Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное учреждение высшего образования

«Пермский национальный исследовательский политехнический университет»

ПНИПУ

**Лабораторная работа**

**«Графы»**

Выполнила:

студентка группы ИВТ-23-2Б

Сычева Евгения Андреевна

Проверила:

Доцент кафедры ИТАС

О. А. Полякова

2024 г.

**Постановка задачи:**

1. Реализовать алгоритмы для собственного, придуманного самим автором, варианта двунаправленного графа, имеющего не менее 6 вершин.

2. Алгоритмы:

* Обход в ширину.
* Обход в глубину.
* Алгоритм Дейкстры.

3. Необходимо реализовать функции для редактирования графа:

- Создание новой вершины.

- Удаление вершины.

- Добавление и удаление ребра.

- Редактирование весов ребер.

- Редактирование матрицы смежности (или инцидентности в зависимости от реализации).

- Реализовать вывод графа.

**Вариант 20:** Реализовать граф, а также Флойда в соответствии с вариантом, построить все необходимые матрицы.

**Анализ задачи:**

1. Необходимо создать класс Graph со значениями:

maxSize – максимальный размер матрицы

adjMatrix – матрица

vertList – список вершин

VertsQueue – очередь из вершин

labeList – список ребер

2. Необходимо прописать для класса следующие функции:

* Graph<T>(const int& Vsize) - Конструктор, который инициализирует значения объектов класса Graph
* ~Graph() – Деструктор
* GetVertPos(const T& vertex) - Получает индекс вершин
* IsEmpty() - Проверка, является ли граф пустым
* IsFull() - Проверка, заполнен ли граф
* GetAmountVerts() - Возвращает количество вершин
* GetAmountEdges() - Получение количества ребер
* GetWeight(const T& vertex1, const T& vertex2) - Возвращает веса между вершинами
* GetNbrs(const T& vertex) - Получение вектора соседей
* InsertVertex(const T& vertex) - Добавление вершины
* InsertEdge(const T& vertex1, const T& vertex2, int weight = 1) - Вставка ребра
* Print() - Печать матрицы смежности графа
* Front() - Возвращает первый элемент в очереди
* removeVertex(const T& vertex) - Удаление вершины
* removeEdge(const T& vertex1, const T& vertex2) - Удаление ребра
* editEdgeWeight(const T& vertex1, const T& vertex2, int newWeight) - Измена веса между ребрами
* obhGlub(T& startVertex, bool\* visitedVerts) - Обход графа в глубину
* obhShir(T& startVertex, bool\* visitedVerts) - Обход графа в ширину
* FillLabels(T& startVertex)
* AllVisited(vector<bool>& visitedVerts)
* Dikstra(const T& startVertex) – Алгоритм Дейкстры
* void Floida() – Алгоритм Флойды
* void printPath(int parent[], int i, int j)

**Код программы на языке С++:**

Graph.h:

#pragma once

#include <vector>

#include <iostream>

#include <queue>

#include <string>

using namespace std;

template<class T>

class Graph

{

private:

int maxSize;

vector <vector<int>> adjMatrix;

vector<T> vertList;

queue<T> VertsQueue;

vector<T> labelList;

public:

inline Graph<T>(const int& Vsize)// Конструктор, который инициализирует значения объектов класса Graph

{

this->maxSize = Vsize;

this->adjMatrix = vector<vector<T>>(Vsize, vector<T>(Vsize));

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

{

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

{

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

}

}

}

~Graph()// Деструктор

{

}

inline int GetVertPos(const T& vertex)// Получает индекс вершин

{

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

{

if (this->vertList[i] == vertex)

{

return i;

}

}

return -1;

}

inline bool IsEmpty()// Проверка, является ли граф пустым

{

return this->vertList.size() == 0;

}

inline bool IsFull()// Проверка, заполнен ли граф

{

return this->vertList.size() == this->maxSize;

}

inline int GetAmountVerts()// Возвращает количество вершин

{

return this->vertList.size();

}

int GetAmountEdges()// Получение количества ребер

{

int amount = 0;

if (!this->IsEmpty())

{

for (int i = 0, vertListSize = this->vertList.size(); i < vertListSize; i++)

{

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

{

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

{

amount++;

}

}

}

}

return amount / 2;

}

inline int GetWeight(const T& vertex1, const T& vertex2)// Возвращает веса между вершинами

{

if (!this->IsEmpty())

{

return 0;

}

int vert\_1 = this->GetVertPos(vertex1);

int vert\_2 = this->GetVertPos(vertex2);

if (vertex1 == -1 || vertex2 == -1)

{

cout << "Одного из выбранных узлов в графе не существует." << endl;

return 0;

}

return this->adjMatrix[vert\_1][vert\_2];

}

vector<T> GetNbrs(const T& vertex)// Получение вектора соседей

{

vector<T> nbrsList;

int pos = this->GetVertPos(vertex);

if (pos != (-1))

{

for (int i = 0, vertListSize = this->vertList.size(); i < vertListSize; i++)

{

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

{

nbrsList.push\_back(this->vertList[i]);

}

}

}

return nbrsList;

}

inline void InsertVertex(const T& vertex)// Добавление вершины

{

if (!this->IsFull())

{

this->vertList.push\_back(vertex);

}

else

{

cout << "Граф уже заполнен. Невозможно добавить новую вершину." << endl;

return;

}

}

void InsertEdge(const T& vertex1, const T& vertex2, int weight = 1)// Вставка ребра

{

if (this->GetVertPos(vertex1) != (-1) && this->GetVertPos(vertex2) != (-1))

{

int vertPos1 = GetVertPos(vertex1);

int vertPos2 = GetVertPos(vertex2);

if (this->adjMatrix[vertPos1][vertPos2] != 0 && this->adjMatrix[vertPos2][vertPos1] != 0)

{

cout << "Ребро между этими вершинами уже существует" << endl;

return;

}

else

{

this->adjMatrix[vertPos1][vertPos2] = weight;

this->adjMatrix[vertPos2][vertPos1] = weight;

}

}

else

{

cout << "Обеих вершин (или одной из них) нет в графе" << endl;

return;

}

}

void Print()// Печать матрицы смежности графа

{

int vertListSize = this->vertList.size();

if (!this->IsEmpty())

{

cout << "Матрица смежности: " << endl;

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

{

cout << this->vertList[i] << " ";

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

{

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

}

cout << endl;

}

}

else

{

cout << "Граф пуст" << endl;

}

}

T& Front()// Возвращает первый элемент в очереди

{

return VertsQueue.front();

}

void removeVertex(const T& vertex)// Удаление вершины

{

int pos = GetVertPos(vertex);

if (pos == -1)

{

cout << "Вершины " << vertex << " нет в графе." << endl;

return;

}

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

{

if (adjMatrix[pos][i] != 0)

{

removeEdge(vertex, vertList[i]);

}

if (adjMatrix[i][pos] != 0)

{

removeEdge(vertList[i], vertex);

}

}

vertList.erase(vertList.begin() + pos);

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

{

adjMatrix[i].erase(adjMatrix[i].begin() + pos);

}

adjMatrix.erase(adjMatrix.begin() + pos);

maxSize--;

cout << "Вершина " << vertex << " удалена." << endl;

}

void removeEdge(const T& vertex1, const T& vertex2)// Удаление ребра

{

int pos1 = GetVertPos(vertex1);

int pos2 = GetVertPos(vertex2);

if (pos1 == -1 || pos2 == -1)

{

cout << "Одной из вершин нет в графе." << endl;

return;

}

if (adjMatrix[pos1][pos2] == 0)

{

cout << "Ребра между вершинами " << vertex1 << " и " << vertex2 << " нет." << endl;

return;

}

adjMatrix[pos1][pos2] = 0;

adjMatrix[pos2][pos1] = 0;

cout << "Ребро между вершинами " << vertex1 << " и " << vertex2 << " удалено." << endl;

}

void editEdgeWeight(const T& vertex1, const T& vertex2, int newWeight)// Измена веса между ребрами

{

int pos1 = GetVertPos(vertex1);

int pos2 = GetVertPos(vertex2);

if (pos1 == -1 || pos2 == -1)

{

cout << "Одной из вершин нет в графе." << endl;

return;

}

if (adjMatrix[pos1][pos2] == 0)

{

cout << "Ребра между вершинами " << vertex1 << " и " << vertex2 << " нет." << endl;

return;

}

adjMatrix[pos1][pos2] = newWeight;

adjMatrix[pos2][pos1] = newWeight;

cout << "Вес ребра между вершинами " << vertex1 << " и " << vertex2 << " изменен на " << newWeight << " ." << endl;

}

void obhGlub(T& startVertex, bool\* visitedVerts)// Обход графа в глубину

{

cout << "Вершина " << startVertex << "посещена." << endl;

visitedVerts[this->GetVertPos(startVertex)] = true;

vector<T>neighbors = this->GetNbrs(startVertex);

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

{

if (!visitedVerts[this->GetVertPos(neighbors[i])])

{

this->obhGlub(neighbors[i], visitedVerts);

}

}

}

void obhShir(T& startVertex, bool\* visitedVerts)// Обход графа в ширину

{

if (visitedVerts[this->GetVertPos(startVertex)] == false)

{

this->vertList.push\_back(startVertex);

cout << "Вершина " << startVertex << " обработана." << endl;

visitedVerts[this->GetVertPos(startVertex)] = true;

}

vector<T>neighbors = this->GetNbrs(startVertex);

this->vertList.pop\_back();

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

{

if (!visitedVerts[this->GetVertPos(neighbors[i])])

{

this->vertList.push\_back(neighbors[i]);

visitedVerts[this->GetVertPos(neighbors[i])] = true;

cout << "Вершина " << neighbors[i] << " обработана." << endl;

}

}

if (this->vertList.empty())

{

return;

}

obhShir(vertList.front(), visitedVerts);

}

void FillLabels(T& startVertex)

{

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

{

labelList[i] = 1000000;

}

int pos = GetVertPos(startVertex);

labelList[pos] = 0;

}

bool AllVisited(vector<bool>& visitedVerts)

{

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

{

if (!visitedVerts[i])

{

return false;

}

}

return true;

}

void Dikstra(const T& startVertex)

{

const int n = 100000;

int size = vertList.size();

int\* min\_d = new int[size];

int\* count = new int[size];

int tmp, minIndex, min;

int begin\_index = 0;

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

{

min\_d[i] = n;

count[i] = 1;

}

min\_d[begin\_index] = 0;

do

{

minIndex = n;

min = n;

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

{

if ((count[i] == 1) && (min\_d[i] < min))

{

min = min\_d[i];

minIndex = i;

}

}

if (minIndex != n)

{

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

{

if (adjMatrix[minIndex][i] > 0)

{

tmp = min + adjMatrix[minIndex][i];

if (tmp < min\_d[i])

{

min\_d[i] = tmp;

}

}

}

count[minIndex] = 0;

}

} while (minIndex < n);

cout << endl << "Кратчайшие расстояния до вершин: " << endl << endl;

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

{

cout << "До " << (i + 1) << " вершины: ";

cout << " " << min\_d[i] << endl << endl;

}

delete[] min\_d;

delete[] count;

}

void Floida()

{

int n = this->GetAmountVerts();

int parent[n][n];

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

{

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

{

if (this->adjMatrix[i][j] != INFINITY && i != j)

{

parent[i][j] = i;

}

else

{

parent[i][j] = -1;

}

}

}

for (int k = 0; k < n; k++)

{

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

{

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

{

if (this->adjMatrix[i][j] > this->adjMatrix[i][k] + this->adjMatrix[k][j])

{

this->adjMatrix[i][j] = this->adjMatrix[i][k] + this->adjMatrix[k][j];

parent[i][j] = parent[k][j];

}

}

}

}

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

{

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

{

cout << "Кратчайший путь между " << i << " и " << j << " это ";

if (this->adjMatrix[i][j] == INFINITY)

{

cout << "Бесконечность";

}

else

{

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

}

cout << "";

printPath(parent[i][j], i, j);

cout << endl;

}

}

}

void printPath(int parent[], int i, int j)

{

if (i == j)

{

cout << i << " ";

}

else

{

if (parent[j] == -1)

{

cout << "No path exists";

}

else

{

printPath(parent, i, parent[i]);

cout << j << " ";

}

}

}

};

main.cpp:

#include "Graf.h"

#include <iostream>

#include <vector>

#include <queue>

#include <string>

using namespace std;

int main()

{

setlocale(LC\_ALL, "RU");

Graph<int> graph(6);

int Verts, Edges, vertex, sourceVertex, targetVertex, Weight, vertex1;

bool\* visitedVerts = new bool[6];

bool\* visitedVerts1 = new bool[6];

fill(visitedVerts, visitedVerts + 20, false);

cout << "Количество вершин графа: ";

cin >> Verts;

cout << "Количество ребер графа: ";

cin >> Edges;

cout << endl;

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

{

cout << "Вершина: ";

cin >> vertex;

graph.InsertVertex(vertex);

}

cout << endl;

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

{

cout << "Исходная вершина: ";

cin >> sourceVertex;

cout << "Конечная вершина: ";

cin >> targetVertex;

cout << "Вес ребра: ";

cin >> Weight;

int\* targetVerPtr = &targetVertex;

graph.InsertEdge(sourceVertex, targetVertex, Weight);

}

cout << endl;

graph.Print();

int n;

cout << "Введите вершину, с которой начинается обход: ";

cin >> vertex;

cout << endl << "Обход в глубину: " << endl;

graph.obhGlub(vertex, visitedVerts);

cout << "Введите вершину, с которой начинается обход: ";

cin >> vertex;

cout << endl << "Обход в ширину: " << endl;

graph.obhGlub(vertex, visitedVerts);

cout << "Введите вершину, с которой начинается обход: ";

cin >> vertex;

cout << endl << "Обход Дейкстры: " << endl;

graph.Dikstra(vertex);

delete[]visitedVerts;

cout << "Введите вершину, которую нужно удалить: ";

cin >> n;

graph.removeVertex(n);

graph.Print();

cout << "Размер графа: " << graph.GetAmountVerts();

cout << endl;

int v, r;

cout << "Введите количество вершин, которые нужно добавить: ";

cin >> v;

cout << "Введите количество ребер, которые нужно добавить: ";

cin >> r;

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

{

cout << "Вершина: ";

cin >> vertex;

graph.InsertVertex(vertex);

cout << endl;

}

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

{

cout << "Исходная вершина: ";

cin >> sourceVertex;

cout << "Конечная вершина: ";

cin >> targetVertex;

cout << "Вес ребра: ";

cin >> Weight;

int\* targetVerPtr = &targetVertex;

graph.InsertEdge(sourceVertex, targetVertex, Weight);

}

graph.Print();

cout << "Введите номера вершин, между которыми нужно удалить ребро: ";

cin >> vertex;

cin >> vertex1;

graph.removeEdge(vertex, vertex1);

cout << "Граф, после удаления ребра: " << endl;

graph.Print();

cout << endl << endl;

cout << "Введите номера вершин, между которыми нужно изменить вес ребра: ";

cin >> vertex;

cin >> vertex1;

cout << endl << endl << "";

cin >> Weight;

graph.editEdgeWeight(vertex, vertex1, Weight);

cout << "Граф, после удаления ребра: " << endl;

graph.Print();

cout << "Алгоритм Флойды: " << endl;

graph.Floida();

return 0;}

**Код для визуализации на OpenGL:**

int main(int argc, char\* argv[])

{

setlocale(LC\_ALL, "rus");

glutInit(&argc, argv);

graph = makeGraph();

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGBA);

glutInitWindowSize(1350, 730);

glutCreateWindow("Graph");

WinW = glutGet(GLUT\_WINDOW\_WIDTH);

WinH = glutGet(GLUT\_WINDOW\_HEIGHT);

glutDisplayFunc(display);

glutReshapeFunc(reshape);

glutMouseFunc(mouseClick);

glutMainLoop();

return 0;

}

#pragma once

#include <glut.h>

#include <stdio.h>

#include <iostream>

#include <vector>

#include <sstream>

using namespace std;

int n;

int\*\* help;

int\* result;

int\*\*\* mat;

int WinW;

int WinH;

const int maxSize = 20;

int amountVerts;

template<class T>

class Graph

{

private:

vector<T> vertList;

vector<T> labelList;

bool\* visitVerts = new bool[vertList.size()];

public:

int adjMatrix[maxSize][maxSize] = { 0 };

Graph();

