Московский авиационный институт

(Национальный исследовательский университет)

факультет “Информационные технологии и прикладная математика”

кафедра “Математическая кибернетика”

КУРСОВАЯ РАБОТА

*по курсу* «Дискретная математика»

*2-й семестр*

|  |  |
| --- | --- |
| Тема: | Теория графов, алгебраические структуры, теория алгоритмов.  Реализация алгоритма Дейкстры для поиска кратчайших путей в нагруженных графах. |

|  |  |
| --- | --- |
| *Студент*: | Чекменев Вячеслав Алексеевич |
| *Группа*: | М8О-107Б |
| *Руководитель*: | Н.С. Алексеев |
| *Оценка*: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| *Дата*: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

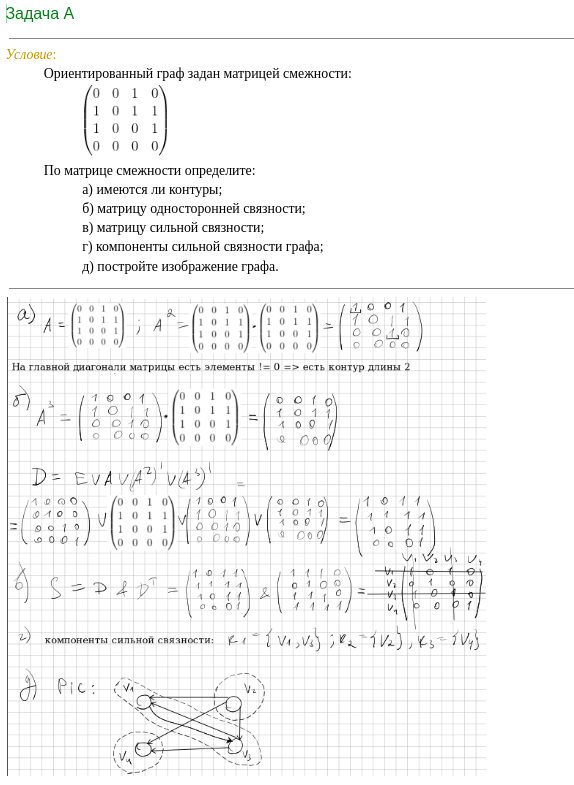
### **Введение.**

В настоящем отчете по курсовой работе приведены результаты, полученные в рамках изучения курса “Дискретная математика” во втором учебном семестре.

В части I приведены решения типовых задач по теории графов, а также представлена реализация алгоритма Дейкстры для поиска кратчайших путей в нагруженных графах. Текст программы и тестовые примеры вынесены в Приложение.

В части II приведены решения типовых задач по темам “Алгебраические структуры” и “Теория алгоритмов”.

