Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«Пермский национальный исследовательский**

**политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**О Т Ч Ё Т**

**по творческой работе**

Дисциплина: Основы алгоритмизации и программирования

Тема: Задача Коммивояжера

Выполнил работу

студент группы РИС-22-1б

Рыжков Н.С.

Проверил

Доцент кафедры ИТАС

Полякова О.А.

Пермь, 2023

**Описание задачи**

Необходимо разработать программу для решения задачи коммивояжера на языке C++. Программа должна позволять добавлять и удалять вершины, добавлять, редактировать и удалять ребра, а также визуализировать целый граф и оптимальный маршрут с помощью библиотеки SFML.

Функциональные требования

1. Добавление вершин:

- Пользователь должен иметь возможность добавлять вершины в граф.

2. Удаление вершин:

- Пользователь должен иметь возможность удалять вершины из графа.

- При удалении вершины все связанные с ней ребра также должны быть удалены.

3. Добавление ребер:

- Пользователь должен иметь возможность добавлять ребра между вершинами.

- При добавлении ребра пользователь должен указывать вес ребра.

4. Редактирование ребер:

- Пользователь должен иметь возможность изменять вес ребра.

5. Удаление ребер:

- Пользователь должен иметь возможность удалять ребра из графа.

6. Визуализация графа:

- Пользователь должен иметь возможность просматривать граф на экране.

- Каждая вершина должна быть отображена на экране в виде круга с меткой.

- Каждое ребро должно быть отображено на экране в виде линии с весом ребра.

7. Решение задачи коммивояжера:

- Пользователь должен иметь возможность решить задачу коммивояжера для заданного графа.

- Оптимальный маршрут должен быть визуализирован на экране.

Нефункциональные требования

1. Язык программирования: C++.

2. Использование библиотеки SFML для визуализации графа и оптимального маршрута.

3. Код должен быть написан в соответствии с принципами объектно-ориентированного программирования.

4. Код должен быть написан с использованием стандартов языка C++11 или выше.

5. Код должен быть написан с использованием системы контроля версий Git.

**План работ**

1. Разработка методов класса для представления вершины графа.

2. Разработка методов класса для представления ребра графа.

3. Разработка методов класса для представления графа.

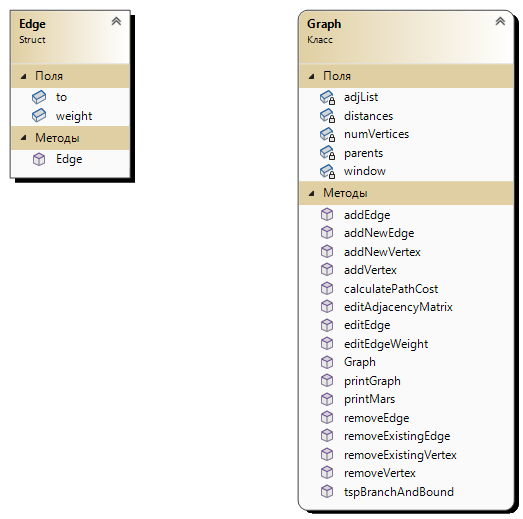
4. Разработка методов класса для решения задачи коммивояжера.

5. Разработка методов класса для визуализации графа и оптимального маршрута с помощью библиотеки SFML.

6. Разработка пользовательского интерфейса.

7. Тестирование программы.

**Диаграмма классов**

****

**Листинг программы**

#include <SFML/Graphics.hpp>

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <limits>

#include <cmath>

#include <Windows.h>

#include <algorithm>

const int NotUsed = system("color 80");

using namespace std;

struct Edge {

int to;

int weight;

Edge(int to, int weight) : to(to), weight(weight) {}

};

class Graph {

private:

vector<vector<Edge>> adjList;

vector<int> distances;

vector<int> parents;

int numVertices;

sf::RenderWindow window;

public:

Graph(int numVertices) : numVertices(numVertices) {

adjList.resize(numVertices);

}

void addEdge(int from, int to, int weight) {

adjList[from].push\_back(Edge(to, weight));

adjList[to].push\_back(Edge(from, weight));

}

void addVertex() {

numVertices++;

adjList.resize(numVertices);

}

void removeVertex(int vertex) {

numVertices--;

adjList.erase(adjList.begin() + vertex);

for (auto& adj : adjList) {

for (auto& edge : adj) {

if (edge.to == vertex) {

edge.to = -1;

