МИНЕСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧЕРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«Брестский государственный технический университет»

Кафедра «Интеллектуальные информационные технологии»

Лабораторная работа №3

По дисциплине «Дискретная математика»

За 3 семестр

Тема: «Графы»

Выполнила:

студентка 2 курса

группы АС-56

Карпенко М.В.

Проверил:

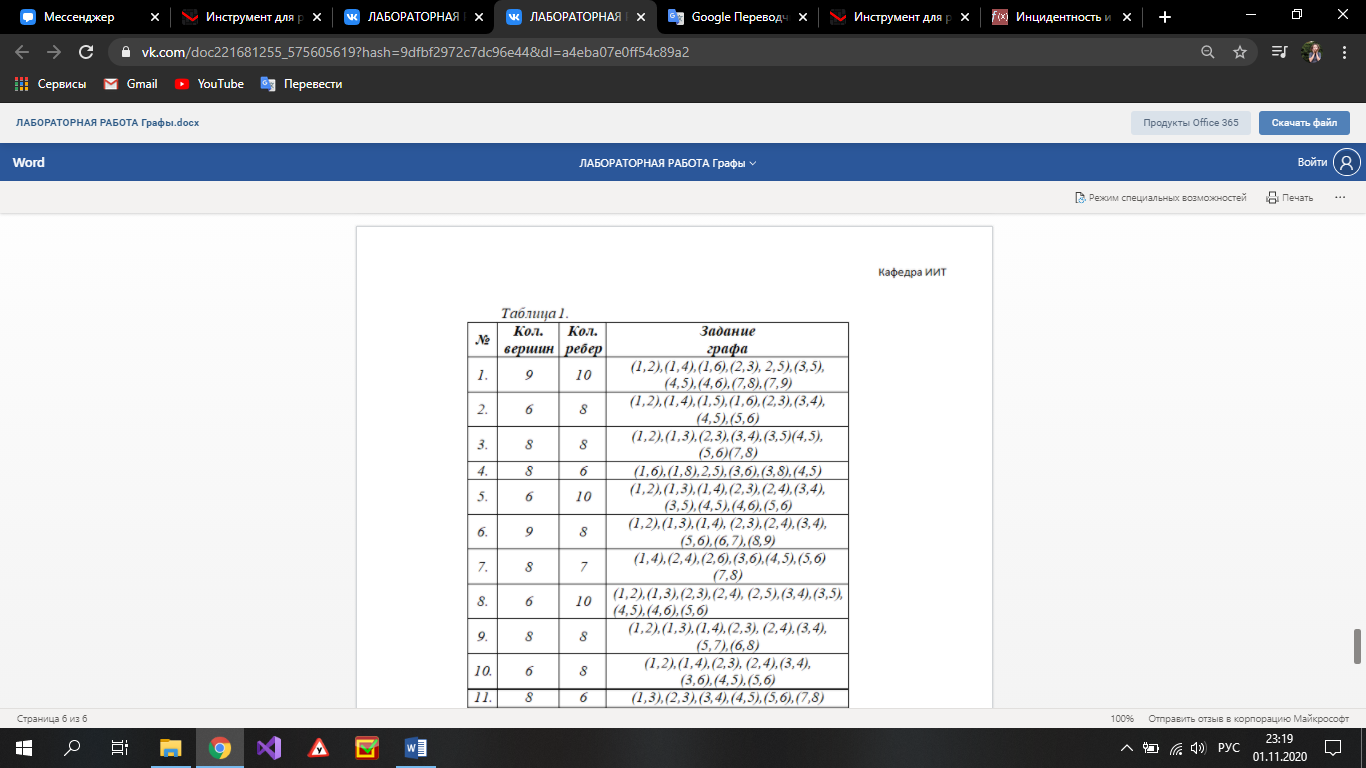
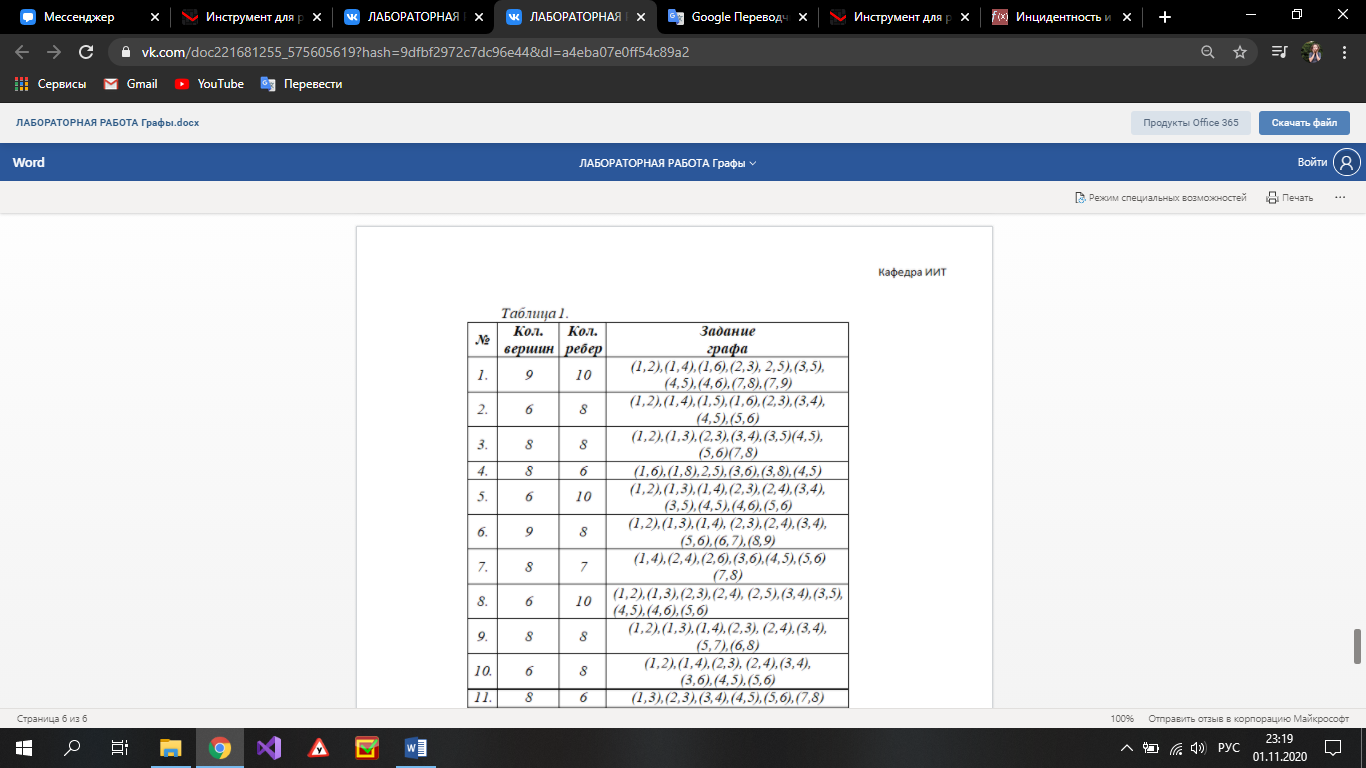
Глущенко Т.А.