### **Часть I. Теория графов.**



### 

### 

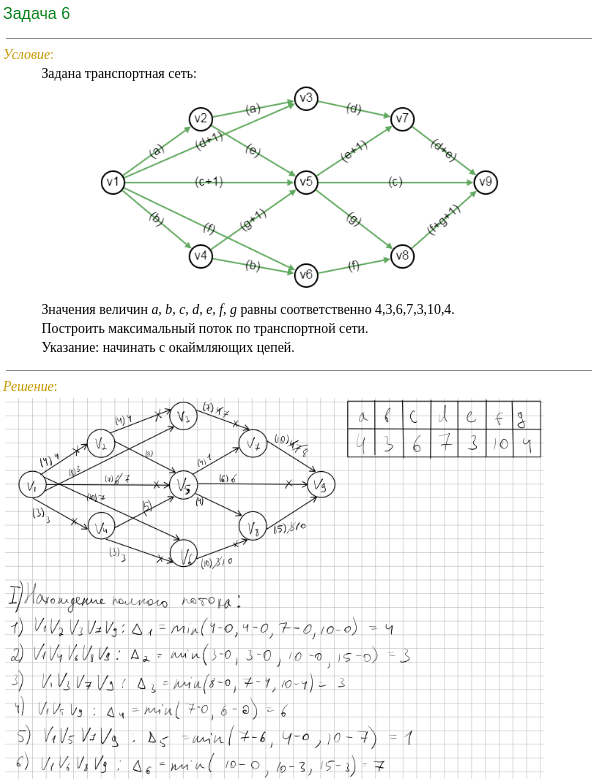
### 

### 

### 

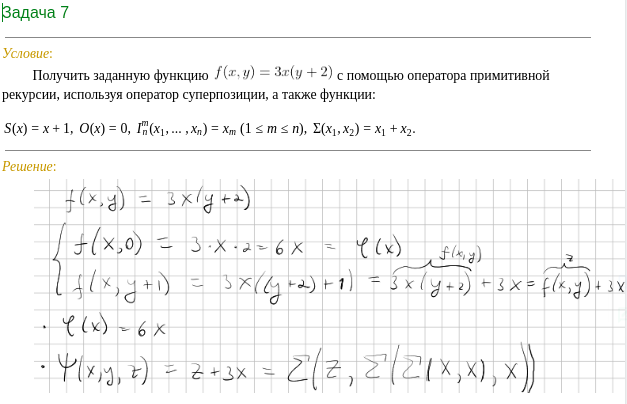
### 

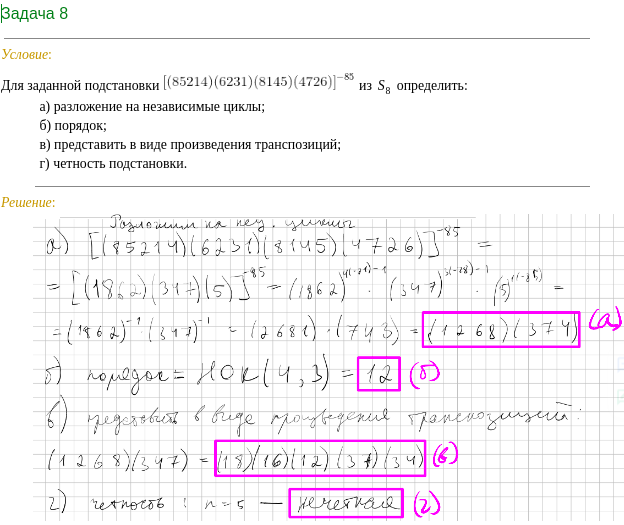
****

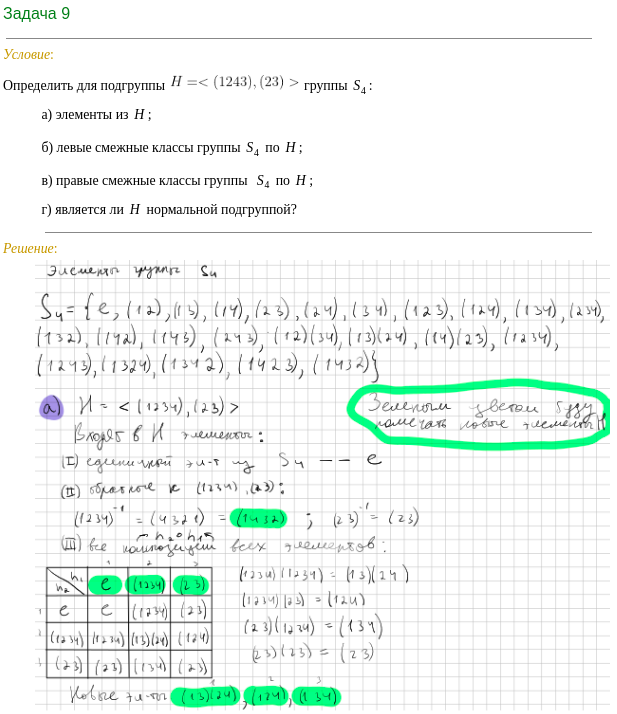
****

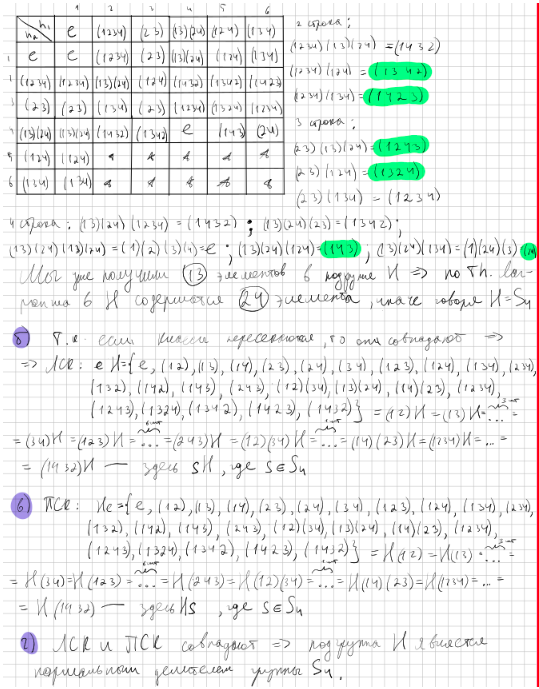
### 

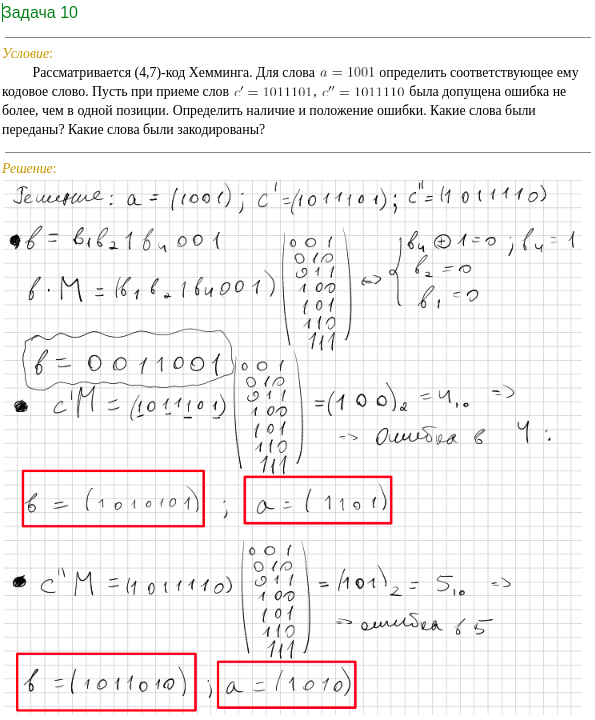
### **Часть II. Теория алгоритмов и алгебраические структуры.**

****

****

****

****

****

### **Часть III. Программная реализация алгоритмов.**

***Описание выбранного алгоритма.***

**Алгоритм Дейкстры** (англ. Dijkstra’s algorithm) — алгоритм на графах, разработанный нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании и технологиях.

**Формальное определение задачи алгоритма.**

Дан взвешенный ориентированный граф G(*V,E*)без дуг отрицательного веса [1]. Найти кратчайшие пути от некоторой вершины a графа *G* до всех остальных вершин этого графа.

**Взвешенный граф** — граф, каждому ребру которого поставлено в соответствие некое значение (вес ребра).

**Ориентированный граф** (кратко орграф) — (мульти) граф, рёбрам которого присвоено направление. Направленные рёбра именуются также дугами, а в некоторых источниках и просто рёбрами. Граф, ни одному ребру которого не присвоено направление, называется неориентированным графом или неорграфом.

**Неформальное объяснение алгоритма.**

- Каждой вершине из *V* сопоставим метку — минимальное известное расстояние от этой вершины до *a*.

- Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки.

- Работа алгоритма завершается, когда все вершины посещены.

-------------------------------------------------------------------------------------------------------------------------

[1] Для графа с отрицательными весами применяется более общий алгоритм — Алгоритм Дейкстры с потенциалами. Но здесь перед мной стоит задача изучить только стандартный алгоритм дейкстры.

