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

Темы: «АРМ маркетолога»

«Задача Коммивояжора»

Выполнил

Студент группы РИС-22-2б

Зубов Р. А.

Проверил доц. Кафедры ИТАС

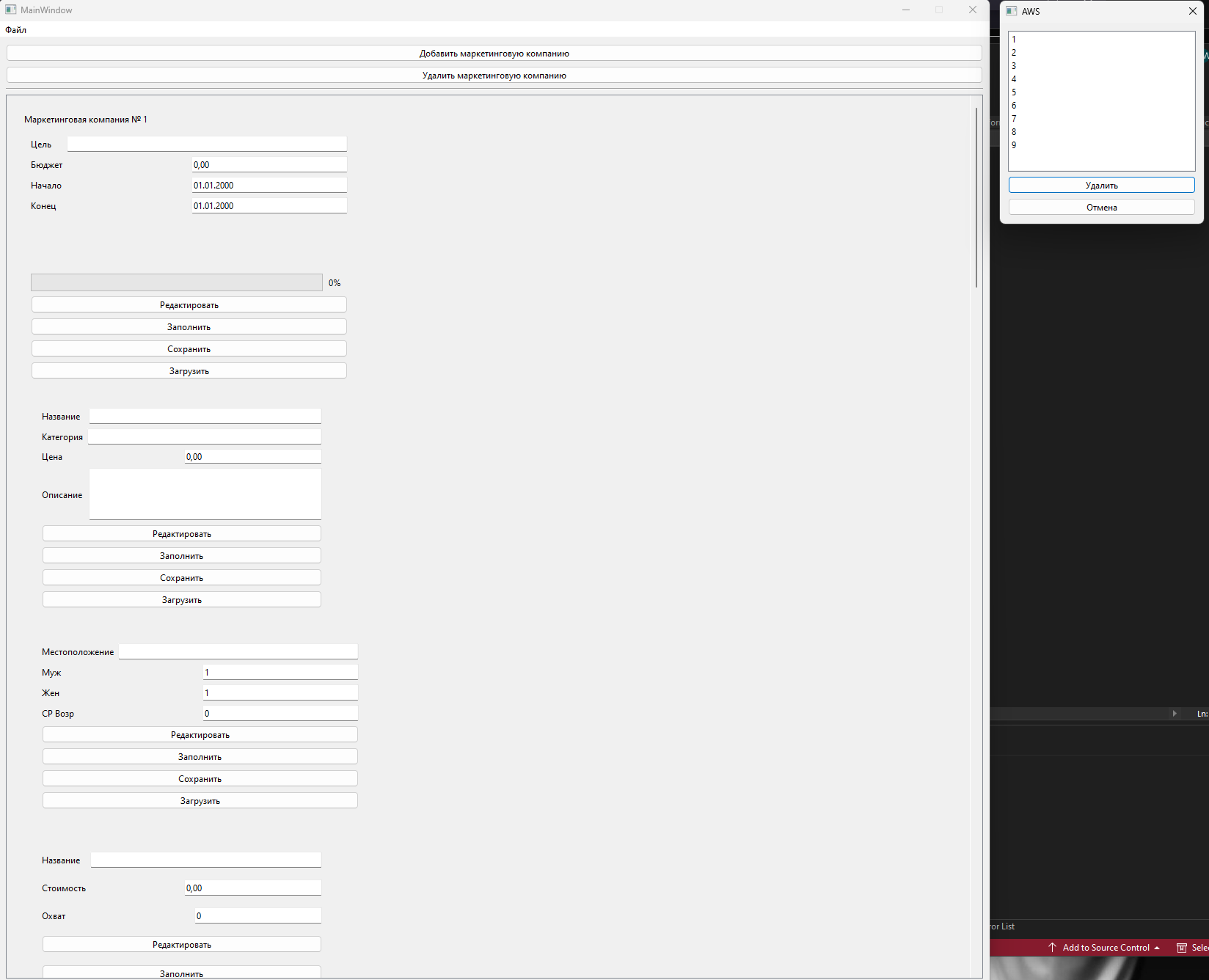
Полякова О. А.

Пермь 2023

**АРМ**

**Постановка задачи**

Реализовать АРМ маркетолога (внесение данных о продукте, целевой аудитории, каналах продвижения; управление маркетинговыми кампаниями) с графическим интерфейсом.