Брест 2020

*Цель:* Усвоить базовые концепции из теории графов, научиться производить обход графа в ширину и в глубину, а также находить число компонент связности. Научиться реализовывать алгоритмы Дейкстры и Флойда-Уоршалла.

*Задание 1 (вариант 7)*

1. Построить *матрицу смежности* и *инцидентности* для заданного графа. Изобразить граф.
2. По матрице смежности (инцидентности) для каждой из вершин вычислить ее *степень*.
3. Используя поиск в глубину и поиск в ширину написать программу, определяющую *число компонент связности* графа. Методы представляются в виде отдельных функций (или классов).
4. Построить *деревья поиска в ширину и глубину*.
5. Из заданного неориентированного графа построить произвольным образом ориентированный граф (добавить к каждому ребру стрелку). Для полученного таким образом ориентированного графа построить *матрицу смежности и инцидентности*.
6. Из заданного неориентированного графа построить произвольным образом *псевдограф*.
7. Варианты заданий указаны в таблице 1.
8. В таблице граф задан списком ребер, например, запись *(1,2)*означает, что существует ребро, соединяющее вершину *1* с вершиной 2.

*Код программы:*

#include <iostream>

#include <Windows.h>

#include <stack>

#include <ctime>

#include <cstdlib>

#include <iomanip>

#include <queue>

using namespace std;

enum outMode { num\_num = 0, num\_alph, alph\_num, alph\_alph };

enum graphType { directed = 0, undirected };

int \*\* adjMatrix(const int &numOfVer, int \*\*e, const int &numOfE, graphType gT = undirected);

int \*\* incMatrix(const int &numOfVer, int \*\*e, const int &numOfE, graphType gT = undirected);

void outMatrix(int \*\* matrix, const int &numOfR, const int &numOfC, outMode m = num\_num);

int \* verDeg(int \*\* adjM, const int &numOfVer);

int \*\* makeDGraph(int \*\*e, const int &numOfE);

int \*\* makePsGraph(const int &numOfVer, int \*\*e, const int &numOfE, int &numOfPsE);

template<typename T>

void enlArr(T \*& arr, int & size);

class DepthFirst

{

private:

    int numOfConnComp;

    int \*\*e;

    int numOfE;

public:

    DepthFirst(const int &numOfVer, int \*\*e, const int& numOfE);

    int \*\* getE(int& numOfE);

    int \*\* getE();

    int getNumOfE();

    int getConnComp();

};

class BreadthFirst

{

private:

    int numOfConnComp;

    int \*\* e;

    int numOfE;

public:

    BreadthFirst(const int &numOfVer, int \*\*e, const int& numOfE);

    int \*\* getE(int& numOfE);

    int \*\* getE();

    int getNumOfE();

    int getConnComp();

};

int main()

{

    const int numOfE = 7;

    const int numOfVer = 8;

    int eTemp[numOfE][2] = { { 1, 4 },{ 2, 4 },{ 2, 6 },{ 3, 6 },{ 4, 5 },{ 5, 6 },{ 7, 8 } };

    int \*\*e = new int\*[numOfE];

    for (int i = 0; i < numOfE; i++)

    {

        e[i] = new int[2];

        e[i][0] = eTemp[i][0];

        e[i][1] = eTemp[i][1];

    }

    int \*\* adjM = adjMatrix(numOfVer, e, numOfE);

    int \*\* incM = incMatrix(numOfVer, e, numOfE);

    cout << "Adjacency matrix: " << endl;

    outMatrix(adjM, numOfVer, numOfVer); cout << endl;

    cout << "Incidence matrix: " << endl;

    outMatrix(incM, numOfVer, numOfE, num\_alph);

    DepthFirst dFTE(numOfVer, e, numOfE);

    BreadthFirst brFTE(numOfVer, e, numOfE);

    cout << "Number of connected components: " << dFTE.getConnComp() << "; " << brFTE.getConnComp() << endl;

    int \*\* eD = makeDGraph(e, numOfE);

    int \*\* adjMD = adjMatrix(numOfVer, eD, numOfE, directed);

    int \*\* incMD = incMatrix(numOfVer, eD, numOfE, directed);

    cout << "Directed graph: " << endl;

    outMatrix(adjMD, numOfVer, numOfVer); cout << endl;

    outMatrix(incMD, numOfVer, numOfE, num\_alph); cout << endl;

    int \*\* dAdjM = adjMatrix(numOfVer, dFTE.getE(), dFTE.getNumOfE());

    cout << "Depth-first tree adjacency matrix: " << endl;

    outMatrix(dAdjM, numOfVer, numOfVer); cout << endl;

    int \*\* brAdjM = adjMatrix(numOfVer, brFTE.getE(), brFTE.getNumOfE());

    cout << "Breadth-first tree adjacency matrix: " << endl;

    outMatrix(brAdjM, numOfVer, numOfVer); cout << endl;

    int numOfPsE;

    int \*\*psE = makePsGraph(numOfVer, e, numOfE, numOfPsE);

    int \*\* adjMPs = adjMatrix(numOfVer, psE, numOfPsE);

    cout << "Pseudograph adjacency matrix: " << endl;

    outMatrix(adjMPs, numOfVer, numOfVer); cout << endl;

    system("pause");

    return 0;

}

int \*\* adjMatrix(const int &numOfVer, int \*\*e, const int &numOfE, graphType gT)

{

    int \*\* adjMatrix = new int\*[numOfVer];

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

    {

        adjMatrix[i] = new int[numOfVer];

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

            adjMatrix[i][j] = 0;

    }

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

        for (int j = 1; j <= numOfVer; j++)

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

            {

                if (e[k][0] == i&&e[k][1] == j)

                    adjMatrix[i - 1][j - 1]++;

                else if (e[k][0] == j&&e[k][1] == i)

                    if (gT == undirected)

                        adjMatrix[i - 1][j - 1]++;

                    else if (gT == directed)

                        adjMatrix[i - 1][j - 1]--;

