1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский политехнический университет Петра Великого
3. —
4. Институт компьютерных наук и технологий
5. **Кафедра «Информационная безопасность компьютерных систем»**

**ОТЧЕТ**

**по лабораторной работе № 1**

1. по дисциплине «Дискретная Математика»
2. Выполнил
3. студент гр. 23508/4 Е.Г. Проценко
4. Проверила
5. ассистент Д.С. Лаврова
6. Санкт-Петербург
7. 2016

# Формулировка задания (Вариант 7)

Цель работы – изучение основ теории графов, базовых понятий и определений компьютерных способов представления графов и операций над ними.

1. Написать на C++ класс, описывающий граф/орграф. Класс должен поддерживать следующую функциональность:

* Определение числа вершин;
* Определение числа ребер (дуг);
* Определение степени произвольной вершины (для орграфа – полустепеней захода);
* Определение степенной последовательности графа;
* Определение матрицы смежности;
* Определение матрицы инцидентности;
* Определение списка смежности;
* Добавления/удаление вершин в граф;
* Добавление/удаление ребра в граф;
* Определение дополнения графа;
* Подразбиение ребра;
* Стягивание графа;
* Отождествление вершин;
* Дублирование вершин;
* Размножение вершин;
* Объединение (дизъюнктивное) графов;
* Соединение графов;
* Произведение графов;
* Ввод/вывод графов в текстовый файл в виде списка смежности в следующем формате: {1 <смежные вершины через пробел>…}{2 <смежные вершины через пробел>}… Например: {1 2}{2 1 4}{3}{4 2}.

1. Главная программа должна демонстрировать возможности разработанного класса. А именно:

* Она должна позволять задавать пользователю граф в любом из трех видов (список смежности, матрица смежности или матрица инцидентности) и получать на выходе любое другое представление;
* По запросу пользователя выводить характеристики графа (число вершин, число ребер, степенную последовательность, степень выбранной вершины);
* Позволять пользователю выполнять реализованные в классе операции;
* Визуализация графа – по желанию.

1. Получить у преподавателя вариант задания и представить результаты его выполнения в отчете.

Граф задан матрицей смежности, граф – матрицей инцидентности, орграф – матрицей инцидентности. В матрицах инцидентности по строкам располагаются номера вершин. Вывод результатов понимается как вывод в файл и/или на экран (в отчете также представить результат). Если выполнить операцию невозможно – в отчете необходимо обоснование.

*Задание 1.* Для заданных графов и орграфов вывести (определить):

а) Число вершин и число ребер (дуг);

б) Списки смежности;

в) Степенные последовательности (для орграфа – полустепени захода и исхода каждой вершины);

г) матрицу инцидентности , матрицы смежности и .

*Задание 2.*

а) добавить в граф новые вершины , и удалить из него вершину . Вывести результат;

б) добавить в полученный на предыдущем шаге граф ребра , , , , и удалить ребра , . Вывести результат в виде матрицы инцидентности;

в) построить дополнение полученного на шаге б) графа. Пусть это граф (вывести его в виде матрицы смежности);

г) добавить в орграф вершины , , и удалить . Добавить дуги , , , удалить дугу . Вывести результат;

д) выполнить операции предыдущего пункта в обратном порядке (сначала дуги, потом вершины). Вывести результат и сравнить с предыдущим.

*Задание 3.*

Получить граф из имеющихся графов , и по заданной формуле (списку операций). Вывести результат.

, ,.

Списки вершин, ребер и дуг для заданий:

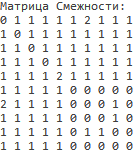
*vi*: {11, 12, 8, 8, 9, 10, 3}; *ei*: {(9,11), (9,10), (4,12), (10, 12), (5,11), (5,5), (7,1)}; *fi*: {(1,4), (2,7), (1,9), (3,4)}.

Получить граф: в *G*4отождествить вершины 1 и 10, затем найти объединение с

# Результаты работы

## Задание 1

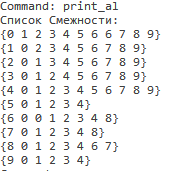
1. Задание 1 для графа .