-------------------------------------------------------------------------------------------------------------------------

**Инициализация**.

Метка самой вершины a полагается равной 0, метки остальных вершин — бесконечности.

Это отражает то, что расстояния от a до других вершин пока неизвестны.

Все вершины графа помечаются как непосещённые.

**Шаг алгоритма.**

Если все вершины посещены, алгоритм завершается.

В противном случае, из ещё не посещённых вершин выбирается вершина u, имеющая минимальную метку.

Мы рассматриваем всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, в которые ведут рёбра из *u*, назовём соседями этой вершины. Для каждого соседа вершины u, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки u и длины ребра, соединяющего u с этим соседом.

Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину u как посещённую и повторим шаг алгоритма.

**Описание**

В простейшей реализации для хранения чисел *d[i]* можно использовать массив чисел, а для хранения принадлежности элемента множеству *U* — массив булевых переменных.

В начале алгоритма расстояние для начальной вершины полагается равным нулю, а все остальные расстояния заполняются большим положительным числом (бо́льшим максимального возможного пути в графе). Массив флагов заполняется нулями. Затем запускается основной цикл.

На каждом шаге цикла мы ищем вершину *v* с минимальным расстоянием и флагом равным нулю. Затем мы устанавливаем в ней флаг в 1 и проверяем все соседние с ней вершины *u*. Если в них (в *u*) расстояние больше, чем сумма расстояния до текущей вершины и длины ребра, то уменьшаем его. Цикл завершается, когда флаги всех вершин становятся равны 1, либо когда у всех вершин c флагом 0 *d[i]=infty* . Последний случай возможен тогда и только тогда, когда граф *G* несвязный.