            }

    return adjMatrix;

}

int \*\* incMatrix(const int &numOfVer, int \*\*e, const int &numOfE, graphType gT)

{

    int \*\* incMatrix = new int\*[numOfVer];

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

    {

        incMatrix[i] = new int[numOfE];

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

            incMatrix[i][j] = 0;

    }

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

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

            if (gT == undirected)

            {

                if (e[j][0] == i || e[j][1] == i)

                    incMatrix[i - 1][j]++;

            }

            else if (gT == directed)

            {

                if (e[j][0] == i)

                    incMatrix[i - 1][j]++;

                else if (e[j][1] == i)

                    incMatrix[i - 1][j]--;

            }

    return incMatrix;

}

void outMatrix(int \*\* matrix, const int &numOfR, const int &numOfC, outMode m)

{

    const int cw = 2;

    char alphabet[] = "abcdefghijklmnopqrstuvwxyz";

    int sizeOfAlph = sizeof(alphabet) / sizeof(\*alphabet);

    cout << setw(cw) << '#' << setw(cw) << ' ';

    if (m == num\_num || m == alph\_num)

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

            cout << setw(cw) << i + 1 << setw(cw) << ' ';

    else

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

            cout << setw(cw) << alphabet[i] << setw(cw) << ' ';

    cout << endl << endl;

    for (int i = 0; i < numOfR; i++)

    {

        if (m == num\_num || m == num\_alph)

            cout << setw(cw) << i + 1 << setw(cw) << ' ';

        else

            cout << setw(cw) << alphabet[i] << setw(cw) << ' ';

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

            cout << setw(cw) << matrix[i][j] << setw(cw) << ' ';

        cout << endl;

    }

}

int \* verDeg(int \*\* adjM, const int &numOfVer)

{

    int \* degM = new int[numOfVer];

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

    {

        int deg = 0;

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

            if (adjM[i][j]) deg++;

        degM[i] = deg;

    }

    return degM;

}

int \*\* makeDGraph(int \*\*e, const int &numOfE)

{

    int \*\*newE = new int\*[numOfE];

    for (int i = 0; i < numOfE; i++)

    {

        newE[i] = new int[2];

        newE[i][0] = e[i][0];

        newE[i][1] = e[i][1];

        if (rand() % 2 == 0)

            swap(newE[i][0], newE[i][1]);

    }

    return newE;

}

int \*\* makePsGraph(const int &numOfVer, int \*\*e, const int &numOfE, int &numOfPsE)

{

    numOfPsE = numOfE;

    int \*\*psE = new int\*[numOfPsE];

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

    {

        psE[i] = new int[2];

        psE[i][0] = e[i][0];

        psE[i][1] = e[i][1];

    }

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

    {

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

        {

            if (!(rand() % 5))

            {

                enlArr(psE, numOfPsE);

                psE[numOfPsE - 1] = new int[2];

                psE[numOfPsE - 1][0] = i;

                psE[numOfPsE - 1][1] = j;

            }

        }

    }

    return psE;

}

template<typename T>

void enlArr(T \*& arr, int & size)

{

    if (size == 0)

    {

        size = 1;

        arr = new T[size];

        return;

    }

    T\* arrCopy = new T[size];

    for (int i = 0; i < size; i++)

        arrCopy[i] = arr[i];

    size++;

    arr = new T[size];

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

        arr[i] = arrCopy[i];

}

DepthFirst::DepthFirst(const int &numOfVer, int \*\*e, const int& numOfE)

{

    this->numOfConnComp = 0;

    bool \* marked = new bool[numOfVer];

    for (int i = 0; i < numOfVer; i++) marked[i] = false;

    stack <int> v;

    while (true)

    {

        int cur = 0;

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

            if (marked[i] == false)

            {

                cur = i + 1;

                break;

            }

        if (!cur) break;

        this->numOfConnComp++;

        v.push(cur);

        while (!v.empty())

        {

            int nextVer = 0;

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

            {

                if (e[i][0] == v.top() || e[i][1] == v.top())

                {

                    int elId;

                    if (e[i][0] == v.top()) elId = 1;

                    else if (e[i][1] == v.top()) elId = 0;

                    bool isSuitable = true;

                    stack <int> v\_copy = v;

                    while (!v\_copy.empty())

                    {

                        if (e[i][elId] == v\_copy.top())

                            isSuitable = false;

                        v\_copy.pop();

                    }

                    if (marked[(e[i][elId] - 1)])

                        isSuitable = false;

                    if (isSuitable)

                    {

                        nextVer = e[i][elId];

                        break;

                    }

                }

            }

            if (!nextVer)

            {

                int lastEl = v.top();

                v.pop();

                marked[(lastEl - 1)] = true;

                if (v.empty())

                    continue;

                enlArr(this->e, this->numOfE);

                this->e[this->numOfE - 1] = new int[2];

                this->e[this->numOfE - 1][0] = lastEl;

                this->e[this->numOfE - 1][1] = v.top();

            }

            else v.push(nextVer);

        }

    }

}

int \*\* DepthFirst::getE(int& numOfE)

{

    numOfE = this->numOfE;

    return this->e;

}

int \*\* DepthFirst::getE()

{

    return this->e;

}

int DepthFirst::getNumOfE()

{

    return this->numOfE;

}

int DepthFirst::getConnComp()

{

    return this->numOfConnComp;

}

BreadthFirst::BreadthFirst(const int &numOfVer, int \*\*e, const int& numOfE)

{

    this->numOfConnComp = 0;

    bool \* marked = new bool[numOfVer];

    for (int i = 0; i < numOfVer; i++) marked[i] = false;

    while (true)

    {

        int cur = 0;

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

        {

            if (marked[i] == false)

            {

                cur = i + 1;

                break;

            }

        }

        if (!cur) break;

        queue <int> v;

        v.push(cur);

        marked[cur - 1] = true;

        this->numOfConnComp++;

        while (!v.empty())

        {

            int fr = v.front();

            v.pop();

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

                if (fr == e[i][0] || fr == e[i][1])

                {

                    int elId;

                    if (fr == e[i][0]) elId = 1;

                    else if (fr == e[i][1]) elId = 0;

                    if (marked[e[i][elId] - 1] == false)

                    {

                        v.push(e[i][elId]);

                        marked[e[i][elId] - 1] = true;

                        enlArr(this->e, this->numOfE);

                        this->e[this->numOfE - 1] = new int[2];

                        this->e[this->numOfE - 1][0] = fr;

                        this->e[this->numOfE - 1][1] = e[i][elId];