Программа позволяет создавать, редактировать и удалять маркетинговые компании по средствам динамических вызовов виджетов. В созданной маркетинговой компании можно заполнять необходимые данные о целях и сроках маркетинговой компании, с отображением оставшегося времени, информацию о продукте: его название, цену, категорию, описание; информацию о аудитории распространения: местоположение, количество мужчин и женщин, их средний возраст; а так же информацию о каналах распространения: название, стоимость, охват.

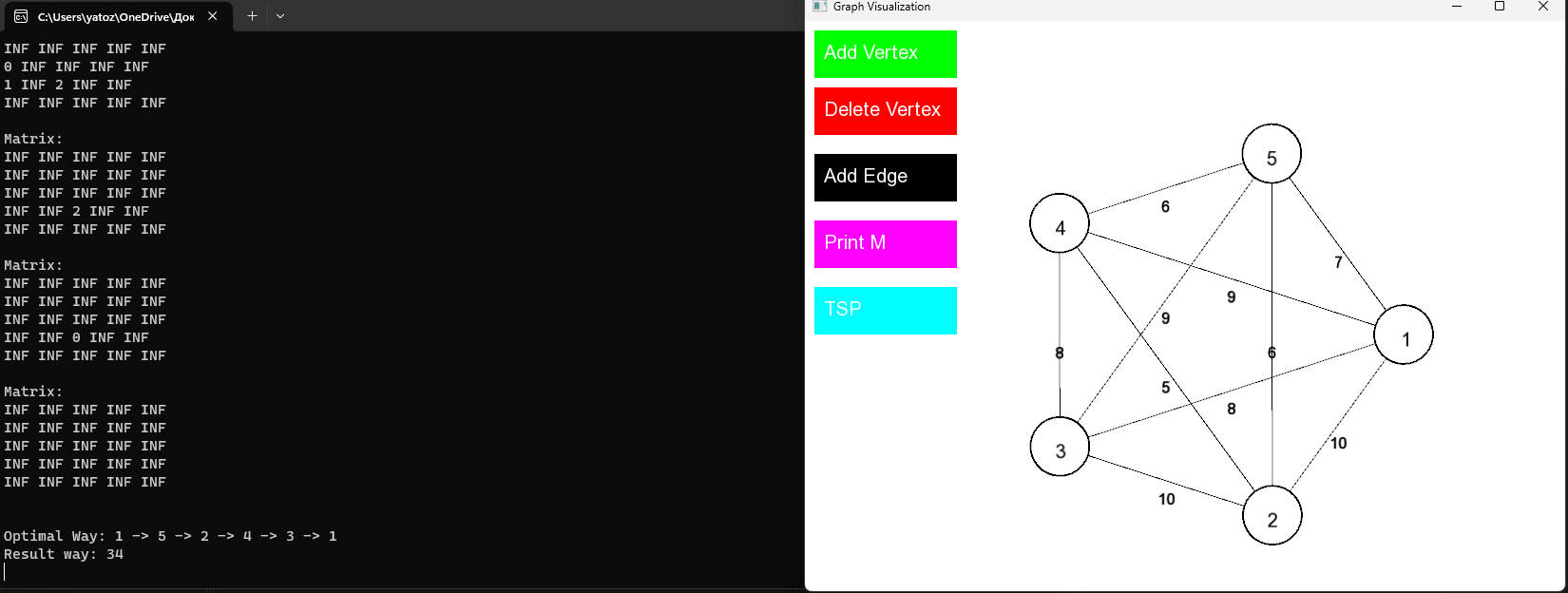
Также предусмотрен функционал сохранения форм по отдельным XML файлам, а так же загрузка информации оттуда.

Представленная программа была написана на фреймворке Qt.

**Задача Коммивояжора**

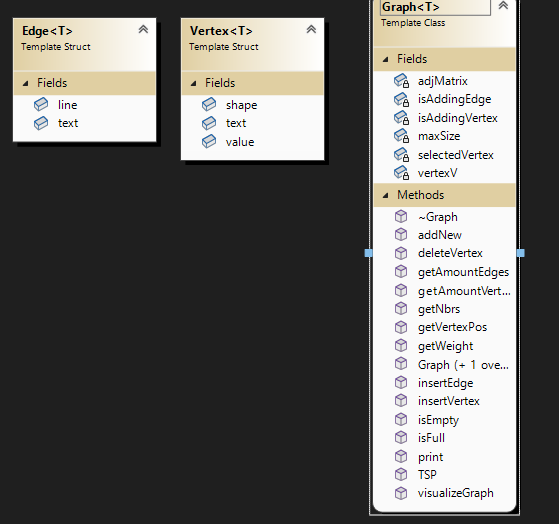
**Постановка задачи**

Реализовать алгоритм Ветвей и Границ для решения задачи Коммивояжора с графическим интерфейсом.