**Доказательство корректности алгоритма**

Пусть *l(v)*— длина кратчайшего пути из вершины a в вершину v. Докажем по индукции, что в момент посещения любой вершины *z d(z)=l(z).*

База. Первой посещается вершина a. В этот момент *d(a)=l(a)=0.*

Шаг. Пусть мы выбрали для посещения вершину *z != a.* Докажем, что в этот момент d*(z)=l(z).* Для начала отметим, что для любой вершины v всегда выполняется *d(v)>=l(v)* (алгоритм не может найти путь короче, чем кратчайший из всех существующих). Пусть P — кратчайший путь из a в z. Вершина z находится на P и не посещена. Поэтому множество непосещённых вершин на P непусто. Пусть y — первая непосещённая вершина на P, x — предшествующая ей (следовательно, посещённая). Поскольку путь P кратчайший, его часть, ведущая из a через x в y, тоже кратчайшая, следовательно *l(y)=l(x)+w(xy).* По предположению индукции, в момент посещения вершины x выполнялось *d(x)=l(x)*, следовательно, вершина y тогда получила метку не больше чем *d(x)+w(xy)=l(x)+w(xy)=l(y)*. Следовательно, *d(y)=l(y).* Если *z=y*, то индукционный переход доказан. Иначе, поскольку сейчас выбрана вершина z, а не y, метка z минимальна среди непосещённых, то есть *d(z)<=d(y)=l(y)<=l(z).* Комбинируя это с *d(z)>=l(z)*, имеем *d(z)=l(z)*, что и требовалось доказать.

Поскольку алгоритм заканчивает работу, когда все вершины посещены, в этот момент *d=l* для всех вершин.

***Описание реализации алгоритма.***

Так как программа будет работать с системой ГРАФОИД , необходимо в программе алгоритма реализовать получение данных и возврат данных с системой ГРАФОИД. Программа будет получать текстовый файл с матрицей смежности, преобразовывать его и возвращать обратно в систему для последующей визуализации и вывода графа кратчайших путей.

Представим граф в виде матрицы смежности *matrix[razmer+1][razmer+1]*. В массиве *distance[razmer+1]* хранятся минимальные расстояния от вершины источника, до остальных. *visited[razmer+1]* – тут хранятся посещенные вершины, добавленные в особый путь. В первом цикле алгоритма выполняется поиск путей и их длина от источника до остальных. Во втором цикле релаксируем длину, так как может найтись меньший путь.

***Оценка вычислительной сложности алгоритма.***

Сложность алгоритма Дейкстры зависит от способа нахождения вершины *v*, а также способа хранения множества непосещённых вершин и способа обновления меток. Обозначим через *n* количество вершин, а через *m* — количество рёбер в графе *G*.

В простейшем случае, когда для поиска вершины с минимальным *d[v]* просматривается всё множество вершин, а для хранения величин *d* используется массив, время работы алгоритма есть *O(n^2)*. Основной цикл выполняется порядка n раз, в каждом из них на нахождение минимума тратится порядка n операций. На циклы по соседям каждой посещаемой вершины тратится количество операций, пропорциональное количеству рёбер *m* (поскольку каждое ребро встречается в этих циклах ровно дважды и требует константное число операций). Таким образом, общее время работы алгоритма *O(n^2+m)*, но, так как *m<=n(n-1)*, оно составляет *O(n^2)*.

Для разреженных графов (то есть таких, для которых m много меньше *n²*) непосещённые вершины можно хранить в двоичной куче, а в качестве ключа использовать значения *d[i]*, тогда время удаления вершины из *U* станет *log(n)* при том, что время модификации *d[i]* возрастёт до *log n*. Так как цикл выполняется порядка n раз, а количество релаксаций (смен меток) не больше m, время работы такой реализации — *O(n\*logn + m\*logn)*.

Если для хранения непосещённых вершин использовать фибоначчиеву кучу, для которой удаление происходит в среднем за *O(log n)*, а уменьшение значения в среднем за *O(1)*, то время работы алгоритма составит *O(n\*logn + m)*.

***Описание программы и инструкции по работе с ней.***

Программа работает с системой ГРАФОИД. Для начала необходимо открыть ГРАФОИД, построить или открыть неориентированный ненагруженный граф.

Чтобы открыть граф:

* Нажать на кнопку “Граф”
* Выбрать “Открыть”
* Нажать на “обычный граф”
* Выбрать текстовый файл из имеющихся на устройстве

Чтобы создать граф:

* Нажать на кнопку “Граф”
* Выбрать “Создать”
* Нажать на “Новый обычный граф”
* В ячейке “Настройки” выбрать “Вершины”
* Расставить вершины первой доли в поле
* В ячейке “Настройки” выбрать “Вершины 2”
* Расставить вершины второй доли в поле
* В ячейке “Настройки” выбрать “Рёбра”
* Расставить их

Для запуска программы:

* В графе “Приложения” “Выбрать исполняемый файл” в формате .exe
* В графе “Приложения” “Запустить исполняемый файл”

После этого граф перестроится, появится граф, состоящий из кратчайших путей между вершинами.

***Тестовые примеры с решением и скриншотами.***

См. приложения 2-5.

***Пример прикладной задачи.***

Вариант 1. Дана сеть автомобильных дорог, соединяющих города Московской области. Некоторые дороги односторонние. Найти кратчайшие пути от города Москвы до каждого города области (если двигаться можно только по дорогам).

Вариант 2. Имеется некоторое количество авиарейсов между городами мира, для каждого известна стоимость. Стоимость перелёта из A в B может быть не равна стоимости перелёта из B в A. Найти маршрут минимальной стоимости (возможно, с пересадками) от Копенгагена до Барнаула.

**Заключение.**

В ходе выполнения курсовой работы были решены 12 задач по различным разделам курса “Дискретная математика”.

Кроме того был изучен вопрос о различных методах поиска путей и маршрутов в графах. Была написана и отлажена программа, реализующая алгоритм Дейкстры. Программа написана на языке программирования С++. Программа обеспечивает связь по установленному формату с системой ГРАФОИД, разработанной на кафедре 805, что дает возможность обеспечить графический интерфейс при ее использовании. Эта программа является основным результатом курсового проектирования.

## 

## **Список использованных источников**

* 1. *В.А. Таланов, В.Е. Алексеев.* Графы и алгоритмы. [https://www.intuit.ru](https://www.intuit.ru/) .
* 2. *Нефедов В.Н., Осипова В.А.* Курс дискретной математики. МАИ, 1992
* 3. *Нефедов В.Н.* Дискретные задачи оптимизации. <https://goo.gl/faUEEU>
* 4. Алгоритм Дейкстры. Википедия https://ru.wikipedia.org/wiki/Алгоритм\_Дейкстры

### **Приложение 1**

* **Текст программы**

*#include <cstdlib>*

*#include <iostream>*

*#include <fstream>*

*#include <climits>*

*#include <string.h>*

*#include <ctype.h>*

*#define STR 10*

*using namespace std;*

*char \*itoa(int value, char\* result, int base) // функция перевода int в char[]*

*{*

*// check that the base if valid*

*if (base < 2 || base > 36) { \*result = '\0'; return result; }*

*char \*ptr = result, \*ptr1 = result, tmp\_char;*

*int tmp\_value;*

*do {*

*tmp\_value = value;*

*value /= base;*

*\*ptr++ = "zyxwvutsrqponmlkjihgfedcba9876543210123456789abcdefghijklmnopqrstuvwxyz" [35 + (tmp\_value - value \* base)];*

*} while ( value );*

*// Apply negative sign*

*if (tmp\_value < 0) \*ptr++ = '-';*

*\*ptr-- = '\0';*

*while(ptr1 < ptr) {*

*tmp\_char = \*ptr;*

*\*ptr--= \*ptr1;*

*\*ptr1++ = tmp\_char;*

*}*

*return result;*

*}*

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

*{*

*//Loading matrix*

*ifstream in(argv[1]);*

*int razmer=0,i,j;*

*in >> razmer;*

*razmer--;*

*int matrix[razmer+1][razmer+1];*

*for(int i=0;i<=razmer;i++) {*

*for(int j=0;j<=razmer;j++) {*

*in >> matrix[i][j];*

*}*

*}*

*in.close();*

*//////// Algorithm*

*for (int l = 1; l <= razmer + 1; l++) {*

*int distance[razmer + 1]; // массив минимальных расстояний*

*int count, index, i, u, st = l - 1, m = st + 1; // st источник*

*bool visited[razmer + 1]; // посещенные вершины, добавленные в особый путь*

*int matr[razmer + 1];*

*for (i = 0; i < razmer + 1; i++) {*

*distance[i] = INT\_MAX; // сохраняем бесконечность в расстояния*

*visited[i] = false; // делаем вершины непосешенными*

*}*

*distance[st] = 0; // путь из v1 -> v1 = 0*

*for (count = 0; count < razmer; count++) {*

*int min = INT\_MAX;*

*for (i = 0; i < razmer + 1; i++) { // ищем минимальные пути в каждую вершину*

*if (!visited[i] && distance[i] <= min) {*

*min = distance[i];*

*index = i;*

*}*

*}*

*u = index; // индекс вершины, добавленной в особый путь*

*visited[u] = true;*

*for (i = 0; i < razmer + 1; i++) { // ищем меньшую длину пути, если есть (через другие вершины)*