}

else if (edge.to > vertex) {

edge.to--;

}

}

}

}

void removeEdge(int from, int to) {

for (auto& edge : adjList[from]) {

if (edge.to == to) {

edge.to = -1;

}

}

}

void editEdgeWeight(int from, int to, int weight) {

for (auto& edge : adjList[from]) {

if (edge.to == to) {

edge.weight = weight;

}

}

}

void editAdjacencyMatrix(int\*\* matrix, int size) {

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

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

if (matrix[i][j] != -1) {

if (i >= size || j >= size) {

removeEdge(i, j);

}

else {

editEdgeWeight(i, j, matrix[i][j]);

}

}

else if (i < size && j < size) {

addEdge(i, j, matrix[i][j]);

}

}

}

}

void addNewVertex() {

addVertex();

cout << "Added new vertex " << numVertices - 1 << endl;

}

void removeExistingVertex() {

int vertex;

cout << "Enter vertex to remove: ";

cin >> vertex;

if (vertex < 0 || vertex >= numVertices) {

cout << "Invalid vertex" << endl;

return;

}

removeVertex(vertex);

cout << "Removed vertex " << vertex << endl;

}

void addNewEdge() {

int from, to, weight;

cout << "Enter source vertex: ";

cin >> from;

cout << "Enter destination vertex: ";

cin >> to;

cout << "Enter weight: ";

cin >> weight;

if (from < 0 || from >= numVertices || to < 0 || to >= numVertices) {

cout << "Invalid vertices" << endl;

return;

}

addEdge(from, to, weight);

cout << "Added edge from " << from << " to " << to << " with weight " << weight << endl;

}

void removeExistingEdge() {

int from, to;

cout << "Enter source vertex: ";

cin >> from;

cout << "Enter destination vertex: ";

cin >> to;

if (from < 0 || from >= numVertices || to < 0 || to >= numVertices) {

cout << "Invalid vertices" << endl;

return;

}

removeEdge(from, to);

cout << "Removed edge from " << from << " to " << to << endl;

}

void editEdge() {

int from, to, weight;

cout << "Enter source vertex: ";

cin >> from;

cout << "Enter destination vertex: ";

cin >> to;

cout << "Enter new weight: ";

cin >> weight;

if (from < 0 || from >= numVertices || to < 0 || to >= numVertices) {

cout << "Invalid vertices" << endl;

return;

}

editEdgeWeight(from, to, weight);

cout << "Changed weight of edge from " << from << " to " << to << " to " << weight << endl;

}

void printMars(const vector<int>& path={}){

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

window.setFramerateLimit(60);

sf::Color myColor(255,162,64);

sf::Color myColor2(2,78,104);

window.clear(myColor);

cout<<"Graph:"<<endl;

sf::Font font;

font.loadFromFile("Barlow-Medium.ttf");

vector<sf::CircleShape> vertices(numVertices);

vector<sf::Text> texts(numVertices);

vector<vector <sf::Vertex>> edges(numVertices,vector<sf::Vertex>());

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

vertices[i].setRadius(20);

vertices[i].setPosition(sf::Vector2f(rand()%700+50,rand()%500+50));

vertices[i].setOutlineThickness(3);

vertices[i].setOutlineColor(myColor2);

vertices[i].setFillColor(myColor);

texts[i].setFont(font);

texts[i].setString(to\_string(i));

texts[i].setCharacterSize(20);

texts[i].setFillColor(myColor2);

texts[i].setPosition(sf::Vector2f(vertices[i].getPosition().x+15,vertices[i].getPosition().y+5));

window.draw(vertices[i]);

window.draw(texts[i]);

for(auto edge:adjList[i]){

if(edge.to!=-1){

bool isPartOfPath=false;

if(!path.empty()){

for(int j=0;j<path.size()-1;++j){

if((i==path[j]&&edge.to==path[j+1])||(i==path[j+1]&&edge.to==path[j])){

isPartOfPath=true;

break;

}

}

}

if(isPartOfPath){

sf::Vector2f from(vertices[i].getPosition().x+20,vertices[i].getPosition().y+20);

sf::Vector2f to(vertices[edge.to].getPosition().x+20,vertices[edge.to].getPosition().y+20);

sf::Vertex line[]={

sf::Vertex(from,myColor2),

sf::Vertex(to,myColor2)

};

edges[i].push\_back(line[0]);

edges[i].push\_back(line[1]);