Визуализация реализована посредствам библиотеки SFML.

В программе предусмотрены функции добавления и удаления вершин, ребер, печать матрицы смежности, а также решение задачи Коммивояжора.



#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->vertexV.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 TSP();

void visualizeGraph(Graph<T>& graph);

void addNew(int newVertex);

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

};

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);

int newSize = this->vertexV.size();

maxSize++;

// Увеличение размера матрицы

this->adjMatrix.resize(newSize, std::vector<int>(newSize, 0));

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

{

this->adjMatrix[i].resize(newSize, 0);

}

}

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;

}

void printMatrix(int\*\*\* matrix, int n) {

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

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

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

if (matrix[i][j] == nullptr)

std::cout << "INF ";

else

std::cout << \*matrix[i][j] << " ";

}

std::cout << std::endl;

}

std::cout << std::endl;

}

void Reduction(int\*\*\* matrix, int VN)

{

int min = 100000;

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

{

for (int j = 0; j < VN; j++)

if (matrix[i][j] && min > \*matrix[i][j])

min = \*matrix[i][j];

for (int j = 0; j < VN; j++) // Редукция строк

if (matrix[i][j])

\*matrix[i][j] -= min;

}

for (int j = 0; j < VN; j++)

{

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

if (matrix[i][j] && min > \*matrix[i][j])

min = \*matrix[i][j];

for (int i = 0; i < VN; i++) // Редукция столбцов

if (matrix[i][j])

\*matrix[i][j] -= min;

}

}

void BranchAndBound(int\*\*\* matrix, int VN, int\*\* zeroComp, int\* wayop)

{

for (int l = 0; l < VN; l++)

{

Reduction(matrix, VN);

printMatrix(matrix, VN);

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

for (int j = 0; j < VN; j++)

zeroComp[i][j] = 0;

for (int i = 0; i < VN; i++) // Нахождение оценок для нулевых элементов

for (int j = 0; j < VN; j++)

{

if (matrix[i][j] && !\*matrix[i][j])

{

int TMPmin = 100000;

int TMP2min = 100000;

for (int l = 0; l < VN; l++)

if (l != i && matrix[l][j] && TMPmin > \*matrix[l][j])

TMPmin = \*matrix[l][j];

for (int l = 0; l < VN; l++)

if (l != j && matrix[i][l] && TMP2min > \*matrix[i][l])

TMP2min = \*matrix[i][l];

zeroComp[i][j] = TMPmin + TMP2min; // Оценка

}

}

int mcost = 0, mi = 0, mj = 0;

for (int i = 0; i < VN; i++) // Выбор нулевого элемента с наибольшей оценкой

for (int j = 0; j < VN; j++)

if (matrix[i][j] && mcost < zeroComp[i][j])

{

mcost = zeroComp[i][j];

mi = i;

mj = j;

}

wayop[mi] = mj;

for (int i = 0; i < VN; i++) // Редукция матрицы по столбцам

matrix[i][mj] = nullptr;

for (int i = 0; i < VN; i++) // Редукция матрицы по строкам

matrix[mi][i] = nullptr;

matrix[mj][mi] = nullptr; // Убираемм нулевой элемент

printMatrix(matrix, VN);

}

}

template<typename T>

void Graph<T>::TSP()

{

int vertexNum = this->vertexV.size();

int\*\* zeroComp = new int\* [vertexNum];

int\* wayop = new int[vertexNum];

int\*\*\* matrix = new int\*\* [vertexNum];

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

{

zeroComp[i] = new int[vertexNum];

}

for (int i = 0; i < vertexNum; i++) // Копия матрицы смежности для последующих редукций

{

matrix[i] = new int\* [vertexNum];

for (int j = 0; j < vertexNum; j++)

{

if (adjMatrix[i][j] == 0) {

matrix[i][j] = nullptr;

continue;

}

matrix[i][j] = new int(adjMatrix[i][j]);

}

}

int s = 0;

int j;

BranchAndBound(matrix, vertexNum, zeroComp, wayop);

std::cout << "\nOptimal Way: ";

int tmp = 0;

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

{

j = wayop[i];

if (!adjMatrix[i][j])