*if (!visited[i] && matrix[u][i] && (distance[u] != INT\_MAX) && (distance[u] + matrix[u][i] < distance[i])) {*

*distance[i] = distance[u] + matrix[u][i];*

*}*

*}*

*}*

*for (int k = 0; k < razmer + 1; k++) {*

*if (distance[k] != INT\_MAX) {*

*matr[k] = distance[k]; // записываем в массив кратчайшие пути*

*} else {*

*matr[k] = 0; // если пути нет*

*}*

*}*

*for (int k = 0; k < razmer + 1; k++) {*

*matrix[l - 1][k] = matr[k]; // конкатенируем матрицы, получаем матрицу смежности*

*}*

*}*

*//Saving new matrix*

*fstream out;*

*out.open(argv[1]);*

*out.clear();*

*char buffer [33];*

*itoa(razmer+1,buffer,10);*

*out << buffer << "\n";*

*for(i=0; i<=razmer;i++) {*

*for(j=0;j<=razmer;j++) {*

*out << matrix[i][j];*

*if(j!=razmer) {*

*out << " ";*

*}*

*}*

*if(i!=razmer) {*

*out << "\n";*

*}*

*}*

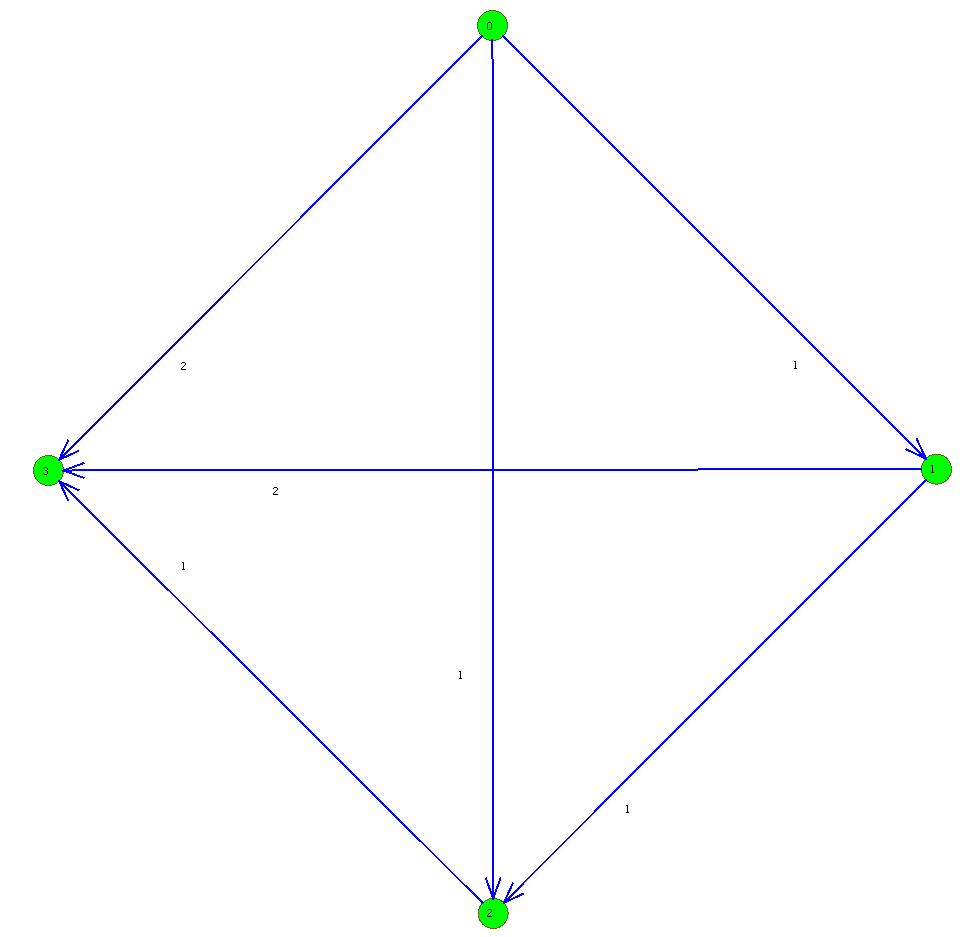
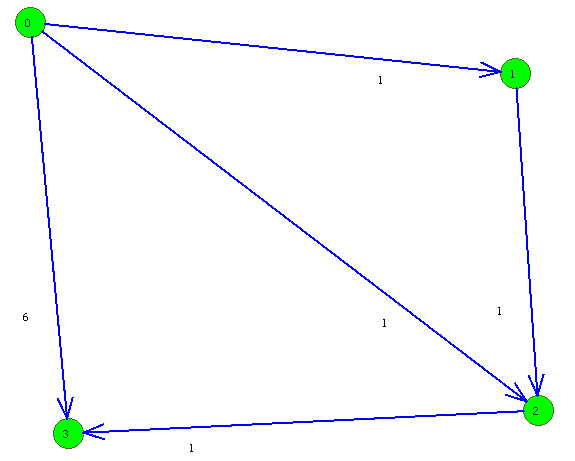
*out.close();*

