку отрисовки в центр

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

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, height;

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) //если ребра между вершинами нет

++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>

graph<T>::~graph()

{

this->\_size = 0;

this->data.clear();

this->matrix.clear();

}

Header.h;

#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:

