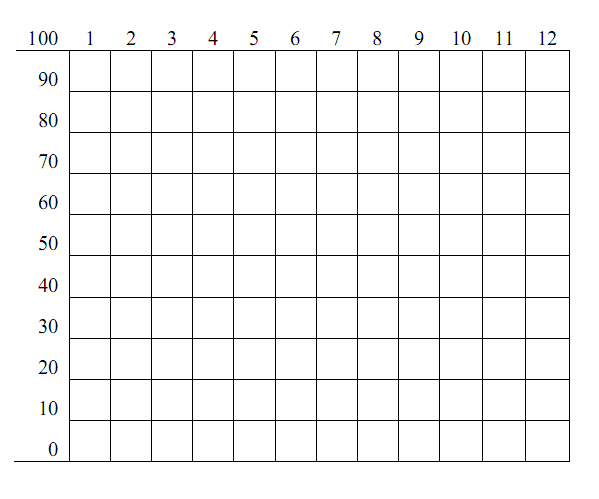
**Министерство образования и науки РФ**

**ФГБОУ ВПО «Уфимский государственный авиационный технический университет»**

**Кафедра** Высокопроизводительных вычислительных технологий и систем

****

**Реализация алгоритма поиска расстояний и пути от каждой вершины до остальных путем удалении n-2 вершин и добавлении их обратно**

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовой работе по дисциплинам**

**«Теория функций комплексных переменных» и «Комбинаторные алгоритмы»**

|  |
| --- |
| 1502.Б3.Б.10900.00 ПЗ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа МКН-308 | Фамилия И.О. | Подпись | Дата | Оценка |
| Студент | Ибатуллин И.Т. |  |  |  |
| Консультант | Житников В.П. |  |  |  |
| Принял | Газизов Р.К. |  |  |  |

**Уфа 2014**

ФГБОУ ВПО «Уфимский государственный авиационный технический университет»

Кафедра ВВТиС

**ЗАДАНИЕ**

на курсовую работу по дисциплинам

**«Теория функций комплексных переменных» и**

**«Комбинаторные алгоритмы»**

Студент: Ибатуллин Ильнур Тимерьярович Группа МКН-308

Консультант: Житников Владимир Павлович

**1. Тема курсовой работы**

Реализация алгоритма поиска расстояний и путей от каждой вершины до остальных.

**2. Основное содержание**

* 1. Требуется написать алгоритм, который путем удаления вершин, не теряя целостность и «истинность», а затем путем добавления находит полную матрицу смежности. Данный алгоритм предназначен для поиска кратчайшего расстояния и пути в графах, полученных из реальных городов.
  2. Выполнить программную реализацию разработанного алгоритма на языке С++ в среде разработки Microsoft Visual Studio 2013(или 2012).
  3. Оформить пояснительную записку к курсовой работе.

**3. Требования к оформлению материалов работы**

3.1. Требования к программе

* Программа должна быть написана на С++, используя STL, приветствуется хорошее оформление кода, наличие комментариев

3.2. Требования к оформлению пояснительной записки

Пояснительная записка к курсовой работе должна быть оформлена в текстовом процессоре Microsoft Word в соответствии с требованиями стандарта СТО УГАТУ 016-2007 и содержать

* титульный лист,
* задание на курсовую работу,
* содержание,
* введение,
* главу, описывающую особенности программной реализации алгоритма,
* главу, описывающую правила работы с программой и примеры ее работы,
* заключение,
* список литературы,
* приложение, содержащее листинг разработанной программы.

**4.** **Список литературы**

Джимми Уэйлс, Ларри Сэнгер. (n.d.). *Википедия*. Retrieved from https://www.wikipedia.org/

Кнут, Д. (2006). *Искусство программирования, том 1. Основные алгоритмы.*

Томас Кормен, Ч. Л. (1990). *Алгоритмы: построение и анализ.*

Дата выдачи задания Дата окончания

\_\_ \_\_\_\_\_\_\_\_\_ 201\_ г. \_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ 201\_ г.

Консультант \_\_\_\_\_\_\_\_\_ Житников В.П.

Принял \_\_\_\_\_\_\_\_\_ Газизов Р.К.

Оглавление

[Аннотация 5](#_Toc407628587)

[1. Теоритические основы 6](#_Toc407628588)

[1.1 Графы. Свойства графов 6](#_Toc407628589)

[2. Особенности программной реализации алгоритма 8](#_Toc407628590)

[2.1. Описание первого этапа – удаления вершин. 8](#_Toc407628591)

[2.2. Описание второго этапа – добавление вершин. 8](#_Toc407628592)

[2.3. Алгоритм Флойда. 8](#_Toc407628593)

[3. Запуск, проверка различных плотностей графов 9](#_Toc407628594)

[3.1. Сравнение двух алгоритмов. 9](#_Toc407628595)

[3.2 Сравнение двух алгоритмов на разряженных графах. 9](#_Toc407628596)

[4. Результаты и выводы 11](#_Toc407628597)

[5. Листинг программы 12](#_Toc407628598)

[литература 16](#_Toc407628599)

## Аннотация

Курсовая работа состоит из следующих разделов:

* в первом разделе содержатся теоретические сведения о графах, различных свойствах графов;
* второй раздел содержит программную реализацию используемых методов – листинг программы, написанной на языке C++, с подробными комментариями;
* третий раздел включает визуализацию результатов полученных значений, составленную средствами разработки Microsoft Visual Studio;
* результаты и выводы.

## 1. Теоритические основы

### Граф. Свойства графа

Граф - основной объект изучения математической теории графов, совокупность непустого множества вершин и наборов пар вершин (связей между вершинами).

Объекты представляются как вершины, или узлы графа, а связи – как дуги, или рёбра. Для разных областей применения виды графов могут различаться направленностью, ограничениями на количество связей и дополнительными данными о вершинах или рёбрах.

Многие структуры, представляющие практический интерес в математике и информатике, могут быть представлены графами. Например, строение Википедии можно смоделировать при помощи ориентированного графа, в котором вершины – это статьи, а дуги (ориентированные рёбра) –гиперссылки (тематическая карта).

Граф, или неориентированный граф *G* – это упорядоченная пара *G := (V, E)*, где *V* – это непустое множество вершин или узлов, а *E* – множество пар (в случае неориентированного графа – неупорядоченных) вершин, называемых рёбрами.

*V* (а значит и, *E*, иначе оно было бы мультимножеством) обычно считаются конечными множествами. Многие результаты, полученные для конечных графов, неверны (или каким-либо образом отличаются) для бесконечных графов, поскольку не все утверждения, имеющие место для конечных совокупностей, выполняются в случае бесконечных множеств.

Вершины и рёбра графа называются также элементами графа, число вершин в графе |*V*| – порядком, число рёбер |*E*| – размером графа.

Вершины *u* и *v* называются концевыми вершинами (или просто концами) ребра *e* = {*u*, *v*}. Ребро, в свою очередь, соединяет эти вершины. Две концевые вершины одного и того же ребра называются соседними.

Два ребра называются смежными, если они имеют общую концевую вершину.

Два ребра называются кратными, если множества их концевых вершин совпадают.

Ребро называется петлёй, если его концы совпадают, то есть *e* = {*v, v*}.

Степенью deg(*V*) вершины *V* называют количество инцидентных ей рёбер (при этом петли считают дважды).

Вершина называется изолированной, если она не является концом ни для одного ребра; висячей (или листом), если она является концом ровно одного ребра.

Ориентированный граф (сокращённо орграф) G – это упорядоченная пара *G* := (*V*, *A*), где *V* – непустое множество вершин или узлов, и *A* — множество (упорядоченных) пар различных вершин, называемых дугами или ориентированными рёбрами.

Дуга – это упорядоченная пара вершин (*v*, *w*), где вершину v называют началом, а *w* – концом дуги. Можно сказать, что дуга *v* → *w* ведёт от вершины *v* к вершине *w*.

Изоморфные графы

Граф *G* называется изоморфным графу *H*, если существует биекция *f* из множества вершин графа *G* в множество вершин графа *H*, обладающая следующим свойством: если в графе *G* есть ребро из вершины *A* в вершину *B*, то в графе *H* должно быть ребро из вершины *f*(*A*) в вершину *f*(*B*) и наоборот – если в графе *H* есть ребро из вершины *A* в вершину *B*, то в графе *G* должно быть ребро из вершины *f-*1(*A*) в вершину *f-*1(*B*). В случае ориентированного графа эта биекция также должна сохранять ориентацию ребра. В случае взвешенного графа биекция также должна сохранять вес ребра.