~Graph();

void DrawGraph();

void InsertEdge(const T& vertex1, const T& vertex2, int weight);

void InsertVertex(const T& vertex);

void DeleteVertex();

int GetVerPos(const T& vertex);

bool IsEmpty();

bool IsFull();

int GetAmountVerts();

int GetAmountEdges();

int GetWeight(const T& vertex1, const T& vertex2);

vector<T> GetNbrs(const T& vertex);

void Print();

};

int R;

struct vertCoord

{

int x, y;

};

vertCoord vertC[20];

Graph<int> graph;

void answer(int\*\*\* mat, int n, int\*\* help, int\* path)

{

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

{

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

{

int min = 1000000;

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

{

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

{

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

}

}

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

{

if (mat[i][j])

{

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

}

}

}

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

{

int min = 1000000;

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

{

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

{

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

}

}

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

{

if (mat[i][j])

{

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

}

}

}

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

{

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

{

help[i][j] = 0;

}

}

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

{

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

{

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

{

int hmin = 1000000;

int vmin = 1000000;

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

{

if (l != i && mat[l][j] && hmin > \*mat[l][j])

{

hmin = \*mat[l][j];

}

}

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

{

if (l != j && mat[i][l] && hmin > \*mat[i][l])

{

vmin = \*mat[i][l];

}

}

help[i][j] = hmin + vmin;

}

}

}

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

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

{

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

{

if (mat[i][j] && mcost < help[i][j])

{

mcost = help[i][j];

mi = i;

mj = j;

}

}

}

path[mi] = mj;

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

{

mat[i][mj] = nullptr;

}

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

{

mat[mi][i] = nullptr;

}

mat[mj][mi] = nullptr;

}

}

void preparation(int\*\*\*& mat, int& n, int\*\*& help, int\*& result)

{

n = amountVerts;

help = new int\* [n];

result = new int[n];

mat = new int\*\* [n];

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

{

help[i] = new int[n];

}

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

{

mat[i] = new int\* [n];

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

{

if (graph.adjMatrix[i][j] == 0)

{

mat[i][j] = nullptr;

continue;

}

mat[i][j] = new int(graph.adjMatrix[i][j]);

}

}

}

void tsalesman(int\*\*\* mat, int n, int\*\* help, int\* result)

{

preparation(mat, n, help, result);

int s = 0;

answer(mat, n, help, result);

cout << endl << "Отрезки путей: ";

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

{

j = result[i];

cout << i + 1 << " -> " << j + 1 << '\t';

s += graph.adjMatrix[i][j];

}

cout << endl;

cout << endl << "Кратчайший путь: ";

int tmp = 0;

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

{

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

{

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

{

if (tmp == 0)

{

cout << i + 1;

}

j = result[i];

tmp = j + 1;

if (tmp > 0)

{

cout << " -> " << tmp;

}

l++;

}

}

}

cout << endl << "Минимальное расстояние: " << s;

cout << endl;

}

template<class T>

vector<T> Graph<T>::GetNbrs(const T& vertex)

{

vector<T> nbrsList;

int pos = this->GetVerPos(vertex);

if (pos != (-1))

{

for (int i = 0, vertListSize = this->vertList.size(); i < vertListSize; i++)

{

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

{

nbrsList.push\_back(this->vertList[i]);

}

}

}

return nbrsList;

}

template<class T>

void Graph<T>::InsertVertex(const T& vertex)

{

if (!this->IsFull())

{

this->vertList.push\_back(vertex);

}

else

{

cout << "Граф уже заполнен. Невозможно добавить новую вершину." << endl;

return;

}

}

template<class T>

void Graph<T>::DeleteVertex()

{

this->vertList.pop\_back();

}

template<class T>

int Graph<T>::GetAmountEdges()

{

int amount = 0;

if (!this->IsEmpty())

{

for (int i = 0, vertListSize = this->vertList.size(); i < vertListSize; i++)

{

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

{

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

{

amount++;

}

}

}

return amount / 2;

}

else

return 0;

}

template<class T>

int Graph<T>::GetWeight(const T& vertex1, const T& vertex2)

{

if (!this->IsEmpty())

{

int vert\_1 = this->GetVertPos(vertex1);

int vert\_2 = this->GetVertPos(vertex2);

return adjMatrix[vert\_1][vert\_2];

}

return 0;

}

template<class T>

int Graph<T>::GetAmountVerts()

{

return this->vertList.size();

}

template<class T>

bool Graph<T>::IsEmpty()

{

return this->vertList.size() == 0;

}

template<class T>

bool Graph<T>::IsFull()

{

return (vertList.size() == maxSize);

}

template<class T>

int Graph<T>::GetVerPos(const T& vertex)

{

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

{

if (this->vertList[i] == vertex)

{

return i;

}

}

return -1;

}

template<class T>

Graph<T>::Graph()

{

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

{

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

{

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

}

}

}

template<class T>

Graph<T>::~Graph()

{

}

Graph<int> makeGraph()

{

Graph<int> graph;

int amountEdges, sourceVertex, targetVertex, edgeWeight, vertex;

cout << "Количество вершин: ";

cin >> amountVerts;

cout << endl;

cout << "Количество ребер: ";

cin >> amountEdges;

cout << endl;

for (int i = 1; i <= amountVerts; i++)

{

int\* vertPtr = &i;

graph.InsertVertex(\*vertPtr);

}

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

{

cout << "Начальная вершина: ";

cin >> sourceVertex;

cout << endl;

int\* sourceVertPtr = &sourceVertex;

cout << "Конечная вершина: ";

cin >> targetVertex;

cout << endl;

int\* targetVertPtr = &targetVertex;

cout << "Вес: ";

cin >> edgeWeight;

cout << endl;

graph.InsertEdge(sourceVertex, targetVertex, edgeWeight);