{

std::cout << "Error with Vertex\n";

break;

}

s += adjMatrix[i][j];

}

for (int l = 0; l < vertexNum;)

{

for (int i = 0, j = 0; i < vertexNum; i++)

{

if (tmp == 0 || i + 1 == tmp)

{

if (tmp == 0) std::cout << i + 1;

j = wayop[i];

tmp = j + 1;

if (tmp > 0) std::cout << " -> " << tmp;

l++;

}

}

}

std::cout << "\nResult way: " << s;

std::cout << std::endl;

}

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");

// Кнопки для создания и удаления вершин

sf::RectangleShape addButton(sf::Vector2f(150, 50));

addButton.setPosition(10.0f, 10.0f);

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

sf::Text addButtonText("Add Vertex", font, 20);

addButtonText.setFillColor(sf::Color::White);

addButtonText.setPosition(addButton.getPosition() + sf::Vector2f(10.0f, 10.0f));

sf::RectangleShape deleteButton(sf::Vector2f(150, 50));

deleteButton.setPosition(10.0f, 70.0f);

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

sf::Text deleteButtonText("Delete Vertex", font, 20);

deleteButtonText.setFillColor(sf::Color::White);

deleteButtonText.setPosition(deleteButton.getPosition() + sf::Vector2f(10.0f, 10.0f));

// Кнопка для создания вершины

sf::RectangleShape addEdgeButton(sf::Vector2f(150, 50));

addEdgeButton.setPosition(sf::Vector2f(10.0f, 140.0f));

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

sf::Text addEdgeButtonText("Add Edge", font, 20);

addEdgeButtonText.setFillColor(sf::Color::White);

addEdgeButtonText.setPosition(addEdgeButton.getPosition() + sf::Vector2f(10.0f, 10.0f));

//Кнопка для печати матрицы смежности

sf::RectangleShape addPrintMatrixButton(sf::Vector2f(150, 50));

addPrintMatrixButton.setPosition(10.0f, 210.0f);

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

sf::Text addPrintMatrixButtonText("Print M", font, 20);

addPrintMatrixButtonText.setFillColor(sf::Color::White);

addPrintMatrixButtonText.setPosition(addPrintMatrixButton.getPosition() + sf::Vector2f(10.0f, 10.0f));

//TSP button

sf::RectangleShape addTSPButton(sf::Vector2f(150, 50));

addTSPButton.setPosition(10.0f, 280.0f);

addTSPButton.setFillColor(sf::Color::Cyan);

sf::Text addTSPButtonText("TSP", font, 20);

addTSPButtonText.setFillColor(sf::Color::White);

addTSPButtonText.setPosition(addTSPButton.getPosition() + sf::Vector2f(10.0f, 10.0f));

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);

}

}

T newVertex;

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

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

window.close();

}

else if (event.type == sf::Event::MouseButtonPressed) {

if (event.mouseButton.button == sf::Mouse::Left) {

sf::Vector2f mousePos = sf::Vector2f(event.mouseButton.x, event.mouseButton.y);

if (addTSPButton.getGlobalBounds().contains(mousePos))

{

TSP();

}

if (addPrintMatrixButton.getGlobalBounds().contains(mousePos))

{

print();

}

if (addEdgeButton.getGlobalBounds().contains(mousePos)) {

int weight, Vertex1, Vertex2;

std::cout << "Choose Vertex 1: "; std::cin >> Vertex1;

std::cout << "Choose Vertex 2: "; std::cin >> Vertex2;

std::cout << "Weight for new: "; std::cin >> weight;

insertEdge(Vertex1, Vertex2, weight);

window.close();

visualizeGraph(graph);

}

if (addButton.getGlobalBounds().contains(mousePos)) {

newVertex = this->getAmountVertexes() + 1;

graph.insertVertex(newVertex);

window.close();

visualizeGraph(graph);

}

else if (deleteButton.getGlobalBounds().contains(mousePos) && vertexV.size() != 0) {

graph.deleteVertex();

window.close();

visualizeGraph(graph);

}

}

}

}

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

window.draw(addButton);

window.draw(addButtonText);

window.draw(deleteButton);

window.draw(deleteButtonText);

window.draw(addEdgeButton);

window.draw(addEdgeButtonText);

window.draw(addPrintMatrixButton);

window.draw(addPrintMatrixButtonText);

window.draw(addTSPButton);

window.draw(addTSPButtonText);

drawGraph(window, graph, vertices, edges);

}

}