Прочие связанные определения

Маршрутом в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершиной ребром. Цепью называется маршрут без повторяющихся рёбер. Простой цепью называется маршрут без повторяющихся вершин (откуда следует, что в простой цепи нет повторяющихся рёбер).

Ориентированным маршрутом (или путём) в орграфе называют конечную последовательность вершин и дуг, в которой каждый элемент инцидентен предыдущему и последующему.

## 2. Особенности программной реализации алгоритма

### 2.1. Описание первого этапа – удаления вершин

По всему графу ищем вершину с наименьшей степенью вершин. Если таковых несколько, выбираем любой из них. Далее, определяем, с какими вершинами связан. Так как мы собираемся удалить эту вершину, мы должны предусмотреть, чтобы граф при этом не потерял связанность и расстояние между другими вершинами. Допустим, имеется полный связанный граф из 3 вершин. Расстояние из 1 в 2 – 1, из 1 в 3 – 2, из 2 в 3 – 4. Мы хотим удалить вершину 1. При этом мы должны помнить, что кратчайшее расстояние из 2 вершины в 3 равна 3, а не 4. В случае, если мы просто удалим 1 вершину, то потеряем данные об этом. Для этого задействуем правило треугольника. Для любых 2 смежных вершин исходной проверяем неравенство: что короче, расстояние между двумя вершинами, или расстояние между одной вершиной и исходной плюс расстояние между исходной и второй? Именно после этого мы можем быть уверены, что граф после удаления исходной вершины останется связанным, кратчайшие расстояния никуда не исчезнут. Таким образом, мы удалим все вершины, кроме двух последних, нет смысла их удалять. На этом этапе можно сказать, что расстояние между этими двумя вершинами является кратчайшим.

### 2.2. Описание второго этапа – добавление вершин

После того, как у нас осталось только 2 вершины, наступает пора добавлять вершины обратно в граф. По аналогии математической индукции будем добавлять вершины в обратно порядке удаления. База индукции: 2 вершины, расстояние между ними кратчайшее. Шаг индукции для k: допустим, что для k вершин выполнено условие кратчайших расстояний. Добавим новую вершину. Теперь применим правило треугольника для данной вершины. То есть, проверяем, а не короче ли путь для 2 вершин из базы через новую вершину. Когда всё проверили, мы найдем кратчайшие расстояния от новой вершины до всех остальных. Таким образом, мы построили граф на 1 вершину больше, при этом все накладываемые условия выполняются.

### 2.3. Алгоритм Флойда

Аналогичную задачу решает алгоритм Флойда – находит кратчайшие расстояния от всех вершин до всех остальных. Принцип этого алгоритма заключается в том же: проверяем правило треугольника для всех возможных треугольник в графе. Поэтому сложность алгоритма равна O(N3). Именно с этим алгоритмом имеет смысл проверяет актуальность нашего решения.

## 3. Запуск, проверка различных плотностей графов

### 3.1. Сравнение двух алгоритмов

Для этого запустим программу и для последовательно будем увеличивать количество вершин. Время учитывает только процесс нахождения кратчайших расстояний, игнорируя процесс создания графа, очистки, инициализации переменных.

Рисунок 1

Рисунок 2

### 3.2 Сравнение двух алгоритмов на разряженных графах

Данный случай интерес тем, что в реальных городах именно такая ситуация. Обычно один перекресток не связан более чем 4 перекрестками. В 1 случае у нас именно такая ситуация и была. Для графа размерности 1300 вершин, каждый перекресток в среднем был связан с 520 другими перекрестками, что является не возможным.

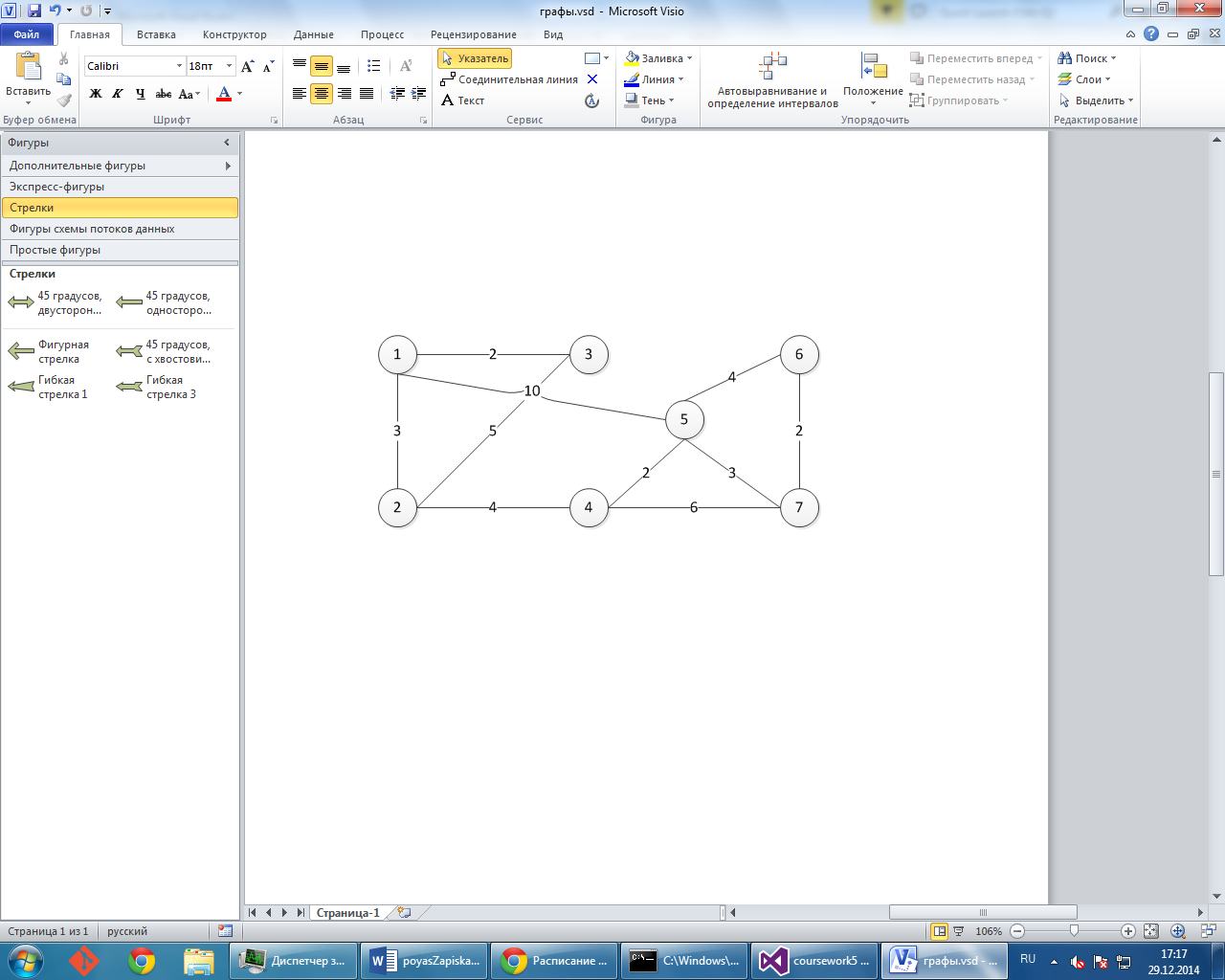
Рисунок 3

Рисунок 4

Как видно из графиков, на графах, приближенных к реальности, новый алгоритм отрабатывает поставленную задачу быстрее, чем всем известный алгоритм Флойда.