                    }

                }

        }

    }

}

int \*\* BreadthFirst::getE(int& numOfE)

{

    numOfE = this->numOfE;

    return this->e;

}

int \*\* BreadthFirst::getE()

{

    return this->e;

}

int BreadthFirst::getNumOfE()

{

    return this->numOfE;

}

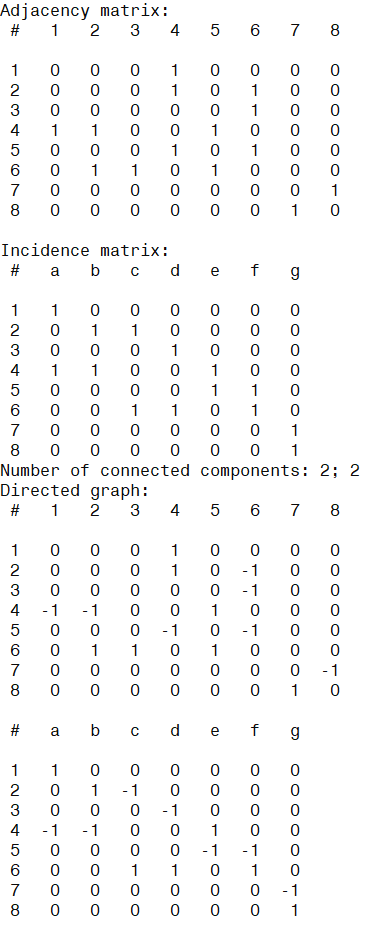
int BreadthFirst::getConnComp()

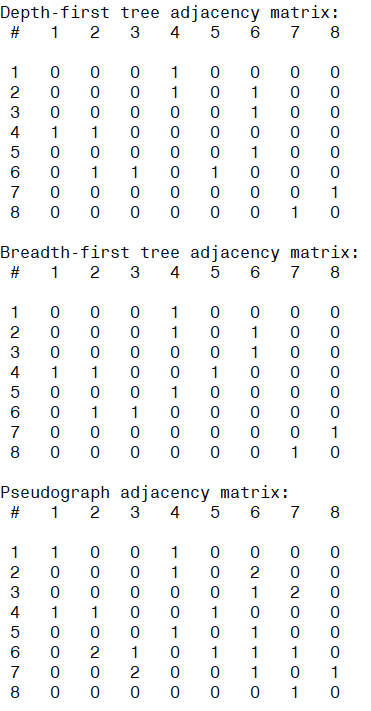
{

    return this->numOfConnComp;