}

cout << endl;

return graph;

}

template<class T>

void Graph<T>::InsertEdge(const T& vertex1, const T& vertex2, int weight)

{

if (this->GetVerPos(vertex1) != (-1) && this->GetVerPos(vertex2) != (-1))

{

int vertPos1 = GetVerPos(vertex1);

int vertPos2 = GetVerPos(vertex2);

if (this->adjMatrix[vertPos1][vertPos2] != 0 && this->adjMatrix[vertPos2][vertPos1] != 0)

{

cout << "Ребро между этими вершинами уже существует" << endl;

return;

}

else

{

this->adjMatrix[vertPos1][vertPos2] = weight;

this->adjMatrix[vertPos2][vertPos1] = weight;

}

}

else

{

cout << "Обеих вершин (или одной из них) нет в графе" << endl;

return;

}

}

template<class T>

void Graph<T>::Print()

{

int vertListSize = this->vertList.size();

if (!this->IsEmpty())

{

cout << "Матрица смежности: " << endl;

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

{

cout << this->vertList[i] << " ";

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

{

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

}

cout << endl;

}

}

else

{

cout << "Граф пуст" << endl;

}

}

void setCord(int i, int n)

{

int R\_;

int x0 = WinW / 2;

int y0 = WinH / 2;

if (WinW > WinH)

{

R = 5 \* (WinH / 13) / n;

R\_ = WinH / 2 - R - 10;

}

else

{

R = 5 \* (WinW / 13) / n;

R\_ = WinW / 2 - R - 10;

}

float theta = 2.0f \* 3.1415926f \* float(i) / float(n);

float y1 = R\_ \* cos(theta) + y0;

float x1 = R\_ \* sin(theta) + x0;

vertC[i].x = x1;

vertC[i].y = y1;

}

void drawCircle(int x, int y, int R)

{

glColor3f(0.13, 0.7, 0.63);

float x1, y1;

glBegin(GL\_POLYGON);

for (int i = 0; i < 360; i++)

{

float theta = 2.0f \* 3.1415926f \* float(i) / float(360);

y1 = R \* cos(theta) + y;

x1 = R \* sin(theta) + x;

glVertex2f(x1, y1);

}

glEnd();

glColor3f(0.0f, 0.0f, 0.0f);

float x2, y2;

glBegin(GL\_LINE\_LOOP);

for (int i = 0; i < 360; i++)

{

float theta = 2.0f \* 3.1415926f \* float(i) / float(360);

y2 = R \* cos(theta) + y;

x2 = R \* sin(theta) + x;

glVertex2f(x2, y2);

}

glEnd();

}

void drawText(int nom, int x1, int y1)

{

GLvoid\* front = GLUT\_BITMAP\_HELVETICA\_18;

string s = to\_string(nom);

glRasterPos2i(x1 - 5, y1 - 5);

for (int j = 0; j < s.length(); j++)

{

glutBitmapCharacter(front, s[j]);

}

}

void drawVertex(int n)

{

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

{

drawCircle(vertC[i].x, vertC[i].y, R);

drawText(i + 1, vertC[i].x, vertC[i].y);

}

}

void drawLine(int text, int x0, int y0, int x1, int y1)

{

glColor3f(0.0f, 0.0f, 0.0f);

glBegin(GL\_LINES);

glVertex2i(x0, y0);

glVertex2i(x1, y1);

glEnd();

drawText(text, (x0 + x1) / 2 + 10, (y0 + y1) / 2 + 10);

}

template<class T>

void Graph<T>::DrawGraph()

{

int n = vertList.size();

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

{

setCord(i, n);

}

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

{

for (int j = i + 1; j < n; j++)

{

int a = adjMatrix[i][j];

if (a != 0)

{

drawLine(a, vertC[i].x, vertC[i].y, vertC[j].x, vertC[j].y);

}

}

}

drawVertex(n);

}

void reshape(int w, int h)

{

WinW = w;

WinH = h;

glViewport(0, 0, (GLsizei)WinW, (GLsizei)WinH);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, (GLdouble)WinW, 0, (GLdouble)WinH);

glutPostRedisplay();

}

void drawMenuText(string text, int x1, int y1)

{

GLvoid\* font = GLUT\_BITMAP\_HELVETICA\_18;

string s = text;

glRasterPos2i(x1 + 5, y1 - 20);

for (int j = 0; j < s.length(); j++)

{

glutBitmapCharacter(font, s[j]);

}

}

void drawMenu()

{

int shift = 60;

int height = 730;

glColor3d(0.0, 0.5, 0.0);

glBegin(GL\_QUADS);

glVertex2i(shift - 10, height - 50);

glVertex2i(shift + 145, height - 50);

glVertex2i(shift + 145, height - shift - 200);

glVertex2i(shift - 10, height - shift - 200);

glEnd();

glColor3d(0.5, 0.0, 0.3);

glBegin(GL\_QUADS);

glVertex2i(shift, height - shift - 30);

glVertex2i(shift + 135, height - shift - 30);

glVertex2i(shift + 135, height - shift);

glVertex2i(shift, height - shift);

glEnd();

glColor3d(1, 1, 1);

drawMenuText("Insert Vertex", shift, height - shift - 2);

glColor3d(0.5, 0.0, 0.3);

glBegin(GL\_QUADS);

glVertex2i(shift, height - shift - 70);

glVertex2i(shift + 135, height - shift - 70);

glVertex2i(shift + 135, height - shift - 40);

glVertex2i(shift, height - shift - 40);

glEnd();

glColor3d(1, 1, 1);