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

sf::Text weightText;

weightText.setFont(font);

weightText.setString(to\_string(edge.weight));

weightText.setCharacterSize(20);

weightText.setFillColor(myColor2);

weightText.setPosition(sf::Vector2f((from.x+to.x)/2,(from.y+to.y)/2));

window.draw(weightText);

}

}

}

}

window.display();

while(window.isOpen()){

sf::Event event;

while(window.pollEvent(event)){

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

window.close();

}

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

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

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

if(vertices[i].getGlobalBounds().contains(event.mouseButton.x,event.mouseButton.y)){

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

sf::Vector2f offset=mousePos-vertices[i].getPosition();

while(sf::Mouse::isButtonPressed(sf::Mouse::Left)){

mousePos=sf::Vector2f(sf::Mouse::getPosition(window));

vertices[i].setPosition(mousePos-offset);

texts[i].setPosition(sf::Vector2f(vertices[i].getPosition().x+15,vertices[i].getPosition().y+5));

window.clear(myColor);

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

for(auto edge:adjList[i]){

if(edge.to!=-1){

bool isPartOfPath=false;

if(!path.empty()){

for(int j=0;j<path.size()-1;++j){

if((i==path[j]&&edge.to==path[j+1])||(i==path[j+1]&&edge.to==path[j])){

isPartOfPath=true;

break;

}

}

}

if(isPartOfPath){

sf::Vector2f from(vertices[i].getPosition().x+20,vertices[i].getPosition().y+20);

sf::Vector2f to(vertices[edge.to].getPosition().x+20,vertices[edge.to].getPosition().y+20);

sf::Vertex line[]={

sf::Vertex(from,myColor2),

sf::Vertex(to,myColor2)

};

edges[i].push\_back(line[0]);

edges[i].push\_back(line[1]);

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

sf::Text weightText;

weightText.setFont(font);

weightText.setString(to\_string(edge.weight));

weightText.setCharacterSize(20);

weightText.setFillColor(myColor2);

weightText.setPosition(sf::Vector2f((from.x+to.x)/2,(from.y+to.y)/2));

window.draw(weightText);

}

}

}

window.draw(vertices[i]);

window.draw(texts[i]);

}

window.display();

}

break;

}

}

}

}

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

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

}

}

}

}

}

void printGraph(const vector<int>& path = {}) {

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

window.setFramerateLimit(60);

sf::Color myColor(255, 162, 64);

sf::Color myColor2(2, 78, 104);

window.clear(myColor);

cout << "Graph:" << endl;

sf::Font font;

font.loadFromFile("Barlow-Medium.ttf");

vector<sf::CircleShape> vertices(numVertices);

vector<sf::Text> texts(numVertices);

vector<vector<sf::Vertex>> edges(numVertices, vector<sf::Vertex>());

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

vertices[i].setRadius(20);

vertices[i].setPosition(sf::Vector2f(rand() % 700 + 50, rand() % 500 + 50));

vertices[i].setOutlineThickness(3);

vertices[i].setOutlineColor(myColor2);

vertices[i].setFillColor(myColor);

texts[i].setFont(font);

texts[i].setString(to\_string(i));

texts[i].setCharacterSize(20);

texts[i].setFillColor(myColor2);

texts[i].setPosition(sf::Vector2f(vertices[i].getPosition().x + 15, vertices[i].getPosition().y + 5));

window.draw(vertices[i]);

window.draw(texts[i]);

for (auto edge : adjList[i]) {

if (edge.to != -1) {

sf::Vector2f from(vertices[i].getPosition().x + 20, vertices[i].getPosition().y + 20);

sf::Vector2f to(vertices[edge.to].getPosition().x + 20, vertices[edge.to].getPosition().y + 20);

sf::Vertex line[] = {

sf::Vertex(from,myColor2),

sf::Vertex(to,myColor2)

};

edges[i].push\_back(line[0]);

edges[i].push\_back(line[1]);

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

sf::Text weightText;

weightText.setFont(font);

weightText.setString(to\_string(edge.weight));

weightText.setCharacterSize(20);

weightText.setFillColor(myColor2);

weightText.setPosition(sf::Vector2f((from.x + to.x) / 2, (from.y + to.y) / 2));

window.draw(weightText);

}

}

}

if (!path.empty()) {

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

sf::Vector2f from(vertices[path[i]].getPosition().x + 20, vertices[path[i]].getPosition().y + 20);

sf::Vector2f to(vertices[path[i + 1]].getPosition().x + 20, vertices[path[i + 1]].getPosition().y + 20);

sf::Vertex line[] = {

sf::Vertex(from, myColor2),

sf::Vertex(to, myColor2)

};

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

}

}

window.display();

while (window.isOpen()) {

sf::Event event;

while (window.pollEvent(event)) {

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

window.close();

}

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

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

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

if (vertices[i].getGlobalBounds().contains(event.mouseButton.x, event.mouseButton.y)) {

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

sf::Vector2f offset = mousePos - vertices[i].getPosition();

while (sf::Mouse::isButtonPressed(sf::Mouse::Left)) {

mousePos = sf::Vector2f(sf::Mouse::getPosition(window));

vertices[i].setPosition(mousePos - offset);

texts[i].setPosition(sf::Vector2f(vertices[i].getPosition().x + 15, vertices[i].getPosition().y + 5));

window.clear(myColor);

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

for (auto edge : adjList[i]) {

if (edge.to != -1) {

sf::Vector2f from(vertices[i].getPosition().x + 20, vertices[i].getPosition().y + 20);

sf::Vector2f to(vertices[edge.to].getPosition().x + 20, vertices[edge.to].getPosition().y + 20);

sf::Vertex line[] = {

sf::Vertex(from,myColor2),

sf::Vertex(to,myColor2)

};

edges[i].push\_back(line[0]);

edges[i].push\_back(line[1]);

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

sf::Text weightText;

weightText.setFont(font);

weightText.setString(to\_string(edge.weight));

weightText.setCharacterSize(20);

weightText.setFillColor(myColor2);

weightText.setPosition(sf::Vector2f((from.x + to.x) / 2, (from.y + to.y) / 2));

window.draw(weightText);

}

}

window.draw(vertices[i]);

window.draw(texts[i]);

}

window.display();

}

break;

}

}

}

}

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

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

}

}

}

}

}

int calculatePathCost(const vector<int>& path) {

int cost = 0;

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

for (const auto& edge : adjList[path[i]]) {

if (edge.to == path[i + 1]) {

cost += edge.weight;

break;

}

}

}

return cost;

}

void tspBranchAndBound(int startVertex) {

vector<int> vertices;

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

if (i != startVertex) {

vertices.push\_back(i);

}

}

int min\_cost = numeric\_limits<int>::max();

vector<int> best\_path;

do {

vector<int> path = { startVertex };

path.insert(path.end(), vertices.begin(), vertices.end());

path.push\_back(startVertex);

int path\_cost = calculatePathCost(path);

if (path\_cost < min\_cost) {

min\_cost = path\_cost;

best\_path = path;

}

} while (next\_permutation(vertices.begin(), vertices.end()));

cout << "Minimum cost:" << min\_cost << endl;

cout << "Path:";

for (int vertex : best\_path) {

cout << vertex << " ";

}

cout << endl;

// Print the graph with only the edges in the best path

printMars(best\_path);

}

};

int main() {

HANDLE hConsole = GetStdHandle(STD\_OUTPUT\_HANDLE);

CONSOLE\_FONT\_INFOEX fontInfo = { sizeof(fontInfo) };

GetCurrentConsoleFontEx(hConsole, FALSE, &fontInfo);

fontInfo.dwFontSize.Y = 24;

SetCurrentConsoleFontEx(hConsole, FALSE, &fontInfo);

int numVertices;

cout << "Enter number of vertices: ";

cin >> numVertices;

Graph g(numVertices);

int choice;

do {

cout << "1. Add vertex" << endl;

cout << "2. Remove vertex" << endl;

cout << "3. Add edge" << endl;

cout << "4. Remove edge" << endl;

cout << "5. Edit edge weight" << endl;

cout << "6. Print graph" << endl;

cout << "7. Branch And Bound" << endl;

cout << "0. Exit" << endl;

cout << "Enter choice: ";

cin >> choice;

switch (choice) {

case 1:

g.addNewVertex();

break;

case 2:

g.removeExistingVertex();

break;

case 3:

g.addNewEdge();

break;

case 4:

g.removeExistingEdge();

break;

case 5:

g.editEdge();

break;

case 6:

g.printGraph();

break;

case 7:

int tspStart;

cout << "Enter starting vertex: ";

cin >> tspStart;

g.tspBranchAndBound(tspStart);

break;

case 0:

break;

default:

cout << "Invalid choice" << endl;

break;

}

} while (choice != 0);

return 0;

}