### 3.3 Пример работы алгоритма на примере конкретного графа

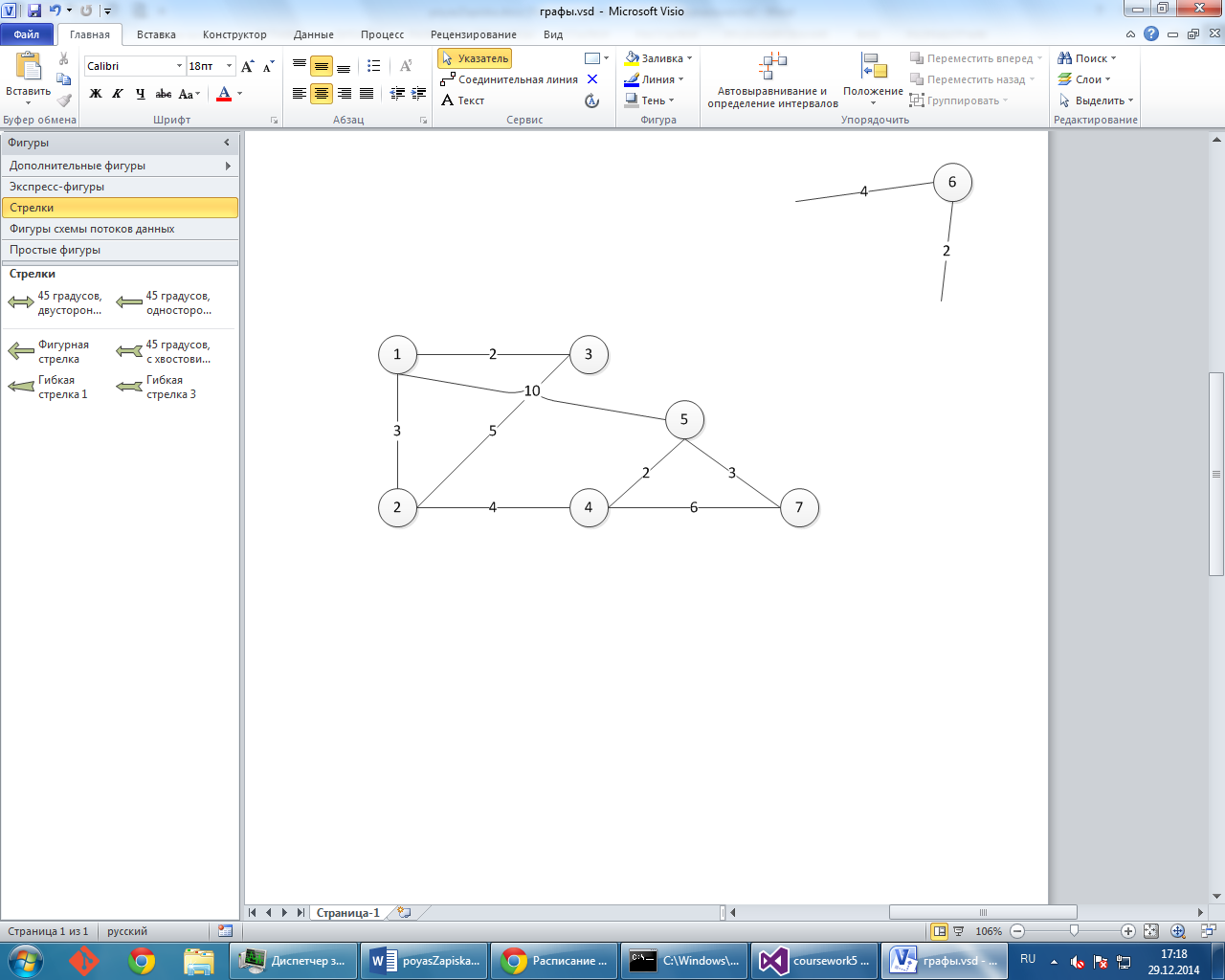
Имеется граф:



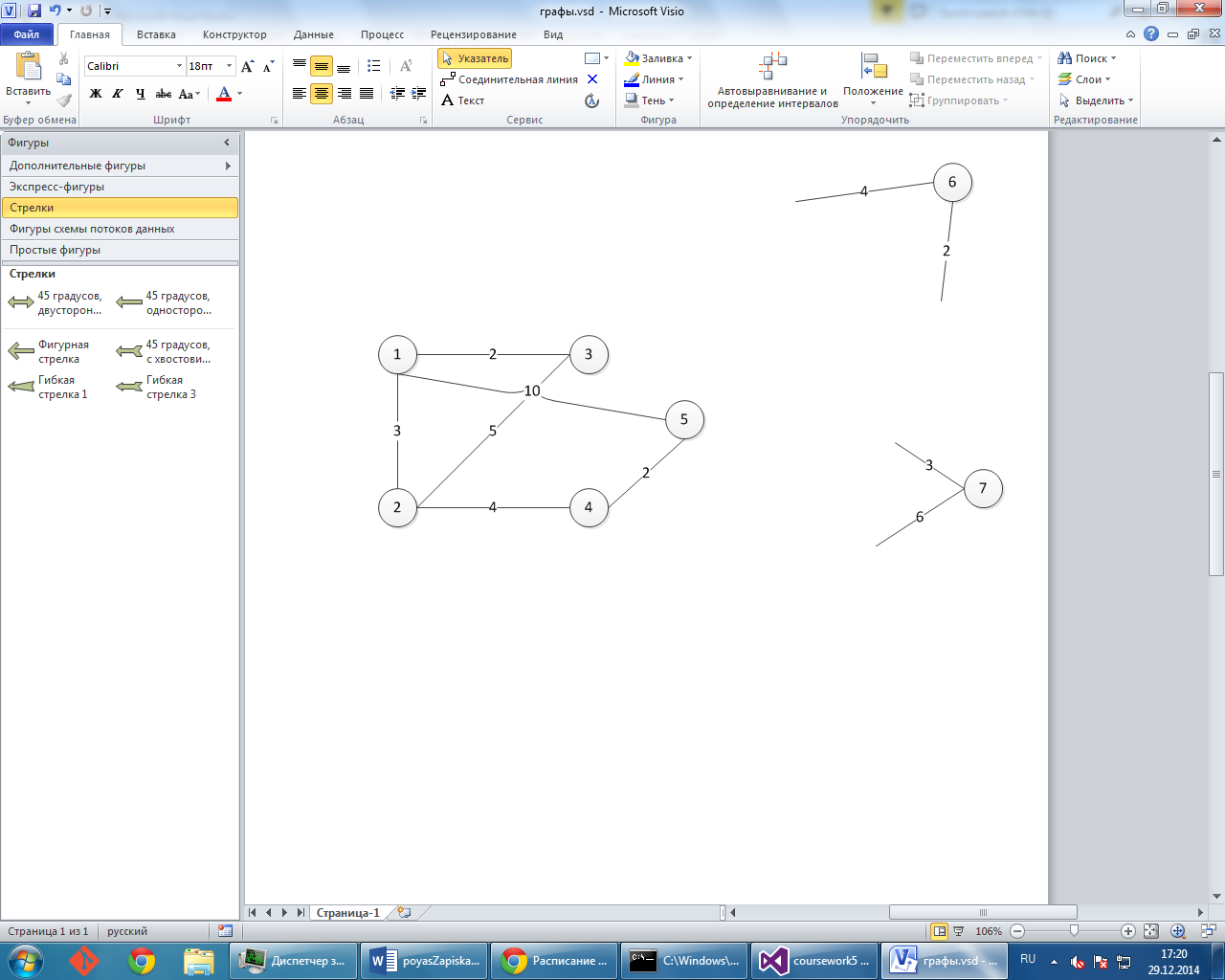
Покажем работу нашего алгоритма на примере этого графа.

Сперва начинаем процесс удаления вершин. У вершин 3 и 6 по 2 смежных вершин. Так как нам без разницы, какую удалять, начнем с 6.