1. Число вершин и ребер



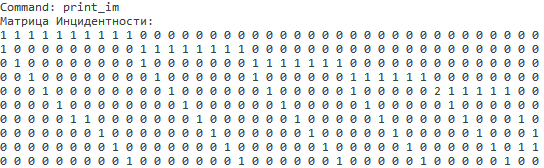
1. Список смежности



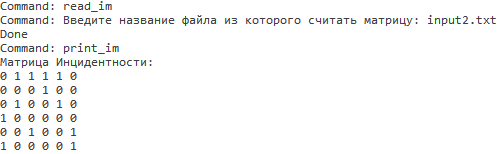
1. Степенная последовательность



1. Матрица инцидентности



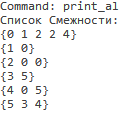
1. Задание 1 для графа .



1. Число вершин и ребер



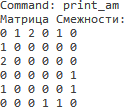
б) Список смежности



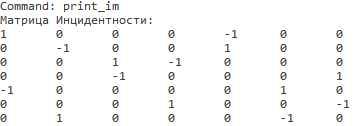
в) Степенная последовательность



г) Матрица смежности



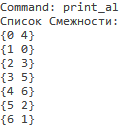
1. Задание 1 для графа .



1. Число вершин и ребер



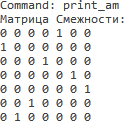
б) Список смежности



в) Степенная последовательность

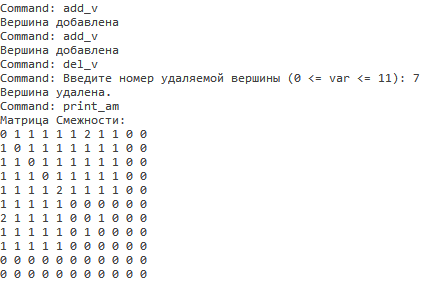


г) Матрица смежности



## Задание 2

1. добавить в граф новые вершины , и удалить из него вершину . Вывести результат;



Не забываем, что нумерация начинается с нуля.

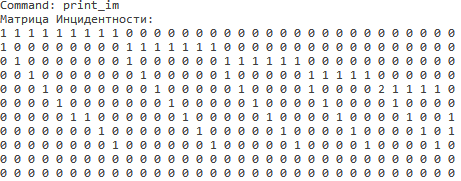
После того как мы удалили вершину 8, вершины с большим индексом сдвинулись вниз.

1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12 – остались

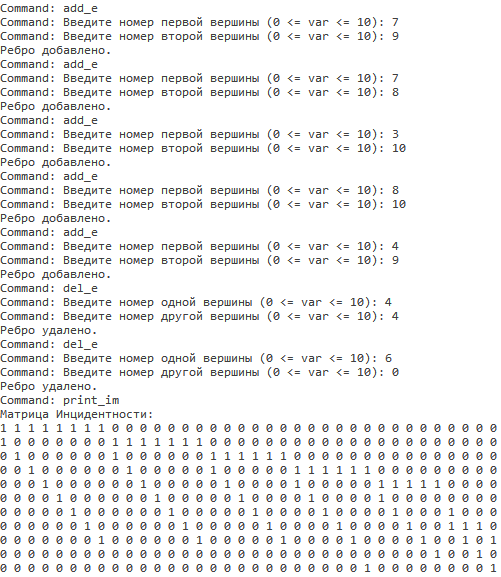
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 – их индексы

1. добавить в полученный на предыдущем шаге граф ребра , , , , и удалить ребра , . Вывести результат в виде матрицы инцидентности;

До преобразований:

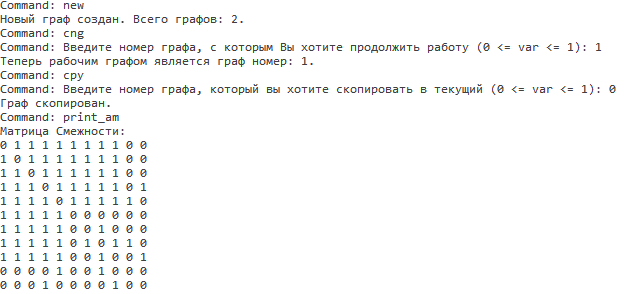


После преобразований:

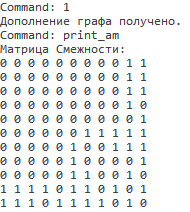


1. построить дополнение полученного на шаге b) графа. Пусть это граф (вывести его в виде матрицы смежности);

До преобразований:

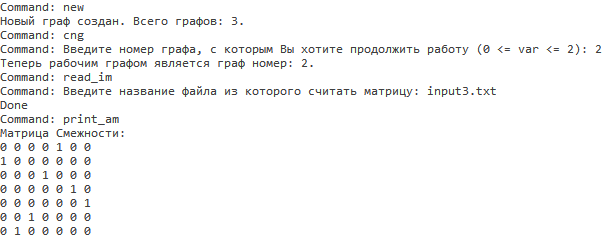


После преобразований:

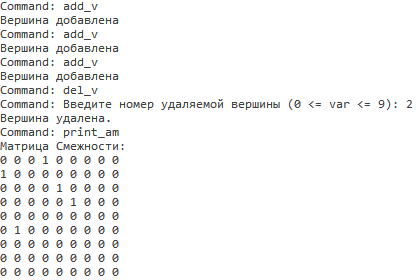


1. Добавить в орграф вершины , , и удалить . Добавить дуги , , , удалить дугу . Вывести результат;

Считали граф:



Добавили, удалили вершины :

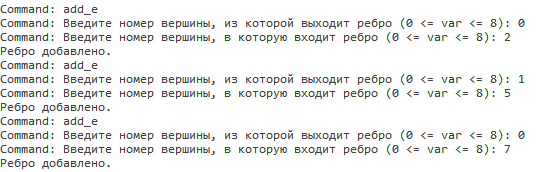


Вершины:

1, 2, 4, 5, 6, 7, 8, 9, 10 – остались

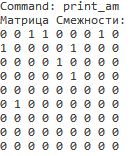
0, 1, 2, 3, 4, 5, 6, 7, 8 – их индексы

Добавили, удалили дуги :

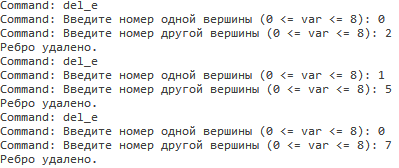


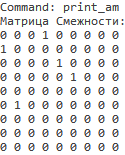
Удалить ребро {3,4} невозможно, т.к. вершина 3 была удалена ранее.

Результат:

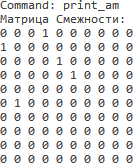


1. выполнить операции предыдущего пункта в обратном порядке (сначала дуги, потом вершины). Вывести результат и сравнить с предыдущим.





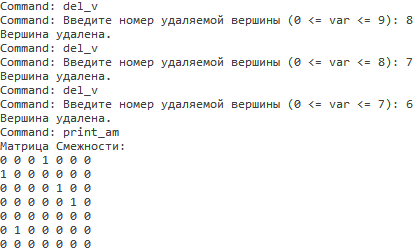
После добавление вершины, она встанет на последнюю позицию:



Вершины:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10 – остались

0, 1, 9, 2, 3, 4, 5, 6, 7, 8 – их индексы

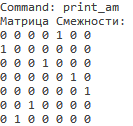
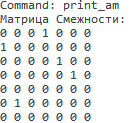


Вершины:

1, 2, 3, 4, 5, 6, 7 – остались

0, 1, 9, 2, 3, 4, 5 – их индексы

До и после преобразований:

🡪 

Вершина 3 переместилась вниз из-за удаления/добавлении.

При удалении вместе с вершиной удалилась дуга {3,4}.

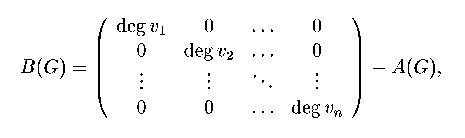
# контрольные вопросы

1. Чему равна сумма степеней всех вершин графа?

По лемме о рукопожатиях {\displaystyle \sum_{v\in V} \deg(v) = 2|E|}

1. Докажите, что алгебраические дополнения всех элементов матрицы Кирхгофа графа равны между собой.

Определим матрицу Кирхгофа *B* = *B*(*G*), полагая



где *A*(*G*) — матрица смежности графа *G*.

Обозначим столбец (1, 1, ..., 1)T длины *n*, состо­ящий из единиц, через 1.

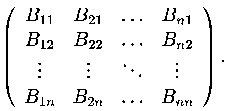
Для матрицы Кирхгофа *B*(*G*) = (*βij*)*n*x*n* выполняется

Σ*j*=1..*n* *βij* = 0 (*i* = 1, 2, ..., *n*), т.е. *B* \* 1 = 0,

Σ*i*=1..*n* *βij* = 0 (*j* = 1, 2, ..., *n*), т.е. 1T \* *B* = 0.

Отсюда следует, что det *B* = 0 и rank *B* ≤ *n* - 1.

Если rank *В* < *n* - 1, то все алгебраические дополнения элементов матрицы *В* равны 0. Пусть rank *В* = *n* - 1 и *C* — присоединённая к В матрица, составленная из алгебраических дополнений *Вij* элементов *βij*, т.е.



В силу свойств *C* получаем

*BC* = *CB* = (det *B*)*E* = 0.

Так как *ВC* = 0, любой столбец *X* матрицы *C* удовлетворяет системе *ВХ* = 0. Эта система линейных уравнений имеет ранг *n* - 1. Так как *B* \* 1 = 0, этой системе удовлетворяет столбец 1. Следовательно, столбцы матрицы *C* пропорциональны столбцу 1, откуда следует

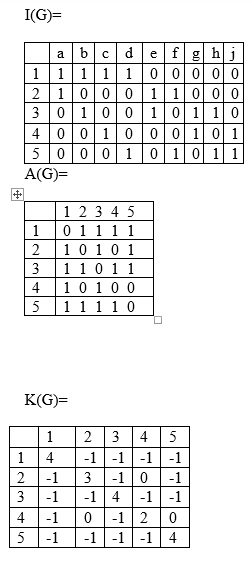
*Bi*1 = *Bi*2 = ... = *Bin* (*i* = 1, 2, ..., *n*).

Аналогично получаем

*B*1*j* = *B*2*j* = ... = *Bnj* (*j* = 1, 2, ..., *n*).

Следовательно, все элементы матрицы *C* одинаковы. Ч.т.д

1. Пусть *G* — граф, множество вершин которого совпадает с отрезком натурального ряда {1,2,...5}, а множество ребер определяется следующим условием: несовпадающие вершины *vi*, и *vj* смежны тогда, когда числа *i* и *j*взаимно просты. Какой вид имеют матрица смежности, матрица инцидентности и матрица Кирхгофа?



4. Задан неориентированный граф *G*. В графе удаляются вершина и два ребра. Существенна ли последовательность выполнения операций? Нет.

5.Верно ли равенство: . Да

6.Графы  и *G* являются подграфами .Выполняется ли для них соотношение:

? Да

# Вывод

В данной лабораторной работе я познакомился с языком C++ и классами. Был написан класс, который работает с графами и орграфами. Методами класса являются все основные операции над графами, начиная от добавления/удаления вершин, заканчивая перемножением графов.  
Во время написания программы я разобрался с теоретическим материалом и применил на практике. Все вычисления проверялись вручную. Теперь я знаю, чем отличается каждая операция от других.

Факты о работе:

* На работу было потрачено суммарно около 25 часов
* Было выпито 5 банок RedBull
* Было потрачено 3 бессонных ночи
* Это самая большая программа на C++
* Это самый большой класс, написанный мной
* По времени больше заняла только курсовая работа за 2 семестр “Покер”

# Приложение

Листинг написанной программы:

main.cpp

#include "Graph.hpp"

#include <iostream>

#include <fstream>

#include <clocale>

#include <string>

int main()

{

setlocale(LC\_ALL, "Russian");

int total = 1;

int current = 0;

Graph \* Graph\_arr = (Graph \*)malloc(sizeof(Graph));

Graph \* \_Graph = Graph\_arr;

\_Graph->help();

\_Graph->init();

std::string str;

//char str[256];

while (true)

{

std::cout << "Command: ";

std::cin >> str;

if (str.compare("help") == 0) \_Graph->help();

else if (str.compare("exit") == 0) break;

else if (str.compare("new") == 0)

{

Graph\_arr = (Graph \*)realloc((void \*)Graph\_arr, (++total) \* sizeof(Graph));

\_Graph = &Graph\_arr[current];

Graph\_arr[total - 1].init();

std::cout << "Новый граф создан. Всего графов: " << total << "." << std::endl;

}

else if (str.compare("change") == 0 || str.compare("cng") == 0)

{

int i;

std::cout << "Command: Введите номер графа, с которым Вы хотите продолжить работу (0 <= var <= " << total - 1 << "): ";

std::cin >> i;

if (i >= 0 && i <= total - 1)