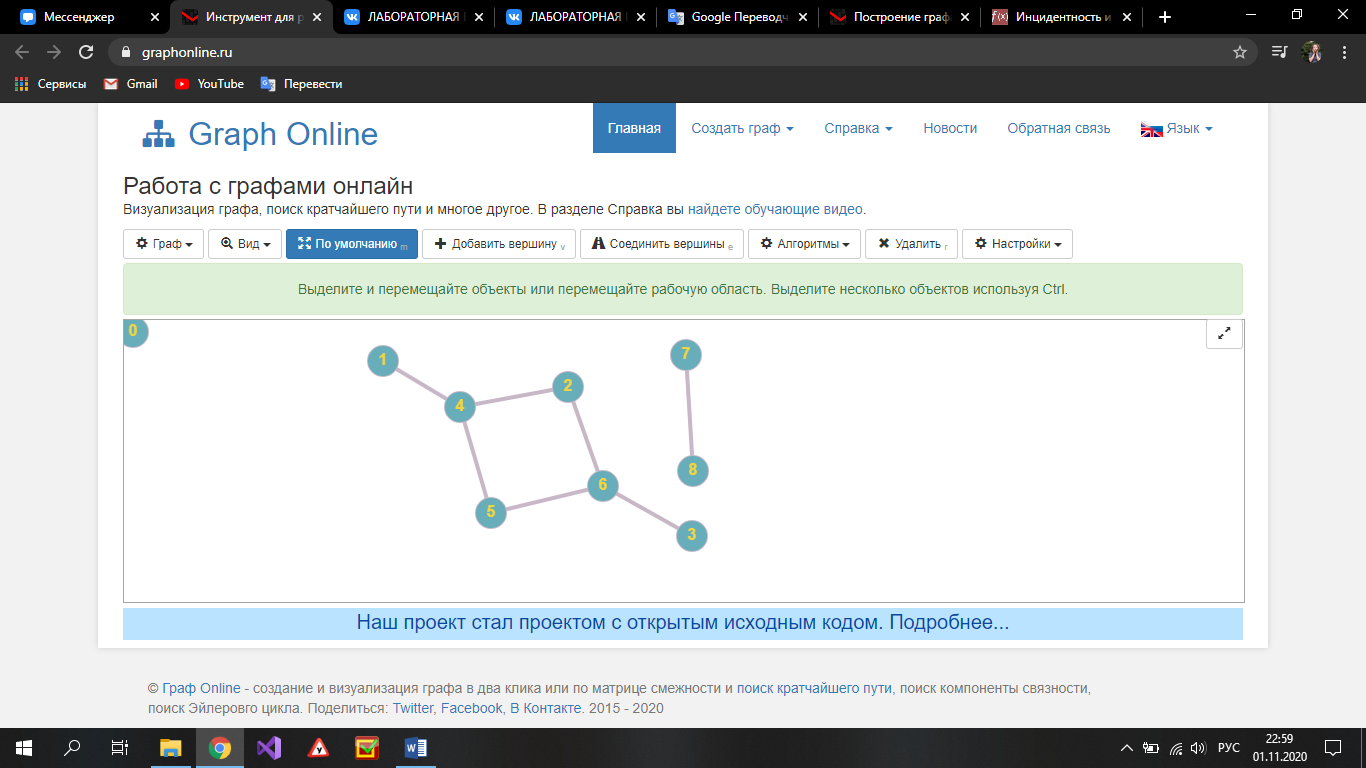
}

*Результат выполнения:*





*Изображение графа*



*Степень вершин*

1 вершина: 1

2 вершина: 2

3 вершина: 1

4 вершина: 2

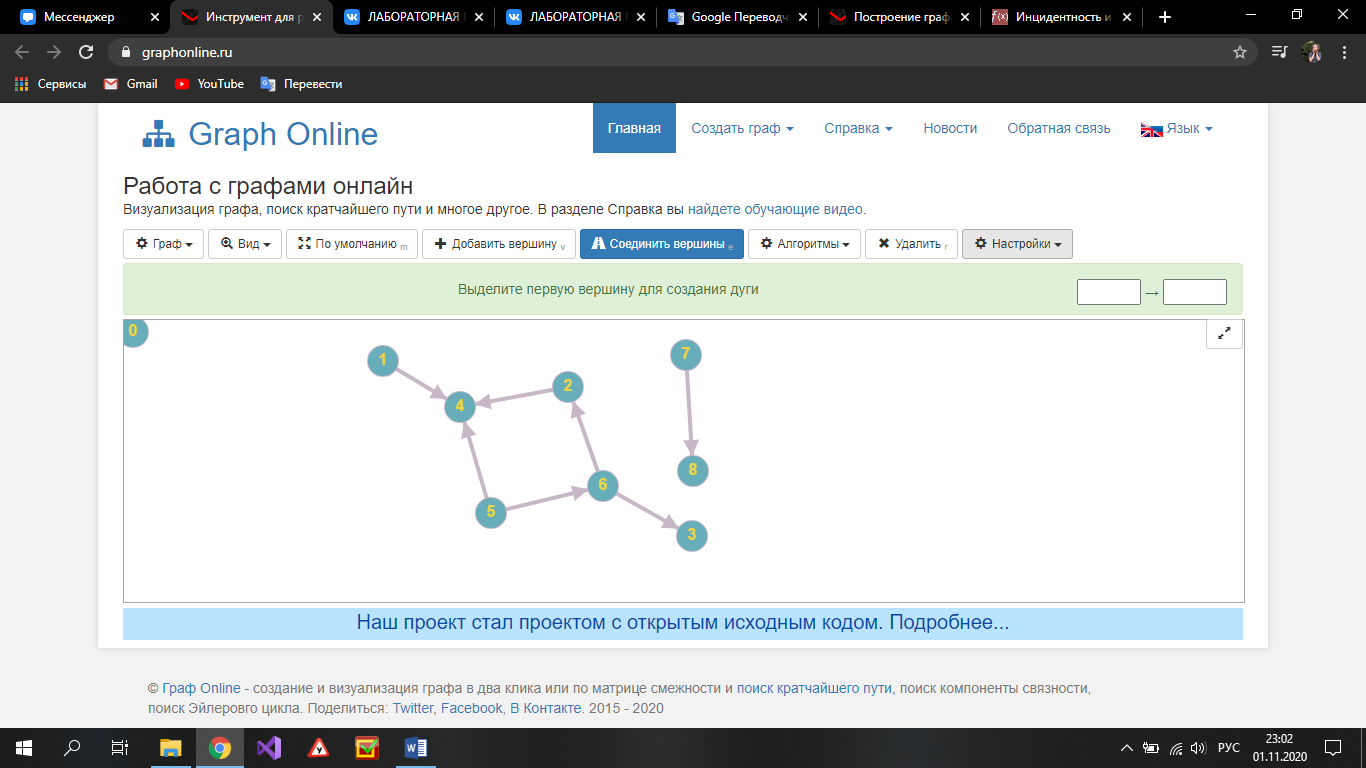
5 вершина: 2

6 вершина: 3

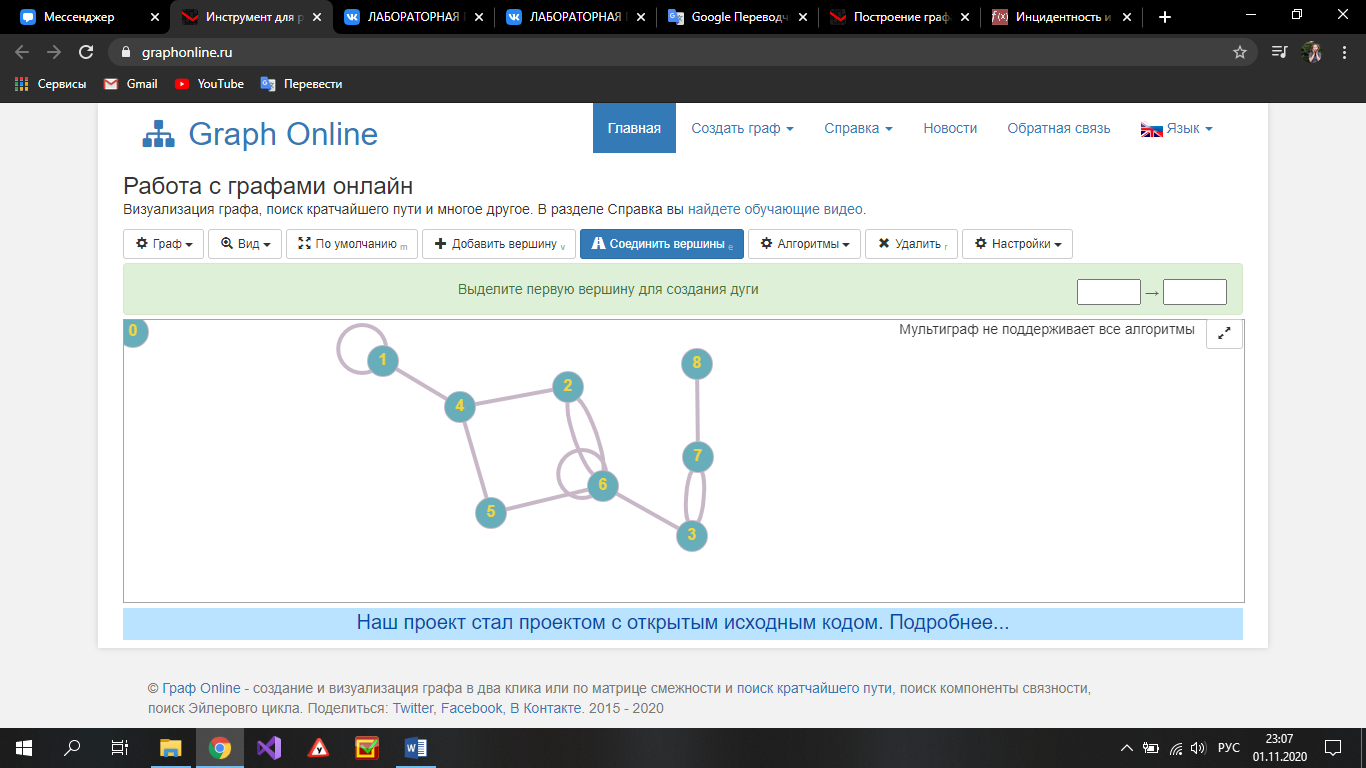
7 вершина: 1

8 вершина: 1

*Изображение ориентированного графа*



*Изображение псевдографа*



*Вопросы к лабораторной работе*(*отвечать письменно, ответы обосновывать*).

