Министерство науки и высшего образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

ПЕРМСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

(ФГАОУ ПНИПУ)

ЭЛЕКТРОТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ

КАФЕДРА ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И АВТОМАТИЗИРОВАННЫЕ СИСТЕМЫ

ОТЧЁТ

О ЛАБОРАТОРНОЙ РАБОТЕ

По дисциплине «Основы алгоритмизации и программирования» на тему:

КОММИВОЯЖЕР

|  |  |  |
| --- | --- | --- |
|  | 2024 г. | Выполнил:  Студент группы РИС-23-2б  Колосов Д. Е.  Проверил:  Доцент кафедры ИТАС  Петренко А. А |

# ВВЕДЕНИЕ

Постановка задачи

Цель работы: получить практические навыки работы с графами.

Задачи:

1. Реализовать обход графа, а также алгоритм Дейкстры
2. Реализовать алгоритм решения задачи коммивояжера
3. Визуализировать решение задачи коммивояжера

# 1 Реализовать обход графа, а также алгоритм Дейкстры

Алгоритм обхода графа в глубину начинает выполнение с одной из вершин графа - начальной вершины, фиксирует информацию о посещении этой вершины, и, перемещаясь по ребру, посещает соседние вершины.

Кроме того, правило левой руки при прохождении лабиринта (идти, ведя левой рукой по стенке) также является обходом в глубину**.** По завершении обхода все вершины окажутся пройдёнными - обработанными. Если при обходе встречается вершина, которая уже была пройдена, то повторной обработки делать не нужно

Алгоритм обхода в ширину основан на следующих принципах:

* поиск начинается с начальной вершины, которая обрабатывается, маркируется и помещается в очередь;
* Основой алгоритма является циклический процесс, в котором обработанная вершина удаляется из очереди, а в очередь помещаются соседствующие с обработанной вершины. Таким образом, тело цикла состоит из двух основных шагов:

1. Удалить вершину v из головы очереди
2. Для каждой непомеченной вершины u, соседней по отношению к вершине v, обработать вершину u, маркировать ее и поместить в очередь (вершина u может иметь соседние необработанные вершины).

Алгоритм Дейкстры оперирует шаг за шагом. На первом этапе выбирается начальная точка, от которой начинаются все пути в графе. Затем алгоритм систематически находит самые короткие маршруты из этой точки ко всем остальным вершинам. Посещенные вершины помечаются, а информация о пройденных путях используется для вычисления новых маршрутов к не посещённым вершинам.

В начале работы алгоритма выбранной начальной вершине устанавливается значение 0, что указывает на начало отсчета расстояний. Расстояние от этой вершины к самой себе устанавливается как ноль, а для всех остальных вершин устанавливается значение бесконечности в качестве начального значения пути.

После завершения начальной инициализации, алгоритм Дейкстры продолжает свою работу, ища следующую не посещенную вершину с наименьшим расстоянием от начальной точки. На каждом шаге он выбирает ближайшую вершину и вычисляет кратчайшие пути до ее соседних вершин с учетом уже пройденных маршрутов, выбирая оптимальные пути.

Для анализируемого графа, где начальной точкой была выбрана вершина 2, алгоритм продолжает свой процесс, «переходя» в эту вершину и изучая ее соседей для дальнейших вычислений кратчайших путей.

Алгоритм продолжает шаги, пока есть не посещенные вершины. При нахождении более короткого пути к какой-либо точке, минимальное расстояние для этой точки перезаписывается. В конце работы алгоритма получается список кратчайших маршрутов до каждой точки из начальной вершины, с указанием их длины.

Рисунок 3 принцип работы сортировки

Вывод: необходимо реализовать класс графа, методы которого буду осуществлять обходы и алгоритм Дейкстры.

# 2 Реализовать алгоритм решения задачи коммивояжера

Алгоритм решения задачи коммивояжера основывается на 4 принципах:

1. Редукция строк
2. Редукция столбцов
3. Составление матрицы оценки
4. Редукция матрицы

Эти шаги будут повторяться пока в матрице смежности графа не останется полей хранящий в себе значения отличные от нуля и -1 значения, принятого для обозначения не участвующих в алгоритме клеток.

Вывод: необходимо добавить к класс графа метод решения задачи коммивояжера, который будет оперировать методами, выполняющими шаги алгоритма коммивояжера

# 3 Визуализировать решение задачи коммивояжера

Визуализация будет реализована с помощью windows forms. Предусмотрены операции добавления узла – клик левой кнопки мыши – вместе клика появляется узел, удаления узла – необходимо после нажатия кнопки “Удалить вершину” щёлкнуть правой кнопкой мыши по узлу. Который необходимо удалить, установка ребра между узлами – щёлок по узлу отправления и по узлу назначения правой кнопкой мыши – дальше нужно будет ввести значения весов для прямого и обратного направления в специальное место во всплывающем окне, также возможно удаление ребра – после нажатия на кнопку удалить ребро необходимо выбрать щелчком правовый кнопки мыши узлы ребро между которыми необходимо удалить

В приложении А находиться скриншоты работы программы, в приложении Б – код класса графа

Вывод: с помощью windows forms реализован функциональный интерфейс программы

# ЗАКЛЮЧЕНИЕ

В данной лабораторной работе были выполнены следующие задачи:

1. реализованы обходы графа и алгоритм Дейкстры;
2. реализован алгоритм решения задачи коммивояжера
3. разработан интерфейс программы для работы с графами

Полученные практические навыки по алгоритмизации и конструированию интерфейса, а также о взаимодействии графических объектов в windows forms, способах их создания, размещения и обрисовки могут быть успешно использованы в дальнейшей практике разработки положений и алгоритмов.

# ПРИЛОЖЕНИЕ А

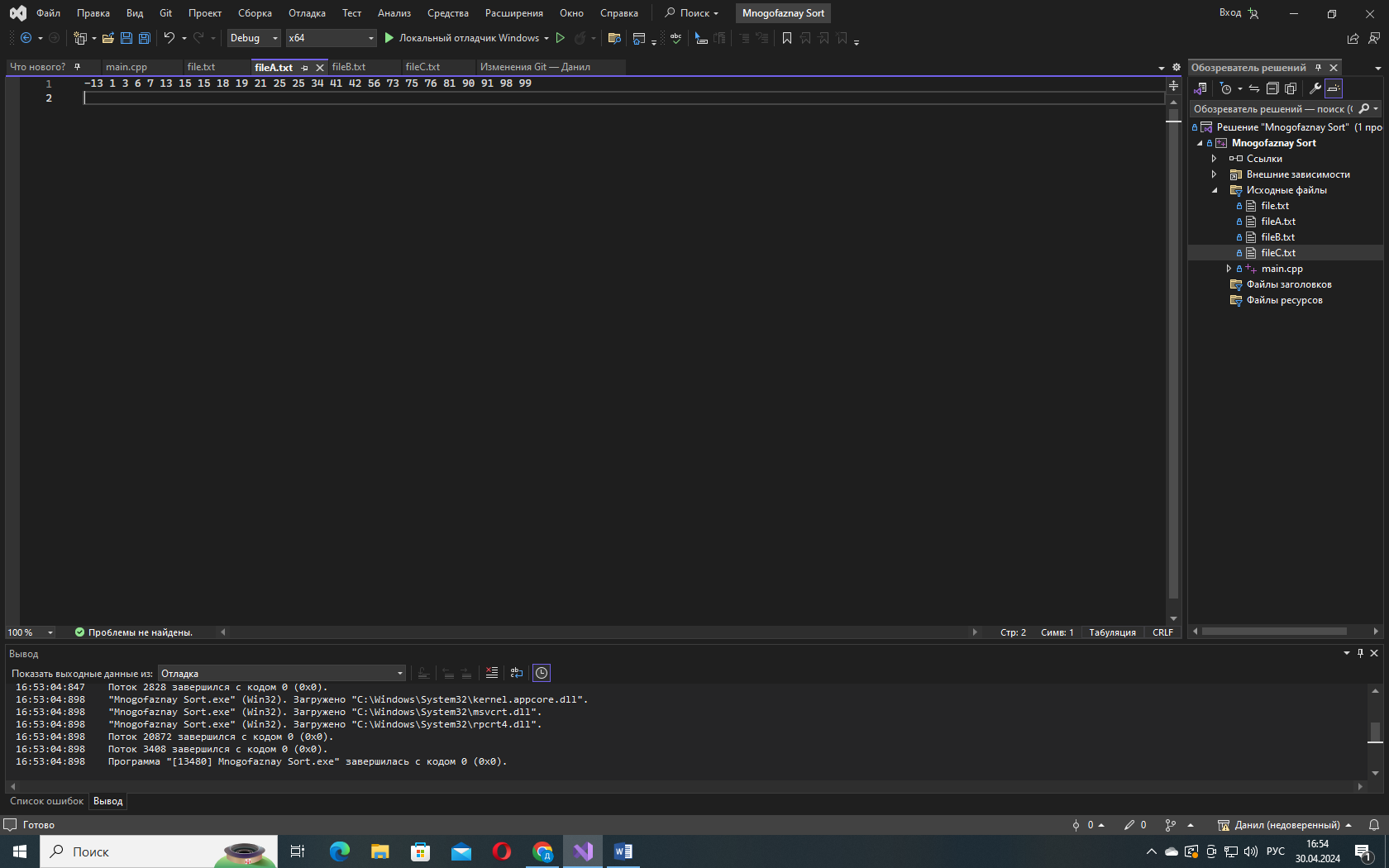
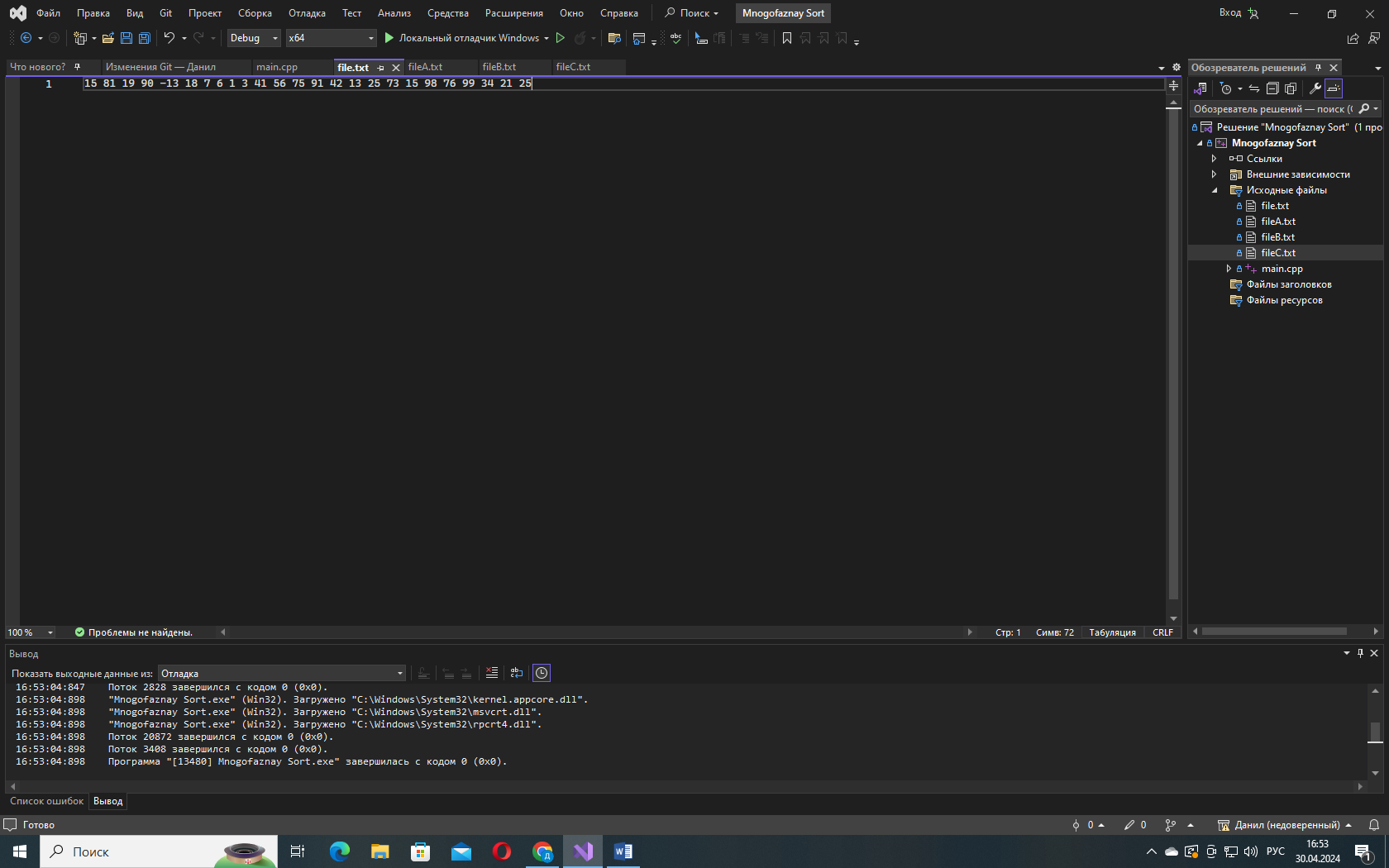
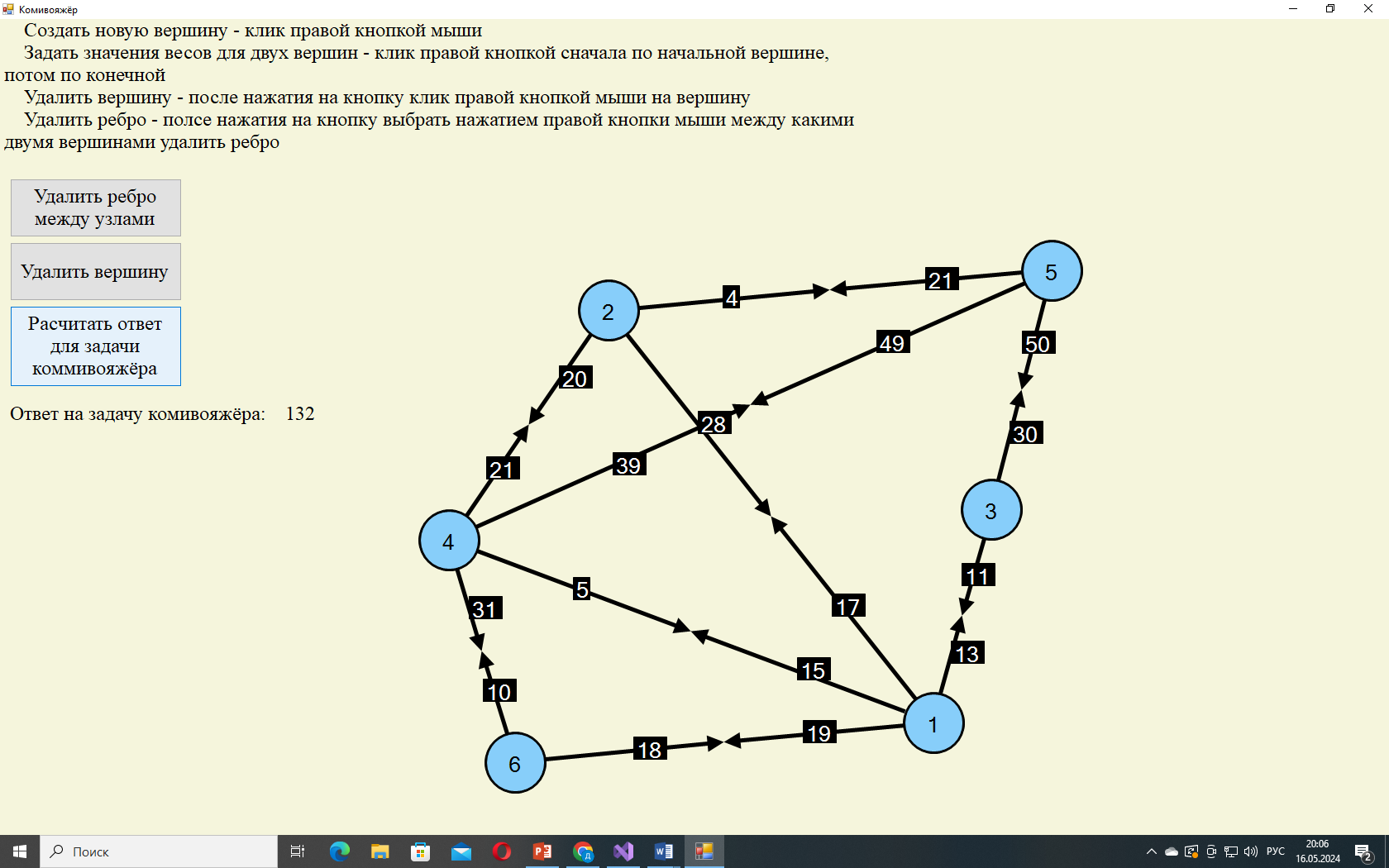


Рисунок 2 вывод результата программы для ориентированного графа

Рисунок 1 вывод программы для не ориентированного графа

Рисунок 5 исходные данные в файле

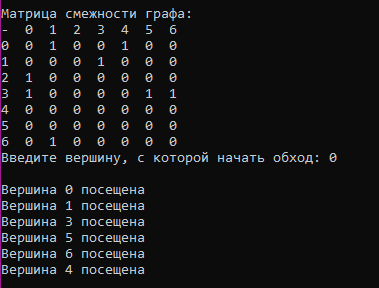


Рисунок 3 вывод результата программы для обхода в глубину

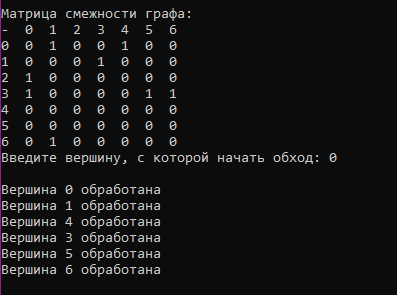


Рисунок 4 вывод результата программы для обхода в ширину

# ПРИЛОЖЕНИЕ Б

#pragma once

using namespace std;

#include <vector>

template <class T>

class Graph {

public://private dolgno

//Вектор вершин

vector<T> vertexList;

//Матрица смежности

vector<vector<int>> Matrix;

//Размер вектора вершин и матрицы смежности

int maxSize;

vector<T> VertsQueue;

//Связь вершин с координатами кружков

vector<int> labelList;

//Ответ на задачу комивояжера

vector<int> answerList;

int answer = 0;

//Связь вершин с координатами кружков

vector<vector<int>> linkList;

public:

Graph(const int&); //Конструктор

bool isFull(); //Проверка на заполненность

bool isEmpty(); //Проверка на пустоту

void insertVertex(const T&); //Вставка вершины

int GetVertPos(const T&); //Получение индекса вершины

int GetAmountVerts(); //Получение количества вершин

int GetWeight(const T&, const T&); //Получение веса между вершинами

vector<T> GetNbrs(const T&); //Получение вектора соседей вершины с данными

void insertEdge( int, int, int); //Вставка ребра для ориентированного графа

int GetAmountEdges(); //Получение количества рёбер для ориентированного графа

void DFS(T&, bool\*);//Обход в глубину

void BFS(T&, bool\*); //Обход в ширину

void deleteVertex(T vert) //Метод удаляющий вершину из матрицы смежности

{

int v\_p = GetVertPos(vert);

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

Matrix[v\_p][i]=-1;

Matrix[i][v\_p]=-1;

}

}

//Методы решения задачи комивояжера

vector<vector<int>> getReducirVecStr(vector<vector<int>>&);//Получение редукции матрицы по строкам

vector<vector<int>> getReducirVecCol(vector<vector<int>>);//Получение редукции матрицы по столбцам

vector<vector<int>> getVecMark(vector<vector<int>>);//Получение матрицы оценки

int getTravelVertexTo(vector<vector<int>>);//Определени индекса вектора, в который ведёт найденный кратчайший путь

int getTravelVertexOut(vector<vector<int>> mark);//Определени индекса вектора, из которого ведёт найденный кратчайший путь

vector<vector<int>> getReducMatrix(vector<vector<int>>);//Получение редукции матрицы

int Count(vector<vector<int>>);//метод, отвечающий за цикличность повторяющихся редукций

int minMark(int I, int J, vector<vector<int>>);//Метод возвращающий оценку дл яодного жлемента матрицы

int Komivoyger();//основной метод решения задачи

int getVertxListSize() { return vertexList.size(); }//получение количества вершин в графе

T getVertexListItem(int index) { return vertexList[index]; }//Получение вершины по индексу

void FillLables(T&); //Начальное заполнение меток расстояния

bool AllVisited(vector<bool>&); //Проверяет все ли вершины были посещены

void Dijkstra(T&); //Алгоритм Дейкстры

};

template<class T>

inline Graph<T>::Graph(const int& size) {

this->maxSize = size;

this->Matrix = vector<vector<T>>(size, vector<T>(size));

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

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

this->Matrix[i][j] = -1;

}

}

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

labelList.push\_back(0);

}

template<class T>

inline bool Graph<T>::isFull() {

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

}

template<class T>

inline bool Graph<T>::isEmpty() {

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

}

template<class T>

inline void Graph<T>::insertVertex(const T& vert) {

if (this->isFull()) {

cout << "Невозможно добавить вершину - переполнение." << endl;

return;

}

this->vertexList.push\_back(vert);

/\*if(vert<vertexList.size()

this->vertexList.push\_back(vert);

else

{

this->vertexList.insert(vertexList.begin()+vert-2,vert);

}\*/

}

template<class T>

inline int Graph<T>::GetVertPos(const T& v) {

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

if (this->vertexList[i] == v) {

return i;

}

}

return -1;

}

template<class T>

inline int Graph<T>::GetAmountVerts() {

return this->vertexList.size();

}

template<class T>

inline int Graph<T>::GetWeight(const T& v1, const T& v2) {

if (this->isEmpty()) {

return 0;

}

int v1\_p = this->GetVertPos(v1);

int v2\_p = this->GetVertPos(v2);

if (v1\_p == -1 || v2\_p == -1) {

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

return 0;

}

return this->Matrix[v1\_p][v2\_p]; //Значение весов вершин с индексами v1\_p и v2\_p находиться в [v1\_p][v2\_p] ячейке матрицы смежности

}

template<class T>

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

vector<T> neighbourList; //Создание списка соседей

int pos = this->GetVertPos(vertex); //Вычисление позиции vertex в матрице смежности

if (pos != -1) {

//проверка, что vertex есть в матрице смежности

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

if (this->Matrix[pos][i] != 0) //Если на пересечении с ршиной с индексом pos есть какая-то вершина

neighbourList.push\_back(this->vertexList[i]);

}

}

return neighbourList; //Возврат списка соседей

}

template<class T>

void Graph<T>::insertEdge( int vertex1, int vertex2, int weight = 1) { //Вставка ребра в одну сторону от v1 к v2

//Можно сделать чтобы в разные стороны разные числа были

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

int vertPos1 = GetVertPos(vertex1);

int vertPos2 = GetVertPos(vertex2);

if (this->Matrix[vertPos1][vertPos2] != 0 && this->Matrix[vertPos1][vertPos2] != -1) {

cout << "Ребро между вершинами уже есть" << endl;

return;

}

else

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

}

else{

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

return;

}

}

template<class T>

int Graph<T>::GetAmountEdges() {

int amount = 0;

int n = this->vertexList.size();

if (!this->isEmpty()) {

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

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

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

amount++;

}

}

}

return amount;

}

template<class T>

void Graph<T>::DFS(T& startVertex, bool\* visitedverts) { //Обход в глубину

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

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

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

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

if (!visitVerts[this->GetVertPos(neighbours[i])])

this->DFS(neighbours[i], visitedVerts);

}

}

template<class T>

void Graph<T>::BFS(T& startVertex, bool\* visitedVerts) {//vertVisited как на 190 стр может AllVisited

//Условие истино только при первом вызове

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

this->VertsQueue.push(startVertex);

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

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

}

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

this->VertsQueue.pop();

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

if (!visitedVerts[thsi->GetVertPos(neighbours[i])]) {

this->VertsQueue.push(neighbours[i]);

visitedVerts[this->Getvertpos(neighbours[i])] = true;

cout << "Вершина " << neighbours[i] << " обработана" << endl;

}

}

if (this->VertsQueue, empty())

return;

BFS(VertsQueue.front(), visitedVerts);

}

template<class T>

void Graph<T>::FillLables(T& startVertex) {

//Начальное заполнение меток расстояния

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

labelList[i] = 1000000;

int pos = GetvertPos(startVertex);

labelList[pos] = 0;

}

template<class T>

bool Graph<T>::AllVisited(vector<bool>& visitedVerts) {

//проверяет все ли вершины были посещены

bool flag = true;

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

if (visitedVertex[i] != true)

flag = false;

}

return flag;

}

template<class T>

void Graph<T>::Dijkstra(T& startVertex) {

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

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

if (Matrix[i][j] < 0)

return;

}

}

if (GetVertPos(startVertex) == -1)

return;

vector<bool> visitedVerts(veretexList.size());

fill(visitedVerts.begin(), visitedVerts.end(), false);

this->FillLabels(startVertex);

T curVert = startVertex;

vector<T> neighbours;

while (!this->AllVisited(visitedVerts)) {

neighbours = this->GetNbrs(curVert);

int startLabel = labelList[GetVertPos(curVert)];

//Самый близкий сосед

T\* minNeighbour\_ptr = nullptr;

int minWeight = 1000000;

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

int weight = this->GetWeight(curVert, neighbours[i]);

int vertIndex = this->GetVertPos(neighbours[i]);

int nextLabel = labelList[vertIndex];

if (startLabel + weight < nextLabel)

labelList[vertIndex] = startLabel + weight;

if (!visitedVerts[vertIndex] && minWeight > labelList[vertIndex]) {

minWeight = labelList[vertIndex];

minNeighbour\_ptr = &neighbours[i];

}

}

visitedVerts[GetVertPos(curVert)] = true;

if (minNeighbour\_ptr != nullptr)

curVert = \*minNeighbour\_ptr;

}

}

template<class T>

vector<vector<int>> Graph<T>::getReducirVecStr(vector<vector<int>>& matrix) {

//Минимум по строкам

vector<int> d(maxSize, 0);

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

int min = 10000;

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

if (matrix[i][j] < min && matrix[i][j] != -1)

min = matrix[i][j];

}

if (min != -1)

d[i] = min;

}

//Редукция строк по минмуму строк

cout << endl << "\*\*" << endl;

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

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

if (matrix[i][j] != -1 && matrix[i][j] != 0) {

if (matrix[i][j] - d[i] < 0)

matrix[i][j] = 0;

else

matrix[i][j] = matrix[i][j] - d[i];//

}

}

}

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

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

{

cout << matrix[i][j] << " ";

}

cout << endl;

}

cout << endl;

return matrix;

}

template<class T>

vector<vector<int>> Graph<T>::getVecMark(vector<vector<int>> matrix) {

//Матрица оценки

vector<vector<int>> mark;//

mark = this->Matrix;

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

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

mark[i][j] = -1;

}

}

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

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

if (matrix[i][j] == 0)

mark[i][j] = minMark(i, j, matrix);

}

}

return mark;

}

template<class T>

vector<vector<int>> Graph<T>::getReducirVecCol(vector<vector<int>> matrix) {

//Минимум по столбцам

vector<int> d(maxSize, 0);

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

int min = 1000;

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

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

min = matrix[i][j];

}

}

if (min != -1)

d[j] = min;

}

//Редукция столбцов

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

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

if (matrix[i][j] != -1 && matrix[i][j] != 0) {

if (matrix[i][j] - d[j] < 0)

matrix[i][j] = 0;

else

matrix[i][j] = matrix[i][j] - d[j];

}

}

}

return matrix;

}

template<class T>

int Graph<T>::getTravelVertexOut(vector<vector<int>> mark) {

int ni;

int nj;

int nmax = -1;

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

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

if (mark[i][j] > nmax)

nmax = mark[i][j];

}

}

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

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

if (mark[i][j] == nmax) {

ni = i + 1;

nj = j + 1;

}

}

}

return ni;

}

template<class T>

int Graph<T>::getTravelVertexTo(vector<vector<int>> mark) {

int ni;

int nj;

int nmax = -1;

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

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

if (mark[i][j] > nmax)

nmax = mark[i][j];

}

}

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

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

if (mark[i][j] == nmax) {

ni = i + 1;

nj = j + 1;

}

}

}

return nj;

}

//Редукция матрицы

template<class T>

vector<vector<int>> Graph<T>::getReducMatrix(vector<vector<int>> matrix) {

vector<vector<int>> mark;//

mark = getVecMark(matrix);

cout << "оценка" << endl;

int ni = getTravelVertexOut(mark);

int nj = getTravelVertexTo(mark);

matrix[ni - 1][nj - 1] = -1;

matrix[nj - 1][ni - 1] = -1;

answerList.push\_back(ni);

answerList.push\_back(nj);

answerList.push\_back(0);

//answer += this->Matrix[ni][nj];

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

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

if (i == (ni - 1) || j == (nj - 1))

matrix[i][j] = -1;

}

}

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

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

cout << mark[i][j] << " ";

}

cout << endl;

}

return matrix;

}

template<class T >

int Graph<T>::Count(vector<vector<int>> matrix) {

int cp = -1;

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

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

if (matrix[i][j] > 0)

cp++;

}

}

return cp;

}

template<class T>

int Graph<T>::minMark(int I, int J, vector<vector<int>> matrix) {

int minStr = 1000;

int minCol = 1000;

for (int i = I; i <= I; i++) {

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

if (matrix[i][j] < minStr && matrix[i][j] != -1 && (j != J || i != I))

minStr = matrix[i][j];

}

}

vector<int> d2 = { maxSize };

for (int j = J; j <= J; j++) {

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

if (Matrix[i][j] < minCol && matrix[i][j] != -1 && (i != I || j != J)) {

minCol = Matrix[i][j];

}

}

}

return minStr + minCol;

}

template<class T>

int Graph<T>::Komivoyger() {

vector<vector<int>>mr = this->Matrix;

while (Count(this->Matrix) != -1) {

this->Matrix = this->getReducirVecStr(this->Matrix);

this->Matrix = this->getReducirVecCol(this->Matrix);

this->Matrix = this->getReducMatrix(this->Matrix);

}

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

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

if (this->Matrix[i][j] == 0) {

this->answerList.push\_back(i + 1);

this->answerList.push\_back(j + 1);

this->answerList.push\_back(0);

}

}

}

for (int i = 0; i < this->answerList.size() - 1; i++) {

if (this->answerList[i] != 0 && this->answerList[i + 1] != 0) {

int c = this->answerList[i] - 1;

int c2 = this->answerList[i + 1] - 1;

answer += mr[this->answerList[i] - 1][this->answerList[i + 1] - 1];

}

}

int otvet = answer;

answer = 0;

answerList.clear();

this->Matrix = mr;

return otvet;

}

/\*int komik() {

//+табличка из какоого пункта отправляемся

//T v1 = textBox->Text;

//v2

//weight

void setMatrix(const T & v1, const T & v2,int weight){

if (this->isEmpty()) {

return 0;

}

int v1\_p = this->GetVertPos(v1);

int v2\_p = this->GeetVertPos(v2);

if (v1\_p == -1 || v2\_p == -1) {

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

return 0;

}

Matrix[v1\_p][v2\_p] = weight;

}

createLine();

}\*/

/\*template<class T>

void Graph<T>::Dijkstra(T& startVertex) {

for

}\*/