{

\_Graph = &Graph\_arr[i];

current = i;

std::cout << "Теперь рабочим графом является граф номер: " << i << "." << std::endl;

}

else std::cout << "Неверные входные данные." << std::endl;

}

else if (str.compare("copy") == 0 || str.compare("cpy") == 0)

{

int i;

std::cout << "Command: Введите номер графа, который вы хотите скопировать в текущий (0 <= var <= " << total - 1 << "): ";

std::cin >> i;

if (i >= 0 && i <= total - 1)

{

\_Graph->copy\_from(&Graph\_arr[i]);

std::cout << "Граф скопирован." << std::endl;

}

else std::cout << "Неверные входные данные." << std::endl;

}

else if (str.compare("read\_am") == 0)

{

std::cout << "Command: Введите название файла из которого считать матрицу: ";

std::string str;

std::cin >> str;

str.c\_str();

std::ifstream fin(str.c\_str());

if (!fin.is\_open()){

std::cout << "Такого файла не существует" << std::endl;

continue;

}

\_Graph->read\_adjacency\_matrix(&str);

std::cout << "Done" << std::endl;

}

else if (str.compare("read\_im") == 0)

{

std::cout << "Command: Введите название файла из которого считать матрицу: ";

std::string str;

std::cin >> str;

str.c\_str();

std::ifstream fin(str.c\_str());

if (!fin.is\_open()){

std::cout << "Такого файла не существует" << std::endl;

continue;

}

\_Graph->read\_incidence\_matrix(&str);

std::cout << "Done" << std::endl;

}

/\*----------------------------------\*/

/\*----------------------------------\*/

/\*----------------------------------\*/

else if (str.compare("get\_v") == 0) std::cout << "Число вершин в графе: " << \_Graph->get\_vertices() << std::endl;

else if (str.compare("get\_e") == 0) std::cout << "Число ребер в графе: " << \_Graph->get\_edges() << std::endl;

else if (str.compare("get\_deg") == 0)

{

int i;

std::cout << "Command: Введите номер вершины: ";

std::cin >> i;

if (i >= 0 && i < \_Graph->get\_vertices())

{

int result[2];

\_Graph->get\_v\_degree(i, result);

if (\_Graph->is\_oriented\_graph())

{

std::cout << "Полуcтепень исхода вершины " << i << ": " << result[0] << "." << std::endl;

std::cout << "Полуcтепень захода вершины " << i << ": " << result[1] << "." << std::endl;

}

else

{

std::cout << "Степень вершины " << i << ": " << result[0] << "."<< std::endl;

}

}

else std::cout << "Wrong v number" << std::endl;

}

else if (str.compare("get\_deg\_sqnce") == 0)

{

int \* sqnce = \_Graph->get\_degree\_sequence();

if (\_Graph->is\_oriented\_graph())

{

std::cout << "Полустепени входа и исхода для каждой вершины: " << std::endl;

int ver = \_Graph->get\_vertices();

for (int i = 0; i < ver; i++) std::cout << "{" << i << ":" << sqnce[i] << "," << sqnce[i + ver] << "}";

std::cout << std::endl;

continue;

}

std::cout << "Степенная последовательность графа: " << std::endl;

for (int i = 0; i < \_Graph->get\_vertices(); i++) std::cout << "{" << sqnce[i + \_Graph->get\_vertices()] << ":" << sqnce[i] << "}";

std::cout << std::endl;

}

/\*----------------------------------\*/

/\*----------------------------------\*/

/\*----------------------------------\*/

else if (str.compare("print\_am") == 0)

{

std::cout << "Матрица Смежности:" << std::endl;

\_Graph->print\_adjacency\_matrix();

}

else if (str.compare("print\_im") == 0)

{

std::cout << "Матрица Инцидентности:" << std::endl;

\_Graph->print\_incidence\_matrix();

}

else if (str.compare("print\_al") == 0)

{

std::cout << "Список Смежности:" << std::endl;

\_Graph->print\_adjacency\_list();

}

else if (str.compare("print\_out") == 0)

{

\_Graph->print\_out();

std::cout << "Список смежнотей теперь в output.txt." << std::endl;

}

/\*----------------------------------\*/

/\*----------------------------------\*/

/\*----------------------------------\*/

else if (str.compare("add\_v") == 0)