1. В каком из методов обхода графа путь в дереве поиска соответствует кратчайшему (т.е. содержащему наименьшее количество ребер) пути от вершины *s* до вершины *v*.

*Ответ:* Путь в дереве поиска соответствует кротчайшему пути от вершины s до вершины v в методе обхода в ширину, так как в обходе в ширину вершины вписываются по степени их отдаления от корня (по уровням). Следовательно, расстояние от вершины v до корня в исходном графе будет также сохраняться и в графе поиска. В графе же поиска в глубину нет четкого согласования с расстоянием от корня.

1. Для какого из обходов строится единственное (с точностью до изоморфизма) дерево поиска, а для какого можно построить их несколько.

*Ответ:* Ни для одного.

1. Какое из утверждений является верным:  
   а) Полный граф всегда является регулярным графом;  
   б) Регулярный граф всегда является полным графом.

*Ответ:* Утверждение «а) Полный граф всегда является регулярным графом» является верным. Если граф полный, то каждая из его вершин соединятся с остальными дугой. Количество таких соединений n – 1 (n – количество вершин графа), так как это простой граф и вершина не соединяется сама с собой или с другой вершиной дважды. Причем количество соединения одинаково для каждой вершины. Следовательно, такой граф регулярный.

Утверждение «б) Регулярный граф всегда является полным графом» не верно. В регулярном графе степень каждой вершины одинакова, но это еще не повод полагать, что она равно n – 1.

*Задание 2(вариант 7)*

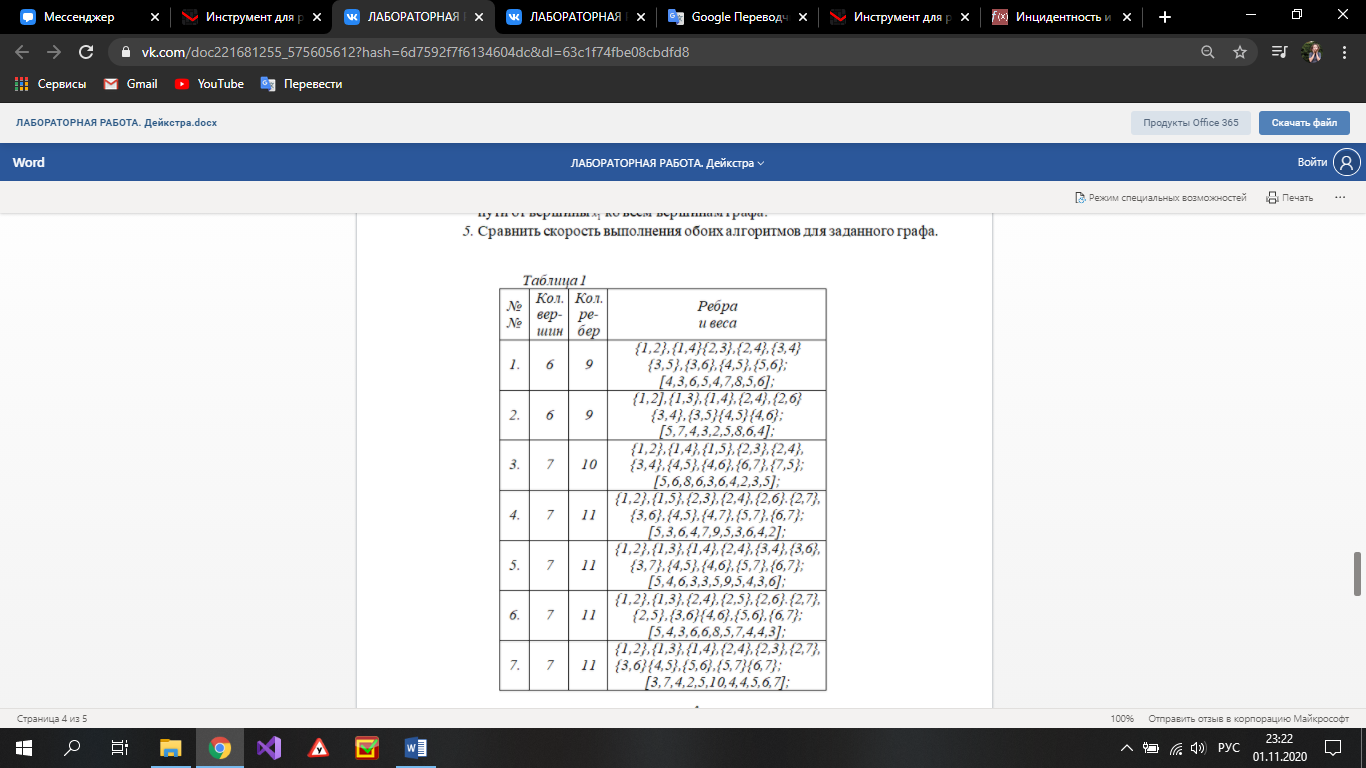
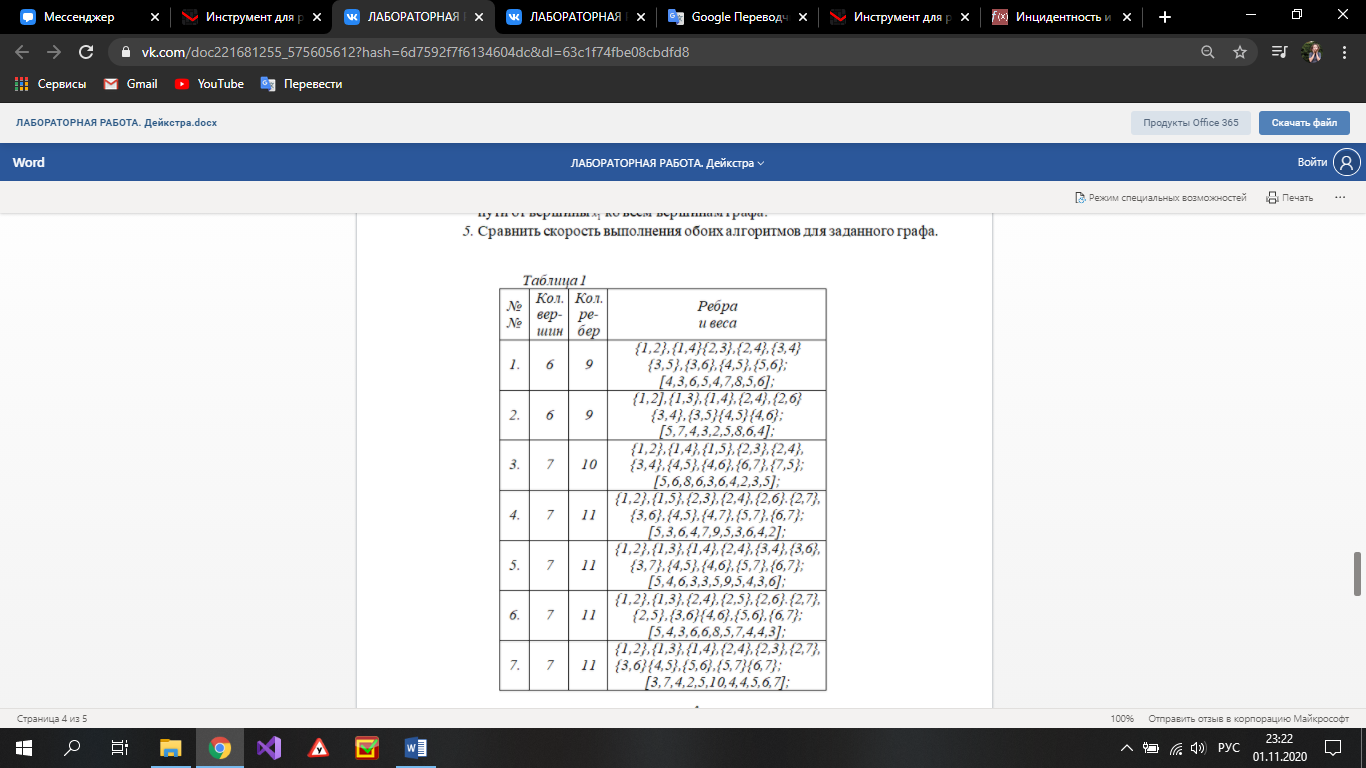
1. Нарисовать граф.