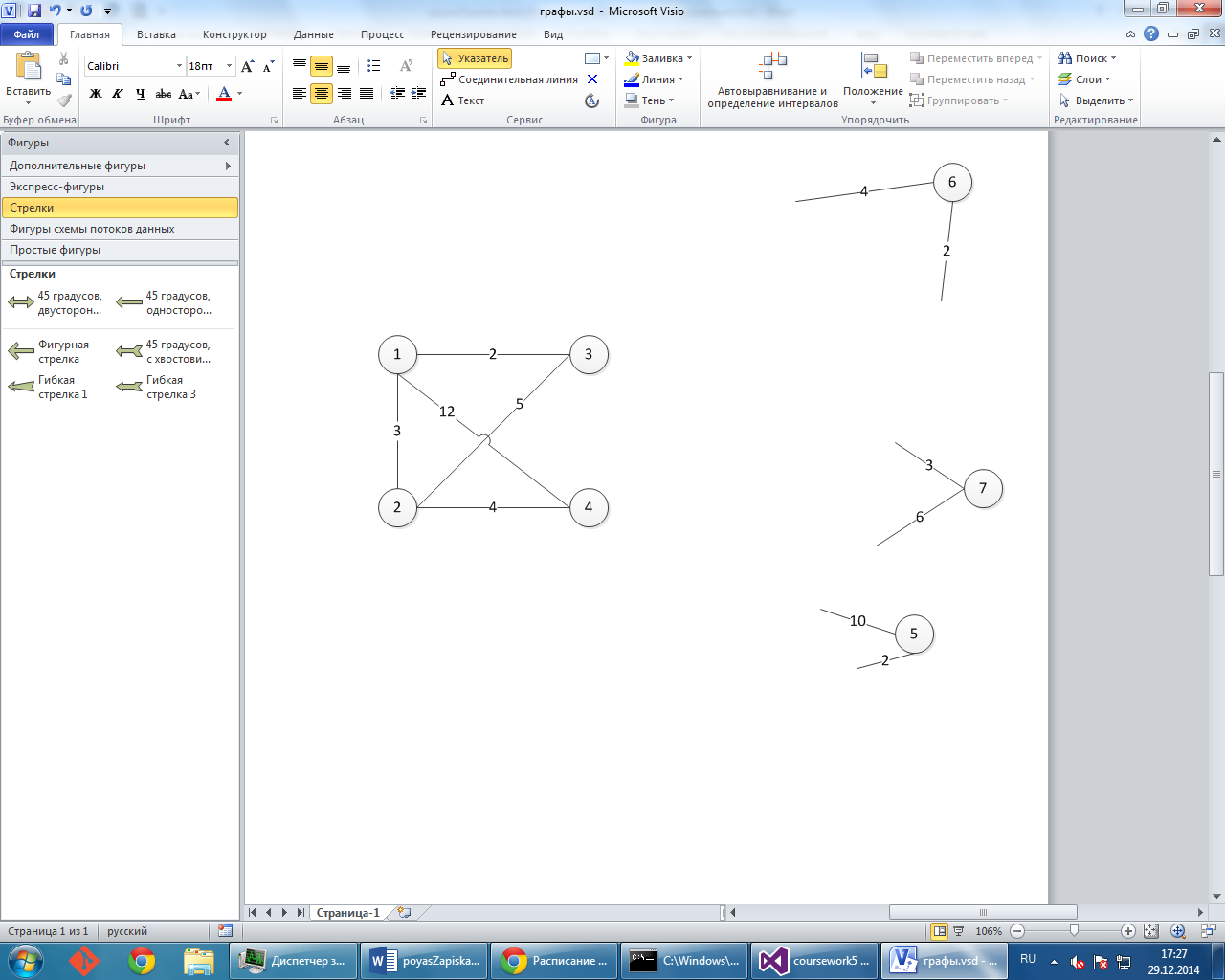
Получили следующее состояние:



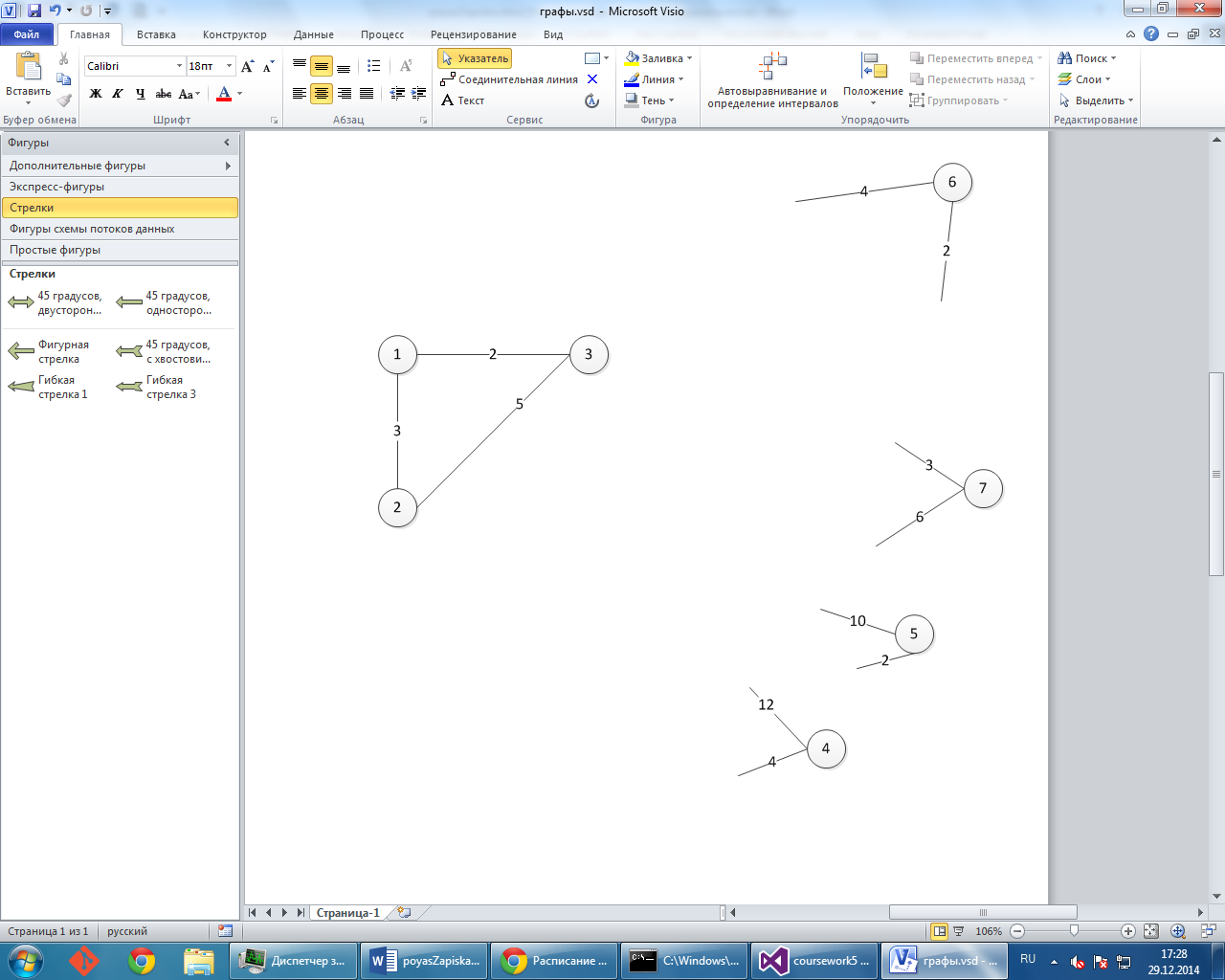
Теперь вершины 3 и 7 имеют степень равную 2. По традиции, удалим старшего:



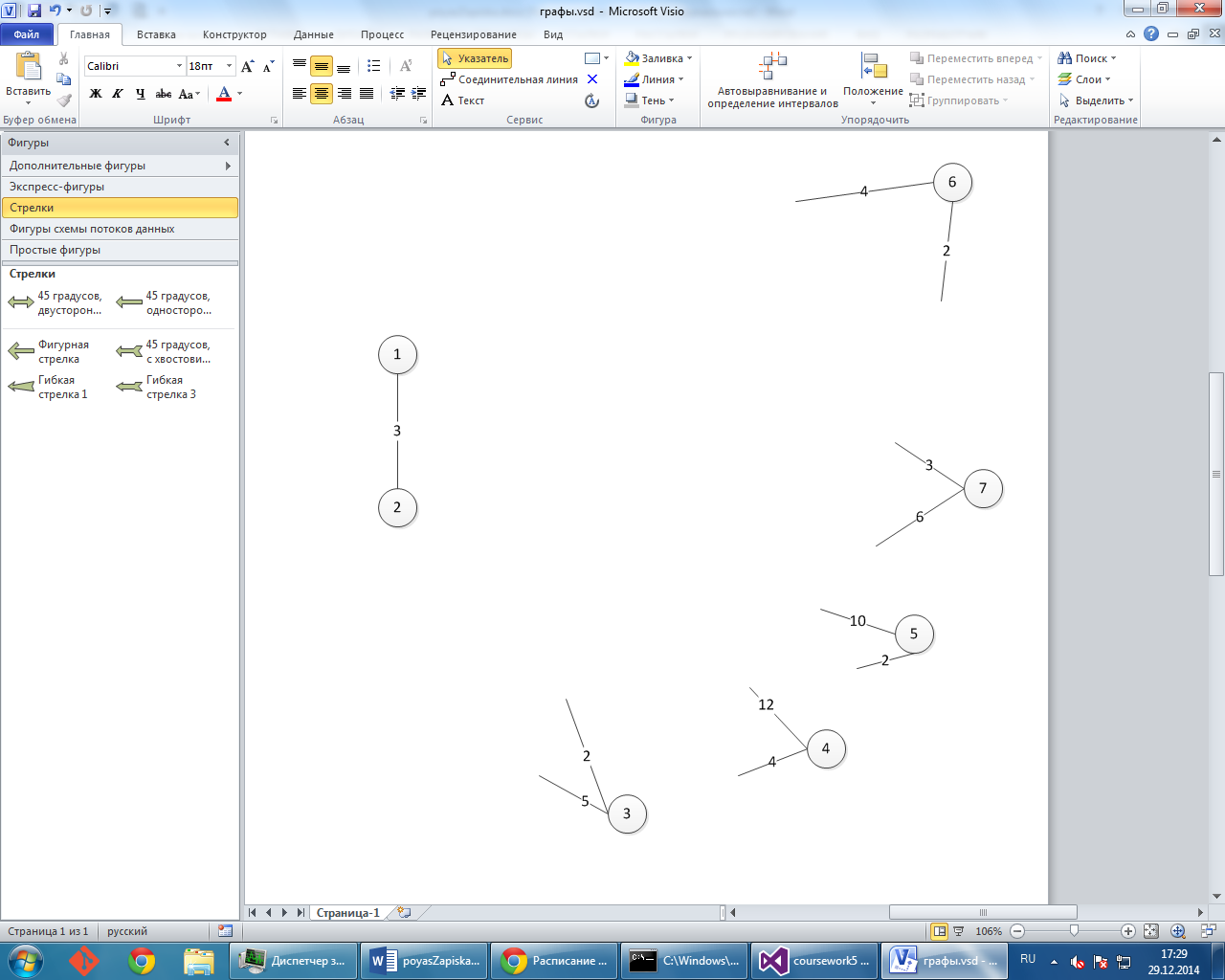
Теперь место 7 вершины занимает вершина 5 со степенью 2, соответственно. Самое интересное начинается здесь. Дело в том, что для вершины 5 смежными являются вершины 1 и 4, ребра между которыми не существует. Но на практике в таких случаях весом ребра ставят очень большое число, чтобы было заведомо больше имеющихся. В нашем случае это 99. Поэтому, так как расстояние между 1 и 4 больше чем сумма расстояний от 1 до 5 и от 5 до 4. Поэтому расстояние между 1 и 4 вершинами определится в 12.



Далее, удаляем вершину 4. Ничего внутри не меняется.



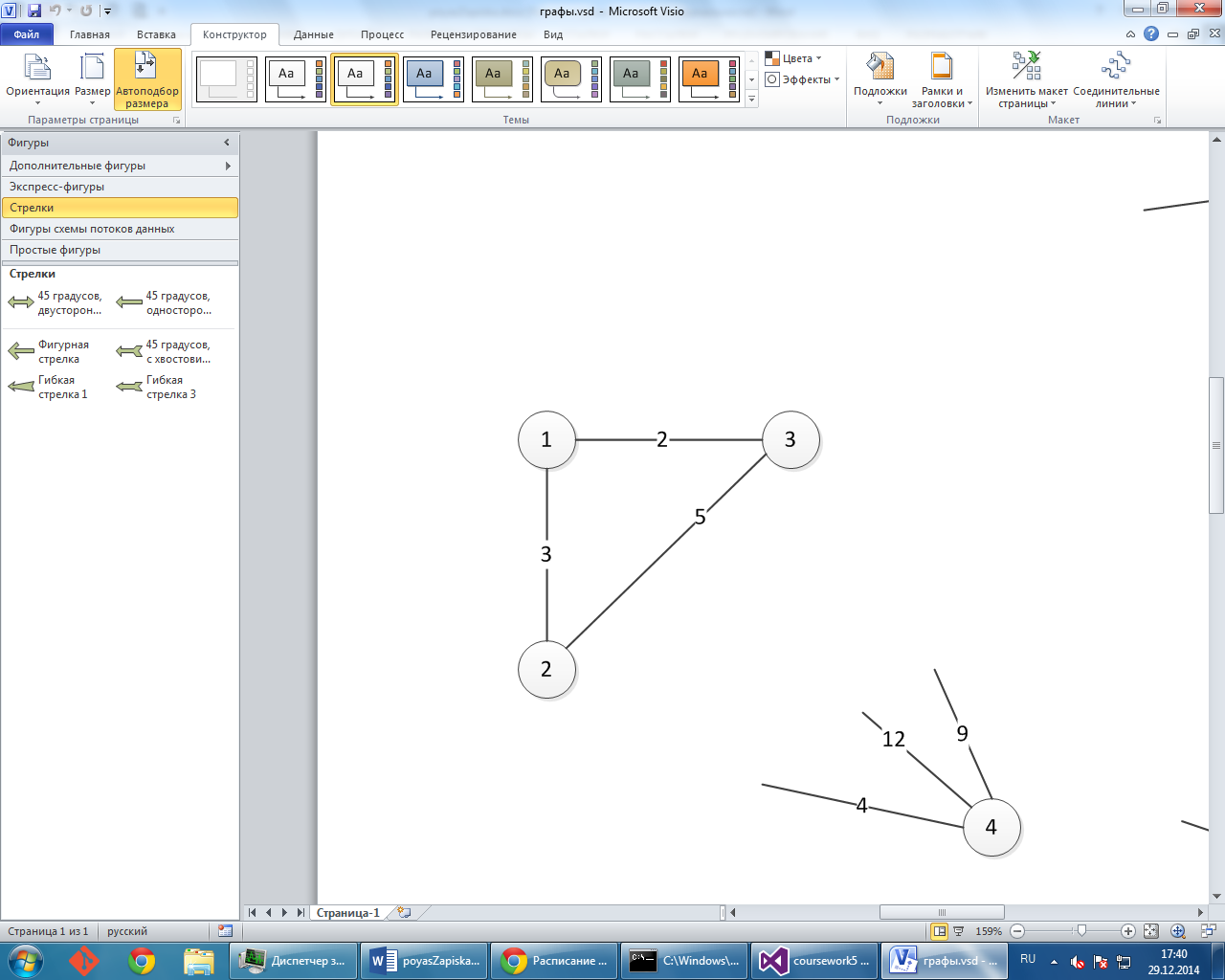
Последним удалим вершину 3.