drawMenuText("Delete Vertex", shift, height - shift - 42);

glColor3d(0.5, 0.0, 0.3);

glBegin(GL\_QUADS);

glVertex2i(shift, height - shift - 110);

glVertex2i(shift + 135, height - shift - 110);

glVertex2i(shift + 135, height - shift - 80);

glVertex2i(shift, height - shift - 80);

glEnd();

glColor3d(1, 1, 1);

drawMenuText("Matrix", shift, height - shift - 82);

glColor3d(0.5, 0.0, 0.3);

glBegin(GL\_QUADS);

glVertex2i(shift, height - shift - 150);

glVertex2i(shift + 135, height - shift - 150);

glVertex2i(shift + 135, height - shift - 120);

glVertex2i(shift, height - shift - 120);

glEnd();

glColor3d(1, 1, 1);

drawMenuText("Answer", shift, height - shift - 122);

glColor3d(0.5, 0.0, 0.3);

glBegin(GL\_QUADS);

glVertex2i(shift, height - shift - 190);

glVertex2i(shift + 135, height - shift - 190);

glVertex2i(shift + 135, height - shift - 160);

glVertex2i(shift, height - shift - 160);

glEnd();

glColor3d(1, 1, 1);

drawMenuText("New Matrix", shift, height - shift - 162);

}

void mouseClick(int btn, int stat, int x, int y)

{

int shift = 60;

int height = 730;

if (stat == GLUT\_DOWN)

{

if (x > shift && x<shift + 135 && y>shift && y < shift + 30)

{

int sourceVertex;

int targetVertex;

int edgeWeight;

cout << "Начальная вершина: ";

cin >> sourceVertex;

cout << endl;

int\* sourceVertPtr = &sourceVertex;

cout << "Конечная вершина: ";

cin >> targetVertex;

cout << endl;

int\* targetVertPtr = &targetVertex;

if (sourceVertex > amountVerts || targetVertex > amountVerts)

{

amountVerts++;

int\* vertPtr = &amountVerts;

graph.InsertVertex(\*vertPtr);

}

cout << "Вес: ";

cin >> edgeWeight;

cout << endl;

graph.InsertEdge(sourceVertex, targetVertex, edgeWeight);

}

if (x > shift && x<shift + 135 && y>shift + 40 && y < shift + 70)

{

int sourceVertex;

int targetVertex;

int edgeWeight;

cout << "Введите вершину: ";

cin >> sourceVertex;

cout << endl;

int\* sourceVertPtr = &sourceVertex;

if (sourceVertex == amountVerts)

{

amountVerts--;

graph.DeleteVertex();

}

else

{

cout << "Невозможно удалить вершину!";

}

}

if (x > shift && x<shift + 135 && y>shift + 80 && y < shift + 110)

{

graph.Print();

}

if (x > shift && x<shift + 135 && y>shift + 120 && y < shift + 150)

{

tsalesman(mat, n, help, result);

}

if (x > shift && x<shift + 135 && y>shift + 160 && y < shift + 190)

{

graph = makeGraph();

}

}

glutPostRedisplay();

}

void display()

{

glShadeModel(GL\_SMOOTH);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

gluOrtho2D(0, WinW, 0, WinH);

glViewport(0, 0, WinW, WinH);

glClearColor(0.6, 0.9, 0.9, 1.0);

glClear(GL\_COLOR\_BUFFER\_BIT);

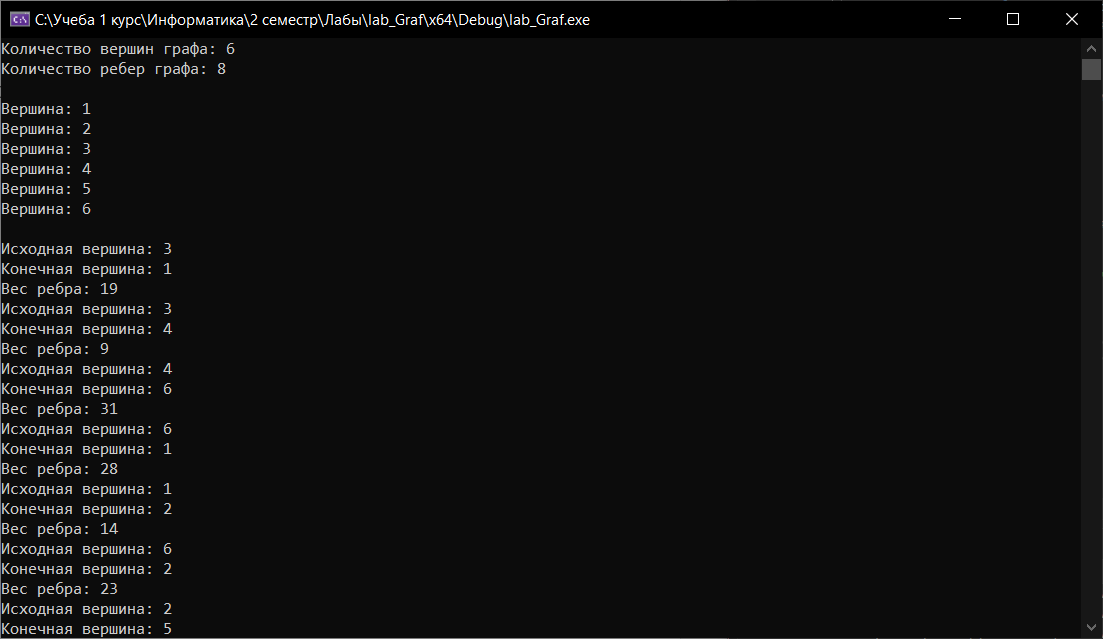
graph.DrawGraph();