2. Программно алгоритмом Дейкстры вычислить кратчайшие пути от вершины ко всем вершинам графа. Варианты графов указаны в таблице 1. Графы заданы списком ребер, в квадратных скобках указаны веса соответствующих ребер.

3. Нарисовать таблицу пошагового выполнения алгоритма Дейкстры.

4. Программно алгоритмом Флойда-Уоршолла вычислить кратчайшие пути от вершины ко всем вершинам графа.

5. Сравнить скорость выполнения обоих алгоритмов для заданного графа.

*Код программы:*

#include <iostream>

using namespace std;

void AdjMatr(int[11][2], int[11]);

void Dijkstra(int\*\*);

void Floid(int\*\*);

int main()

{

setlocale(0, "");

int GraphEdges[11][2] = { { 1,2 },{ 1,3 },{ 1,4 },{ 2,4 },{ 2,3 },{ 2,7 },{ 3,6 },{ 4,5 },{ 5,6 },{5,7}, {6,7} };

int Weight[11] = { 3,7,4,2,5,10,4,4,5,6,7 };

cout << endl;

AdjMatr(GraphEdges, Weight);

cout << endl;

system("pause");

return 0;

}

void AdjMatr(int GraphEdges[11][2], int Weight[11])

{

int\*\* AdjacencyMatrix;

AdjacencyMatrix = new int\* [7];

for (int h = 0; h < 7; h++)

{

AdjacencyMatrix[h] = new int[7];

}

for (int i = 0; i < 7; i++)

for (int j = 0; j < 7; j++)

AdjacencyMatrix[i][j] = 0;

for (int i = 0; i < 11; i++)

{

AdjacencyMatrix[GraphEdges[i][0] - 1][GraphEdges[i][1] - 1] = Weight[i];

AdjacencyMatrix[GraphEdges[i][1] - 1][GraphEdges[i][0] - 1] = Weight[i];

}

cout << " Матрица весов:" << endl;

for (int i = 0; i < 7; i++)

{

for (int j = 0; j < 7; j++)

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

cout << endl;

}

Dijkstra(AdjacencyMatrix); //Вычисление кратчайшего пути алгоритмом Дейкстры.

cout << endl;

Floid(AdjacencyMatrix); //Вычисление кратчайшего пути алгоритмом Флойда-Уоршалла.

for (int count = 0; count < 7; count++)

delete[] AdjacencyMatrix[count];

}

void Dijkstra(int\*\* AdjacencyMatrix)

{

int ShortestPath[7] = { 0,0,0,0,0,0,0 };

int Path[7] = { 0,0,0,0,0,0,0 };

bool P[7] = { false };

int t = 0;

for (int i = 0; i < 6; i++)

{

for (int j = 0; j < 7; j++)

{

if (t == 0)

{

if (AdjacencyMatrix[t][j] != 0)

ShortestPath[j] = AdjacencyMatrix[t][j];

else ShortestPath[j] = 1000;

P[t] = true;

}

else if (AdjacencyMatrix[t][j] != 0 && AdjacencyMatrix[t][j] < ShortestPath[j] && P[j] == false)

{

if ((AdjacencyMatrix[t][j] + Path[t]) < ShortestPath[j])

ShortestPath[j] = AdjacencyMatrix[t][j] + Path[t];

}

}

int Min = ShortestPath[0];

for (int h = 1; h < 7; h++)

{

if (ShortestPath[h] < Min && P[h] == false)

{

Min = ShortestPath[h];

t = h;

}

}

P[t] = true;

Path[t] = Min;

}

cout << endl << "Кратчайшие пути от вершины 1 ко всем вершинам графа: ";

for (int i = 1; i < 7; i++)

cout << Path[i] << " ";

}

void Floid(int\*\* A)

{

for (int k = 0; k < 7; k++)

{

for (int i = 0; i < 7; i++)

{

for (int j = 0; j < 7; j++)

{

if (A[i][k] && A[k][j] && i != j)

if (A[i][k] + A[k][j] < A[i][j] || A[i][j] == 0)

A[i][j] = A[i][k] + A[k][j];

}

}

}

cout << endl << "Кратчайшие пути по Флойду-Уоршеллу:" << endl;

for (int i = 0; i < 7; ++i)

{

for (int j = 0; j < 7; ++j)

{

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

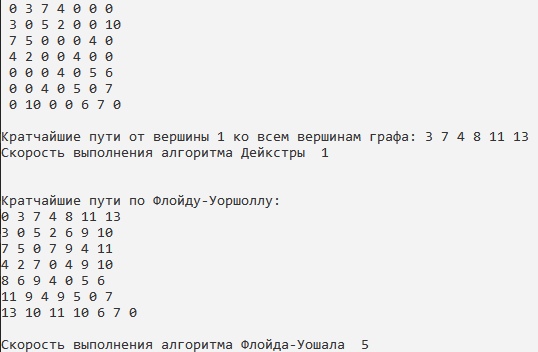
}

cout << endl;