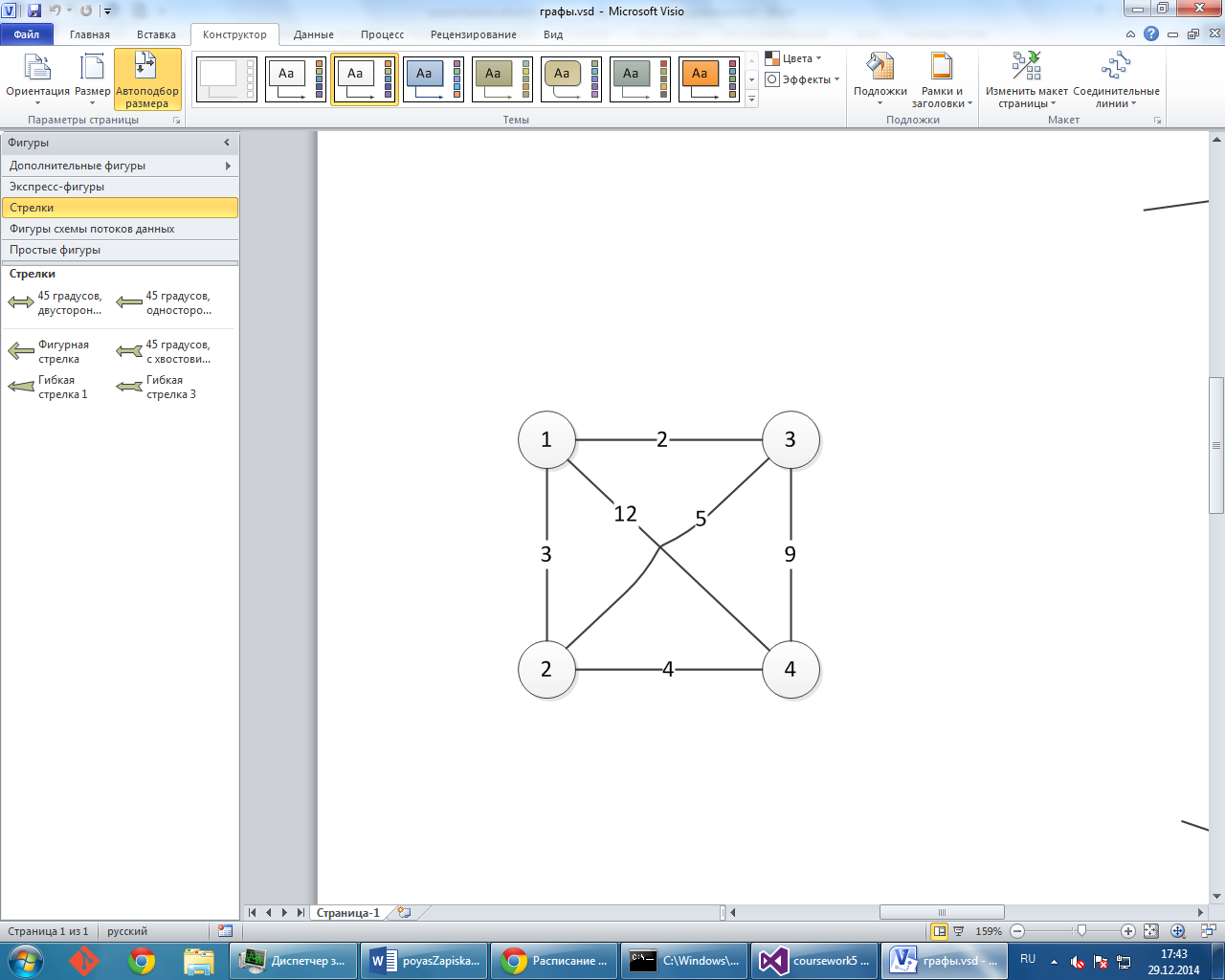
Итак, это расстояние является самым коротким между вершинами 1 и 2.

Теперь начнем этап добавления вершин в обратном порядке.

Добавление вершины 3:

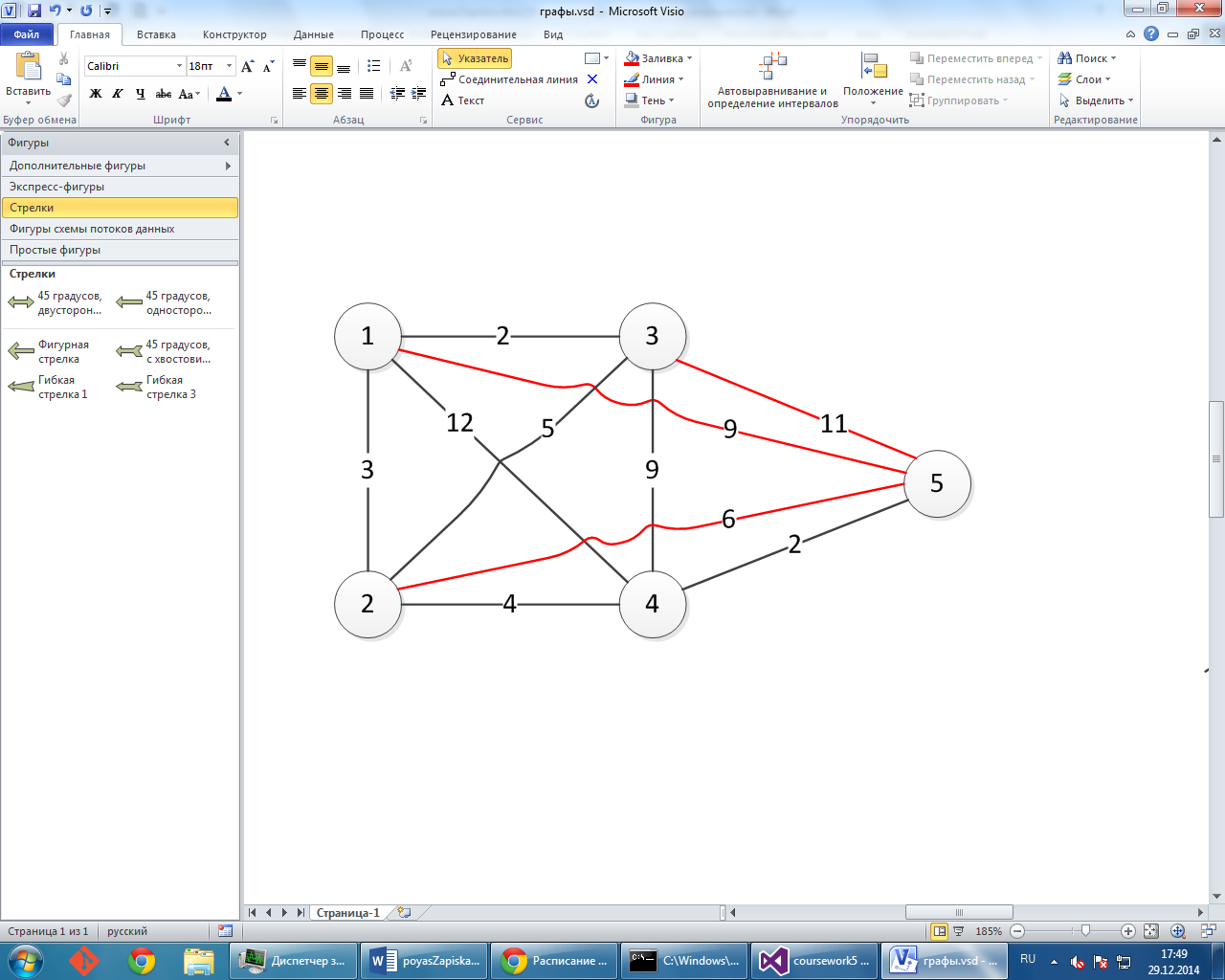


Добавление вершины 4. Вот тут опять происходит кое-что интересное. Ребра между 3 и 4 не было, то есть было равно 99. Теперь же, мы его оптимизируем:



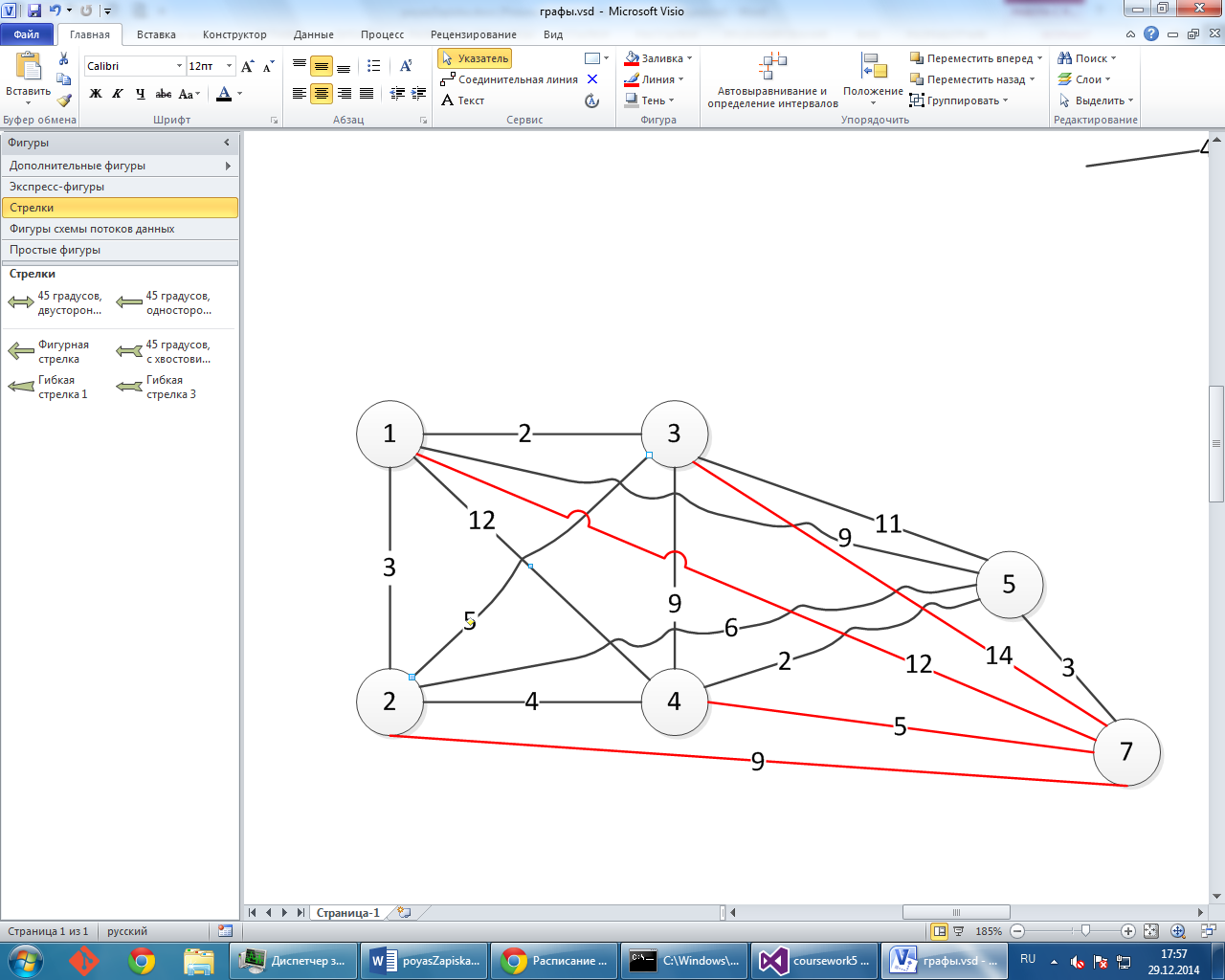
Вес ребра 3 – 4 стал равен 9. Это путь через 2, 3.