*return 0;*

*}*

### **Приложение 2**

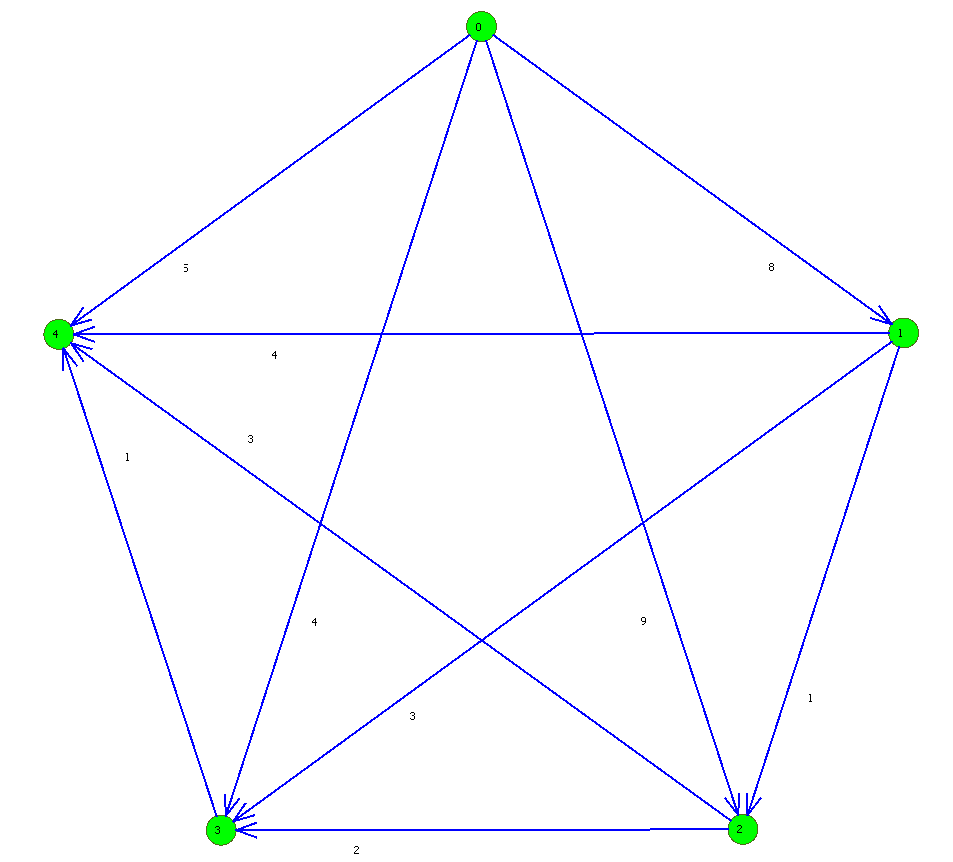
**Граф: Результат работы:**

****

### **Приложение 3**

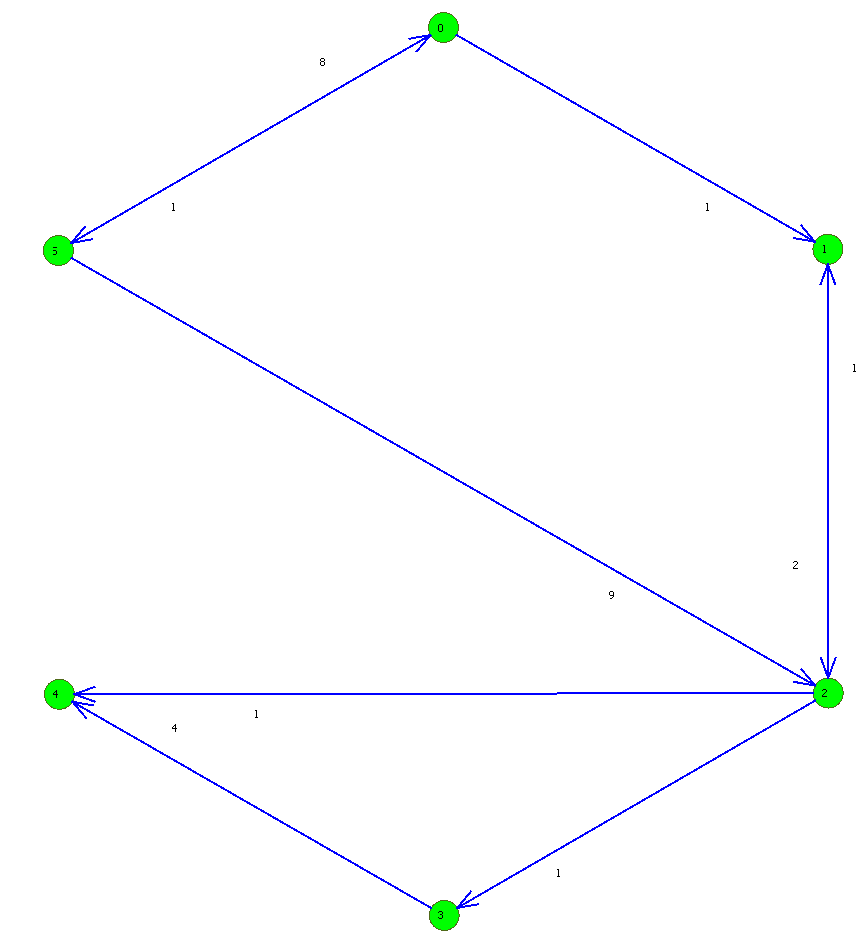