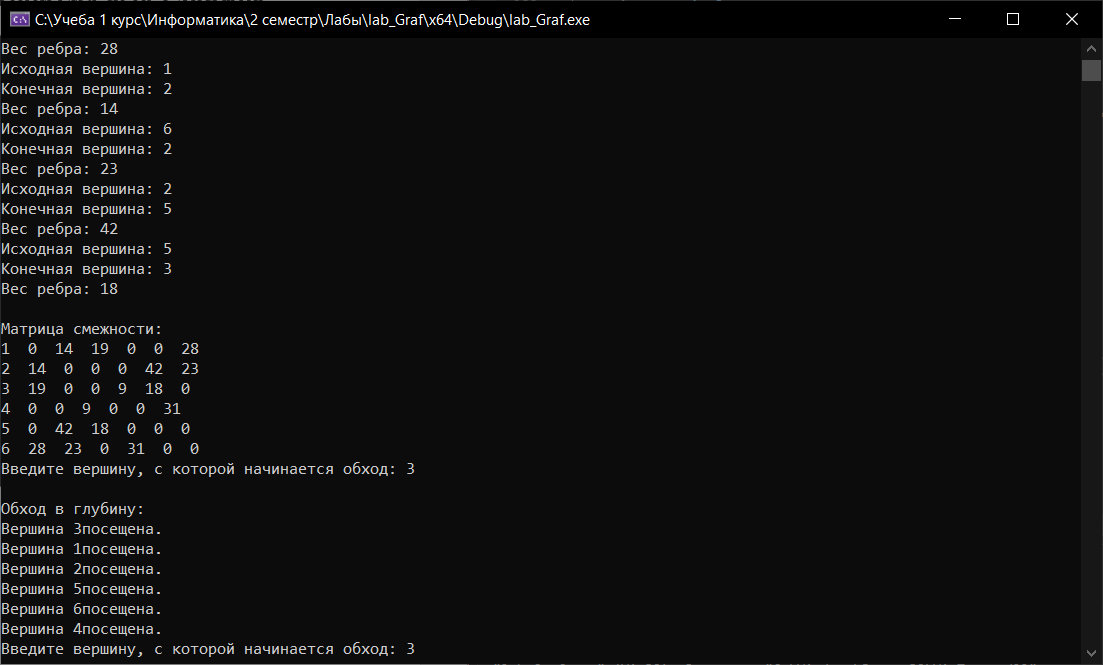
drawMenu();

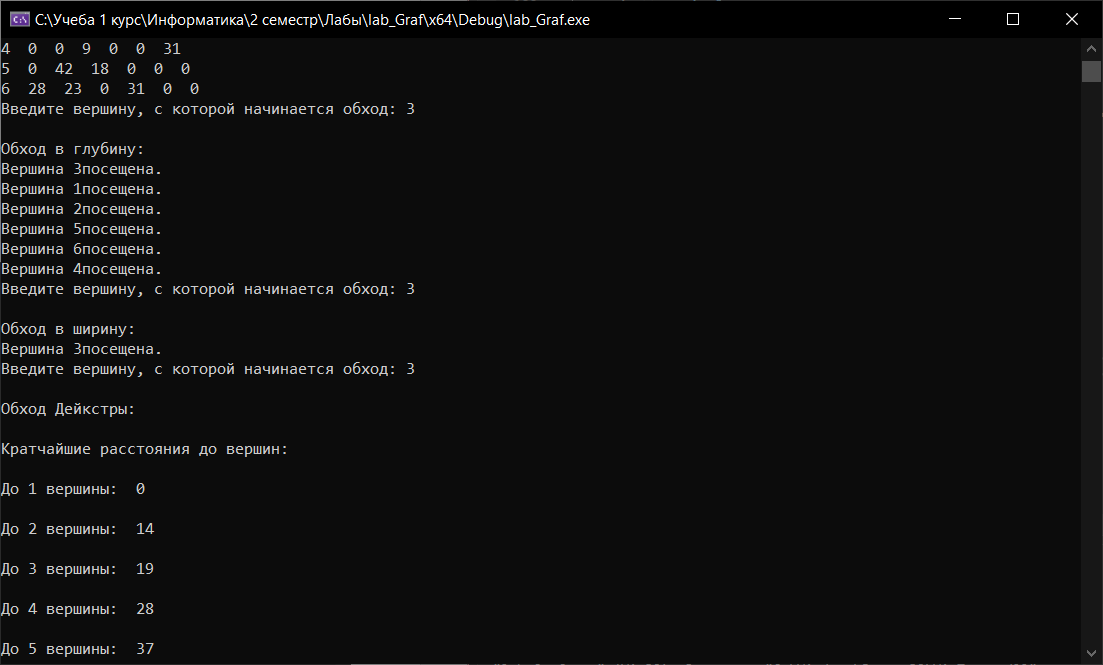
glutSwapBuffers();