}

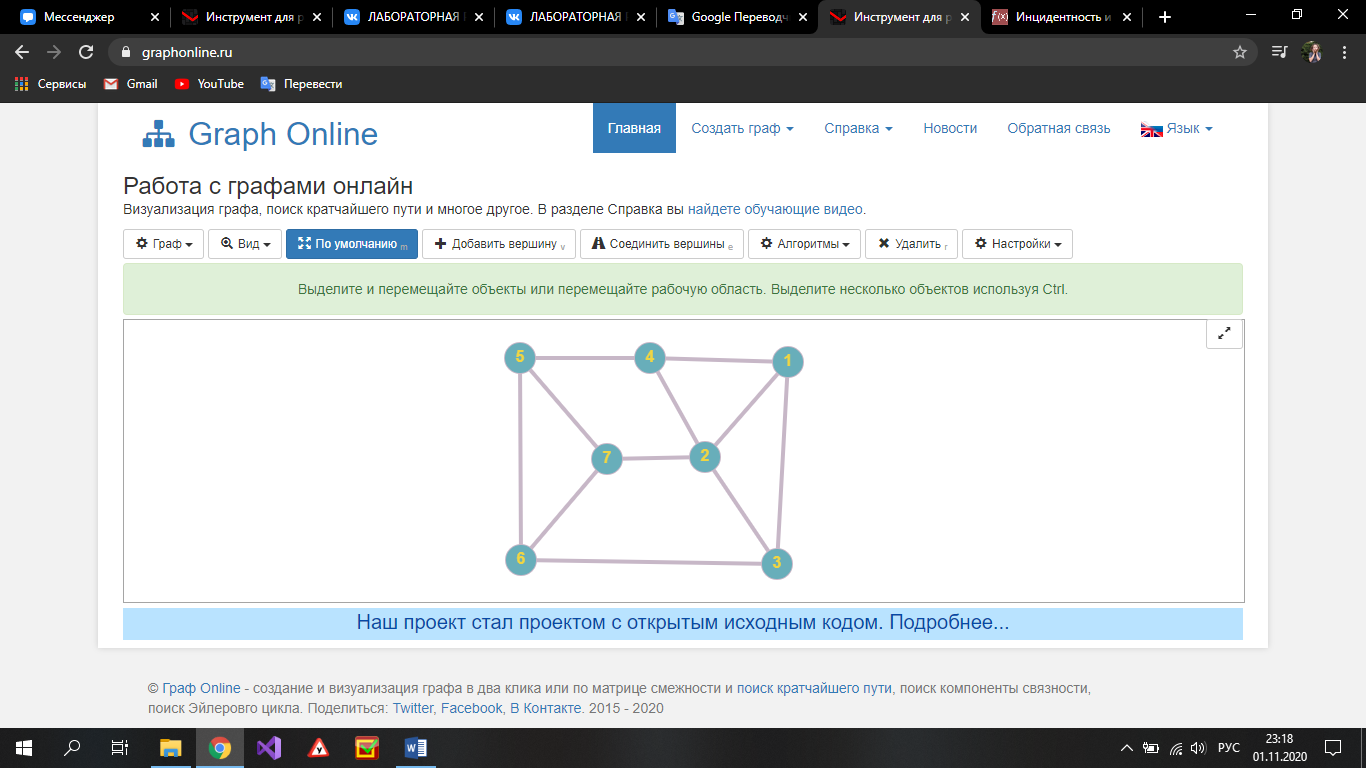
}

*Результат выполнения:*



Скорость выполнения алгоритма Дейкстры выше, чем скорость выполнения алгоритма Флойда-Уоршолла.

*Изображение графа*



*Пошаговое выполнение алгоритма Дейкстры*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № итерации |  |  |  |  |  |  |  | Кратчайшее расстояние от до |
| 1 |  |  | 7 | 4 | ∞ | ∞ | ∞ |  |
| 2 |  |  | 7 |  | ∞ | ∞ | 13 |  |
| 3 |  |  |  |  | ∞ | 11 | 13 |  |
| 4 |  |  |  |  |  | 11 | 13 |  |
| 5 |  |  |  |  |  |  | 13 |  |
| 6 |  |  |  |  |  |  |  |  |

*Вопросы к лабораторной работе (отвечать письменно, ответы обосновывать)*

1. Что такое «жадный» алгоритм и какой из указанных алгоритмов является «жадным»? Указать «О большое» для обоих алгоритмов.

*Ответ:* Жадный алгоритм — это алгоритм, который на каждом шагу делает локально наилучший выбор в надежде, что итоговое решение будет оптимальным. Алгоритм Дейкстры нахождения кратчайшего пути в графе вполне себе жадный, потому что мы на каждом шагу ищем вершину с наименьшим весом, в которой мы еще не бывали, после чего обновляем значения других вершин. При этом можно доказать, что кратчайшие пути, найденные в вершинах, являются оптимальными. Алгоритм Флойда, который тоже ищет кратчайшие пути в графе (правда, между всеми вершинами), не является примером жадного алгоритма. Флойд демонстрирует другой метод — метод динамического программирования.   
«О большое» для Дейкстры   
«О большое» для Флойда

2. Почему классический алгоритм Дейкстры не работает для отрицательных весов?

*Ответ:* Потому что он является жадным.

3. Описать алгоритм А стар (А\*) и область его применения

*Ответ:* А\* - это алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной). В процессе работы алгоритма для вершин рассчитывается функция f(v)=g(v)+h(v), где  
g(v) — наименьшая стоимость пути в v из стартовой вершины,  
h(v) — эвристическое приближение стоимости пути от v до конечной цели.  
Фактически, функция f(v) — длина пути до цели, которая складывается из пройденного расстояния g(v) и оставшегося расстояния h(v). Исходя из этого, чем меньше значение f(v), тем раньше мы откроем вершину v, так как через неё мы предположительно достигнем расстояние до цели быстрее всего. Открытые алгоритмом вершины можно хранить в очереди с приоритетом по значению f(v). А\* действует подобно алгоритму Дейкстры и просматривает среди всех маршрутов ведущих к цели сначала те, которые благодаря имеющейся информации (эвристическая функция) в данный момент являются наилучшими. Используется для поиска кратчайшего пути в графах.

*Вывод:* Усвоила базовые концепции из теории графов, научиться производить обход графа в ширину и в глубину, а также находить число компонент связности. Научилась реализовывать алгоритмы Дейкстры и Флойда-Уоршалла.