**Граф:**

**Результат работы:**

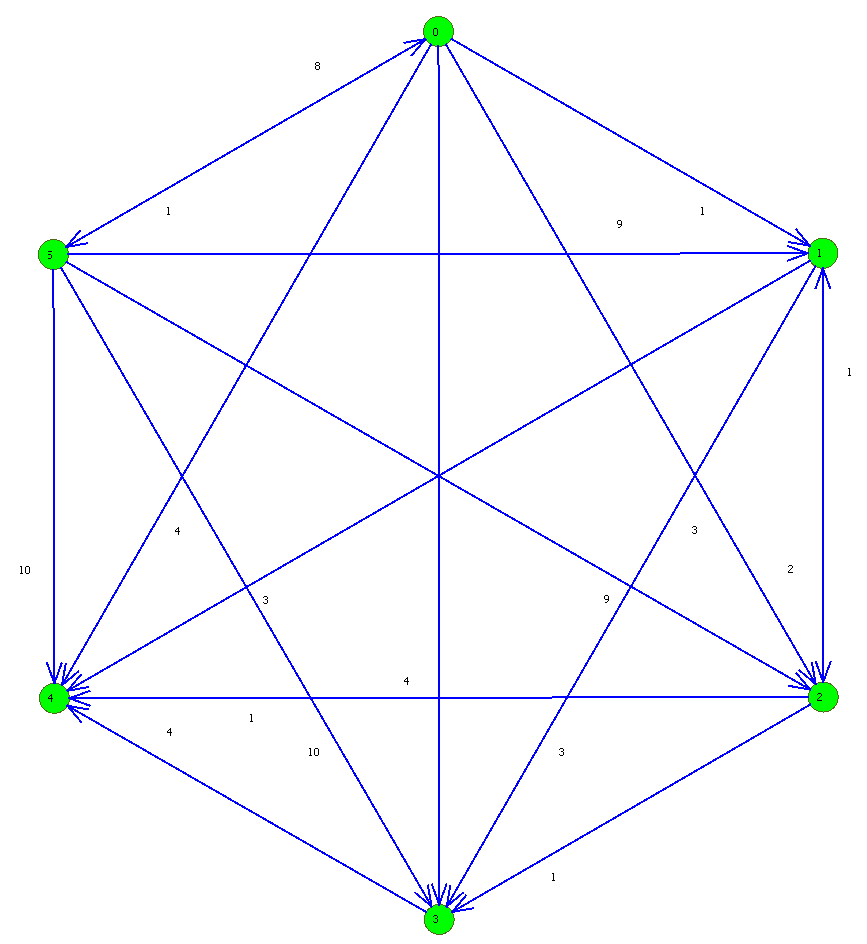
****

### **Приложение 4**

**Граф:**



**Результат работы:**

****