Добавляем вершину 5. Новые ребра – красным цветом.



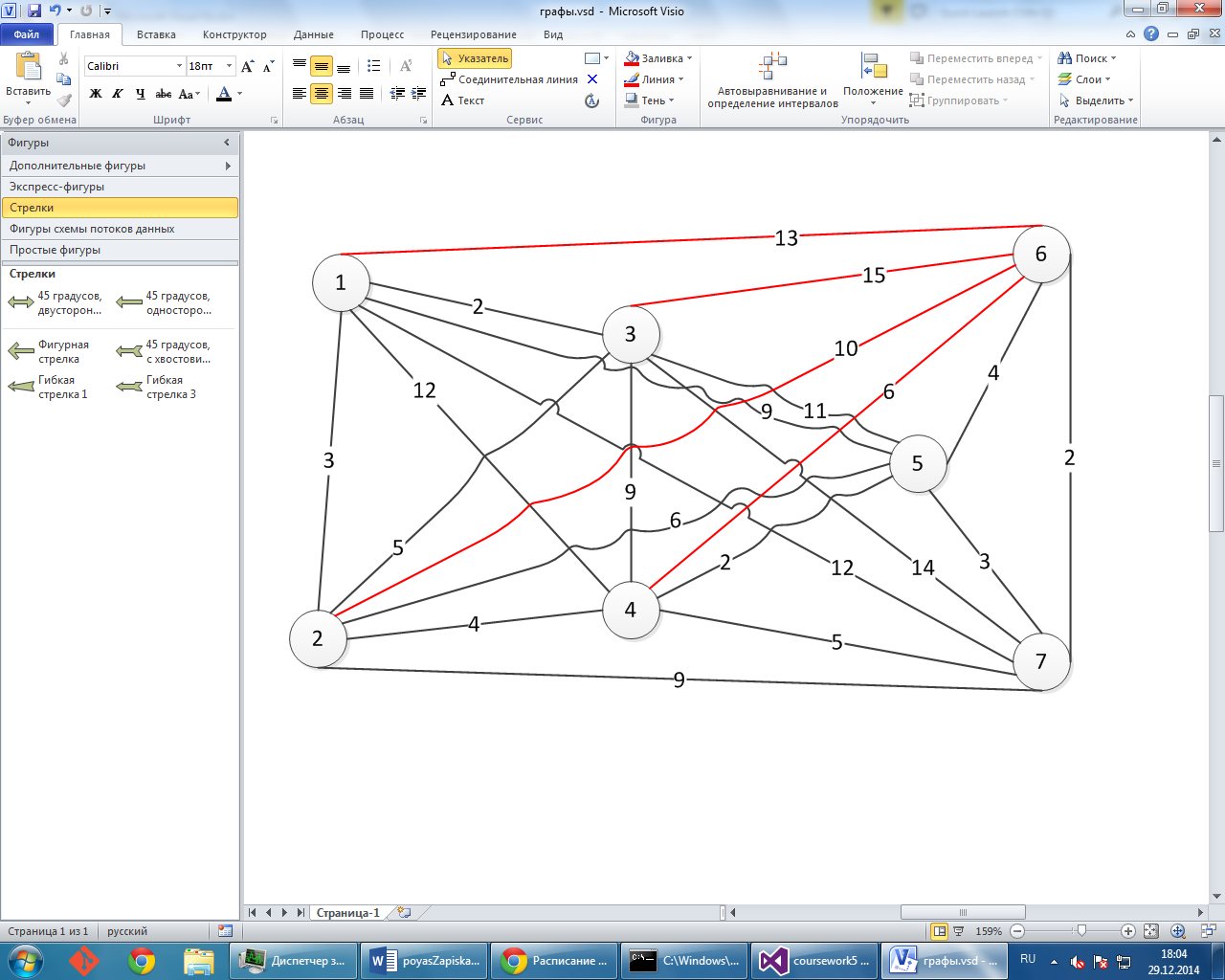
Ещё один интересный момент: ребро между 1 и 5 мы уже вычисляли, но сейчас становится понятно, что есть более короткое расстояние.

Добавляем вершину 7



Почти все ребра обновились. Результат на лицо.

Добавляем последнюю вершину – 6:



На этом алгоритм заканчивается и выдает получившийся результат.

Дебаг код для данного графа:

Starting delete procedure:

deleting vertex: 6

after deleting the vertex:

0 3 2 99 10 99 99

3 0 5 4 99 99 99

2 5 0 99 99 99 99

99 4 99 0 2 99 6

10 99 99 2 0 4 3

99 99 99 99 4 0 2

99 99 99 6 3 2 0

================================

deleting vertex: 7

after deleting the vertex:

0 3 2 99 10 99 99

3 0 5 4 99 99 99

2 5 0 99 99 99 99

99 4 99 0 2 99 6

10 99 99 2 0 4 3

99 99 99 99 4 0 2

99 99 99 6 3 2 0

================================

deleting vertex: 5

after deleting the vertex:

0 3 2 12 10 99 99

3 0 5 4 99 99 99

2 5 0 99 99 99 99

12 4 99 0 2 99 6

10 99 99 2 0 4 3

99 99 99 99 4 0 2

99 99 99 6 3 2 0

================================

deleting vertex: 4

after deleting the vertex:

0 3 2 12 10 99 99

3 0 5 4 99 99 99

2 5 0 99 99 99 99

12 4 99 0 2 99 6

10 99 99 2 0 4 3

99 99 99 99 4 0 2

99 99 99 6 3 2 0

================================

deleting vertex: 3

after deleting the vertex:

0 3 2 12 10 99 99

3 0 5 4 99 99 99

2 5 0 99 99 99 99

12 4 99 0 2 99 6

10 99 99 2 0 4 3

99 99 99 99 4 0 2

99 99 99 6 3 2 0

================================

================================

starting proccess of adding vertexes

Adding the vertex: 3

after adding the vertex:

0 3 2 12 10 99 99

3 0 5 4 99 99 99

2 5 0 99 99 99 99

12 4 99 0 2 99 6

10 99 99 2 0 4 3

99 99 99 99 4 0 2

99 99 99 6 3 2 0

================================

Adding the vertex: 4

after adding the vertex:

0 3 2 7 10 99 99

3 0 5 4 99 99 99

2 5 0 9 99 99 99

7 4 9 0 2 99 6

10 99 99 2 0 4 3

99 99 99 99 4 0 2

99 99 99 6 3 2 0

================================

Adding the vertex: 5

after adding the vertex:

0 3 2 7 9 99 99

3 0 5 4 6 99 99

2 5 0 9 11 99 99

7 4 9 0 2 99 6

9 6 11 2 0 4 3

99 99 99 99 4 0 2

99 99 99 6 3 2 0

================================

Adding the vertex: 7

after adding the vertex:

0 3 2 7 9 99 12

3 0 5 4 6 99 9

2 5 0 9 11 99 14

7 4 9 0 2 99 5

9 6 11 2 0 4 3

99 99 99 99 4 0 2

12 9 14 5 3 2 0

================================

Adding the vertex: 6

after adding the vertex:

0 3 2 7 9 13 12

3 0 5 4 6 10 9

2 5 0 9 11 15 14

7 4 9 0 2 6 5

9 6 11 2 0 4 3

13 10 15 6 4 0 2

12 9 14 5 3 2 0

================================

new algorithm:

0 3 2 7 9 13 12

3 0 5 4 6 10 9

2 5 0 9 11 15 14

7 4 9 0 2 6 5

9 6 11 2 0 4 3

13 10 15 6 4 0 2

12 9 14 5 3 2 0

## 4. Результаты и выводы

При написании данной работы выполнены следующие задания:

1. Разработана программа на С++, решающая поставленную задачу.
2. Выполнен анализ двух алгоритмов, сравнение на различных входных данных, на двух различных типов графов. Показано ускорение нового алгоритма. Приведен пример работы алгоритма на конкретном графе

## 5. Листинг программы

#include <stdio.h>

#include <cmath>

#include <vector>

#include <iostream>

#include <algorithm>

#include <time.h>

#define WR printf

#define FOR(i, k, n) for(long i = (k); i <= (n); i ++)

using namespace std;

//матрицы расстояний и послдователей

vector< vector <long> > A, S;

//массив стостояний - удалена ли вершина?

vector<long> removed;

//Массив соседей для конкретной вершины

vector<long> neighbohoor;

//Массив БАЗЫ. БАЗА - элементы, в котором точно установлено кратчайшие пути

vector<long> base;

//Размерность

long N;

//Максимальное отличние расстояний между городами

const long gMax = 100;

//Расстояние, если между вершинами графа нету ребер.

const long pMax = 99;

//Ищем вершину с минимальным количеством смежных вершин с остальными вершинами

//которые ещё не были удалены

long findMinVert()

{

long Min = N, iMin;

//Находим первую неудаленную вершину. Ставим его начальной точкой приближения

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

if (removed[i] == 0)

{

iMin = i;

break;

}

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

{

if (removed[i] == 0)

{

long count = 0;

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

{

if (removed[j] == 0 && A[i][j] < pMax)

count++;

}

if (count <= Min){

Min = count;

iMin = i;

}

}

}

return iMin;

}

//Генерация графа, точнее матрицы

//Граф неориентированный

void generation()

{

srand(time(NULL));

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

{

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

{

if (rand() % 100 > 40) {

A[i][j] = pMax;

A[j][i] = pMax;

}

else {

A[i][j] = rand() % gMax + 51;

A[j][i] = A[i][j];

}

}

}

}

//Генерация графа, приближенного к реальному городу

void generationCity()

{

srand(time(NULL));

for (int i = 1; i <= N; i++) A[i][i] = 0;

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

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

if (j - i > 2) {

A[i][j] = pMax;

A[j][i] = pMax;

}

else {

A[i][j] = rand() % gMax + 51;

A[j][i] = A[i][j];

}

}

//Здесь мы инициализируем матрицы

void init()

{

//Очистка переменных

//cin >> N;

N = 7;

A.clear();

S.clear();

removed.clear();

base.clear();

A.resize(N + 1);

S.resize(N + 1);

removed.resize(N + 1, 0);

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

{

A[i].resize(N + 1);

S[i].resize(N + 1, 0);

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

{

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

A[i][j] = pMax;

}

}

int aa[7][7] = {

{ 0, 3, 2, pMax, 10, pMax, pMax },

{ 3, 0, 5, 4, pMax, pMax, pMax },

{ 2, 5, 0, pMax, pMax, pMax, pMax },

{ pMax, 4, pMax, 0, 2, pMax, 6 },

{ 10, pMax, pMax, 2, 0, 4, 3 },

{ pMax, pMax, pMax, pMax, 4, 0, 2 },

{ pMax, pMax, pMax, 6, 3, 2, 0 } };

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

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

A[i + 1][j + 1] = aa[i][j];

//generationCity();

}

// Собственно говоря, само решение

void solve()

{

//Процесс удаления вершин

//p - порядковый номер удаления вершин

WR("Starting delete procedure:\n");

for (int p = 1; p < N - 1; p++)

{

// Вершина, которую будем удалять

long Min = findMinVert();

removed[Min] = p;

WR("\tdeleting vertex: %d\n", Min);

//находим все смежные вершины, которые ещё не удалены

neighbohoor.clear();

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

{

// существует путь

if (removed[i] == 0 && A[Min][i] < pMax)

neighbohoor.push\_back(i);

}

//пробегаем по всем из них, смотрим, не выполняется ли правило треугольника

vector<long>::iterator I, J;

for (I = neighbohoor.begin(); I != neighbohoor.end(); I++)

{

for (J = neighbohoor.begin(); J != neighbohoor.end(); J++)

{

//не выполняется правило треугольника

if (A[\*I][\*J] > A[\*I][Min] + A[Min][\*J])

A[\*I][\*J] = A[\*I][Min] + A[Min][\*J];

}

}

WR("after deleting the vertex:\n");

FOR(ii, 1, 7) {

FOR(jj, 1, 7)

WR("%d\t", A[ii][jj]);

WR("\n");

}

WR("================================\n");

}

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

if (removed[i] == 0)

base.push\_back(i);

//Процесс обратной вставки вершин

WR("\n\n================================\n\n");

WR("starting proccess of adding vertexes\n");

for (int p = N - 2; p > 0; p--)

{

//найти вершину, которую удалили в p-той итерации

long Insert = -1;

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

if (removed[i] == p)

Insert = i;

WR("Adding the vertex: %d\n", Insert);

//Восстанавливаем статус вершины в неудаленную

removed[Insert] = 0;

vector<long>::iterator I, J;

for (I = base.begin(); I != base.end(); I++)

{

for (J = base.begin(); J != base.end(); J++)

{

if (A[Insert][\*I] > A[Insert][\*J] + A[\*J][\*I])

A[Insert][\*I] = A[Insert][\*J] + A[\*J][\*I];

}

}

base.push\_back(Insert);

WR("after adding the vertex:\n");

FOR(ii, 1, 7) {

FOR(jj, 1, 7) {

A[ii][jj] = min(A[ii][jj], A[jj][ii]);

WR("%d\t", A[ii][jj]);

}

WR("\n");

}

WR("================================\n");

}

printf("new algorithm:\n");

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

{

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

{

A[i][j] = min(A[i][j], A[j][i]);

printf("%d\t", A[i][j]);

}

printf("\n");

}

}

//Всем известный алгоритм Флойда.

void solveFloid()

{

//printf("Floid's algorithm:\n");

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

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

for (int k = 1; k <= N; k++)

A[i][j] = min(A[i][j], A[i][k] + A[k][j]);

/\*for (int i = 1; i <= N; i++) {

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

printf("%d\t", A[i][j]);

printf("\n");

}\*/

}

int main()

{

freopen("output.txt", "w", stdout);

//while (1)

{

init();

long timeStart = clock();

solve();

long timeNew = (clock() - timeStart), timeFloid;

cout << timeNew / (double)CLOCKS\_PER\_SEC << "\t\t - new Algorithm\n";

init();

timeStart = clock();

solveFloid();

timeFloid = (clock() - timeStart);

cout << timeFloid / (double)CLOCKS\_PER\_SEC << "\t\t - Floid Alogorithm\n";

cout << 100 - (timeNew \* 100) / (double)timeFloid << "\t\t - Acceleration\n";

cout << "=============================\n";

}

return 0;

}

## литература

Джимми Уэйлс, Ларри Сэнгер. (n.d.). *Википедия*. Retrieved from https://www.wikipedia.org/

Кнут, Д. (2006). *Искусство программирования, том 1. Основные алгоритмы.*

Томас Кормен, Ч. Л. (1990). *Алгоритмы: построение и анализ.*