}

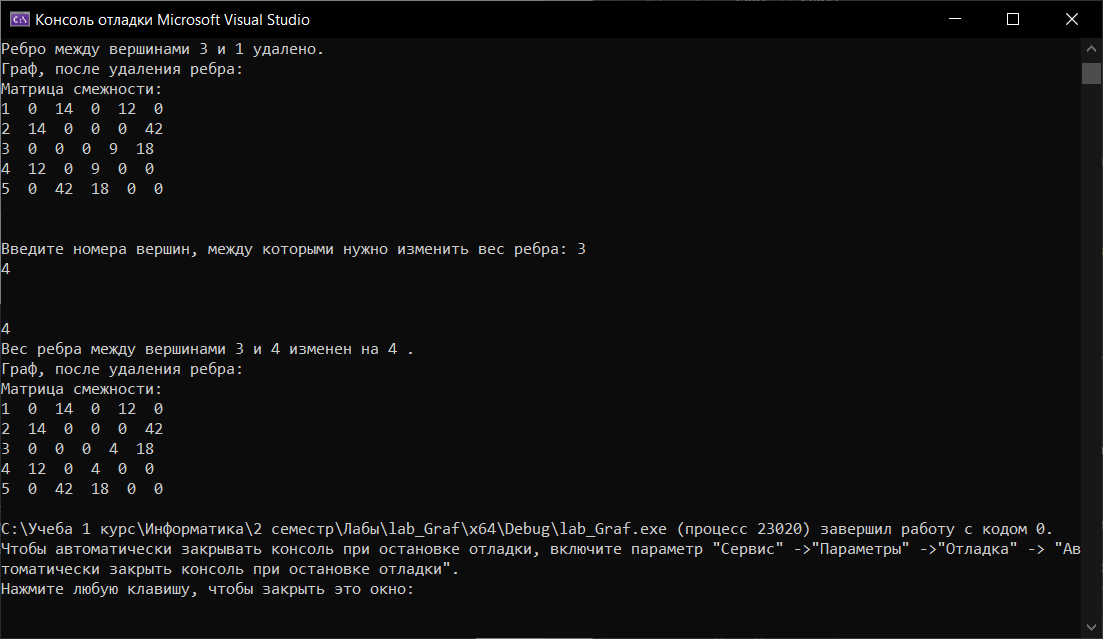
**Скриншоты пропущенного алгоритма через движок:**

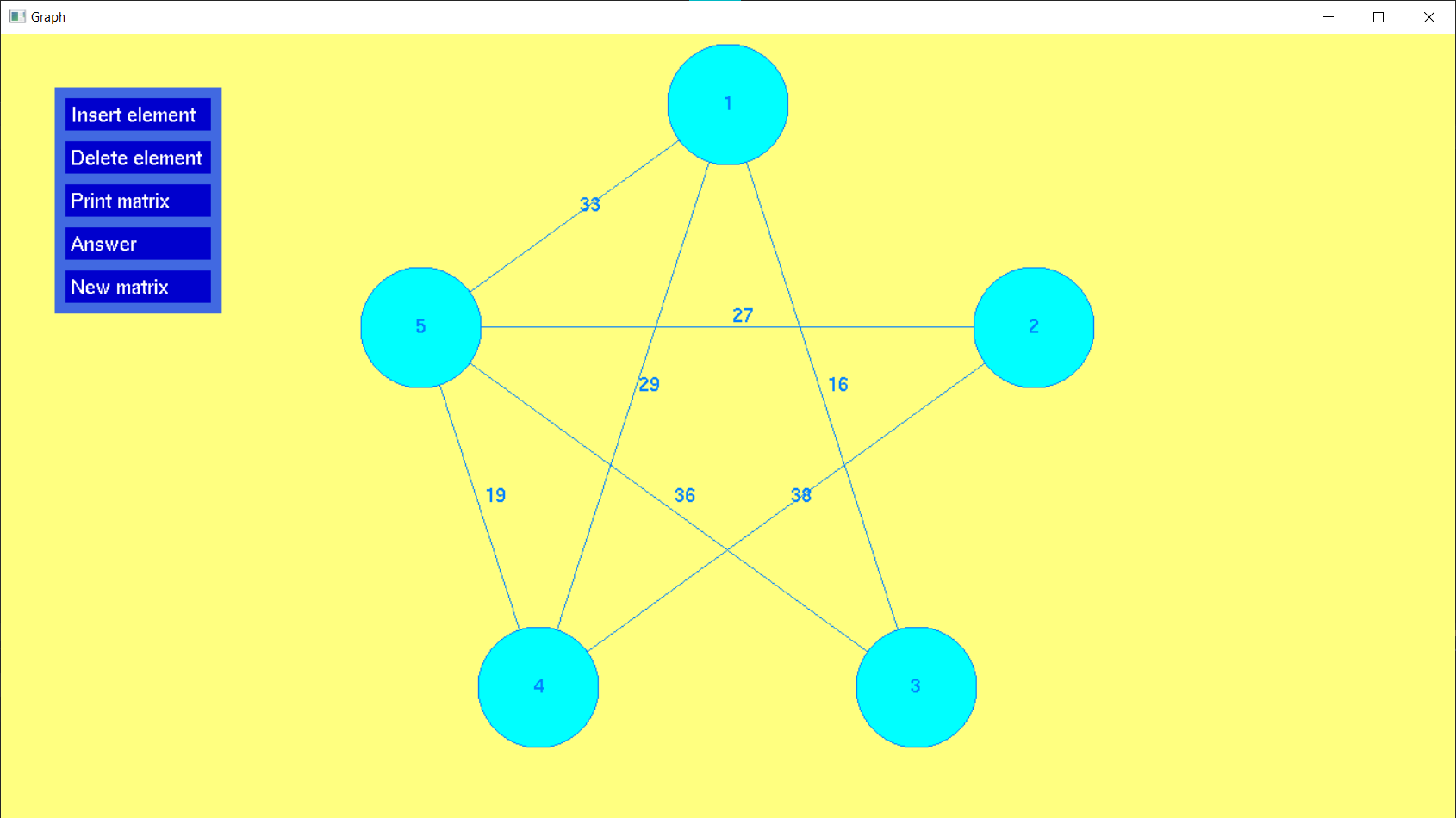
****

****

****

****

****

****

**UML-диаграмма:**