{

\_Graph->add\_vortex();

std::cout << "Вершина добавлена" << std::endl;

}

else if (str.compare("del\_v") == 0)

{

int vertices = \_Graph->get\_vertices();

if (vertices > 0) std::cout << "Command: Введите номер удаляемой вершины (0 <= var <= " << vertices - 1 << "): ";

else

{

std::cout << "Граф пуст. В нем нечего удалять" << std::endl;

continue;

}

int v;

std::cin >> v;

if (v >= 0 && v <= vertices - 1)

{

\_Graph->delete\_vortex(v);

std::cout << "Вершина удалена." << std::endl;

}

else std::cout << "Неверные входные данные" << std::endl;

}

else if (str.compare("add\_e") == 0)

{

int vertices = \_Graph->get\_vertices();

if (vertices == 0)

{

std::cout << "Граф пуст. Невозможно добавить ребро." << std::endl;

continue;

}

int out, in;

if (\_Graph->is\_oriented\_graph() == true) std::cout << "Command: Введите номер вершины, из которой выходит ребро (0 <= var <= " << vertices - 1 << "): ";

else std::cout << "Command: Введите номер первой вершины (0 <= var <= " << vertices - 1 << "): ";

std::cin >> out;

if (out < 0 || out >= vertices)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

if (\_Graph->is\_oriented\_graph() == true) std::cout << "Command: Введите номер вершины, в которую входит ребро (0 <= var <= " << vertices - 1 << "): ";

else std::cout << "Command: Введите номер второй вершины (0 <= var <= " << vertices - 1 << "): ";

std::cin >> in;

if (in < 0 || in >= vertices)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

\_Graph->add\_edge(out, in);

std::cout << "Ребро добавлено." << std::endl;

}

else if (str.compare("del\_e") == 0)

{

/\*int e;

int edges = \_Graph->get\_edges();

if (edges == 0)

{

std::cout << "В графе нет ребер. Нечего удалять." << std::endl;

continue;

}

else std::cout << "Command: Введите номер удаляемого ребра (0 <= var <= " << edges - 1 << "): ";

std::cin >> e;

if (e < 0 || e >= edges)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

\_Graph->delete\_edge(e);

std::cout << "Ребро удалено." << std::endl;\*/

int v1, v2;

int edges = \_Graph->get\_edges();

int ver = \_Graph->get\_vertices();

if (edges == 0)

{

std::cout << "В графе нет ребер. Нечего удалять." << std::endl;

continue;

}

std::cout << "Command: Введите номер одной вершины (0 <= var <= " << ver - 1 << "): ";

std::cin >> v1;

if (v1 < 0 || v1 >= ver)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

std::cout << "Command: Введите номер другой вершины (0 <= var <= " << ver - 1 << "): ";

std::cin >> v2;

if (v2 < 0 || v2 >= ver)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

bool err = \_Graph->delete\_edge\_2(v1, v2);

if (err == false) std::cout << "Ребро удалено." << std::endl;

else std::cout << "Такое ребро отсутствует" << std::endl;

}

/\*----------------------------------\*/

/\*----------------------------------\*/

/\*----------------------------------\*/

else if (str.compare("1") == 0)

{

if (!\_Graph->is\_oriented\_graph()) \_Graph->complement\_graph();

else

{

std::cout << "Недопустимая операция для ориентированного графа" << std::endl;

continue;

}

std::cout << "Дополнение графа получено." << std::endl;

}

else if (str.compare("2") == 0)

{

int e;

int edges = \_Graph->get\_edges();

if (edges == 0)

{

std::cout << "В графе нет ребер. Нечего подразбивать." << std::endl;

continue;

}

else std::cout << "Command: Введите номер подразбиваемого ребра (0 <= var <= " << edges - 1 << "): ";

std::cin >> e;

if (e < 0 || e >= edges)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

\_Graph->edge\_subdivision(e);

std::cout << "Подразбиение выполнено." << std::endl;

}

else if (str.compare("3") == 0)

{

int vertices = \_Graph->get\_vertices();

int count = 0;

int \* v\_index = NULL;

std::cout << "Последовательно введите неповторяющиеся вершины, которые нужно стянуть" << std::endl;

std::cout << "Введите -1, чтобы закончить ввод вершин" << std::endl;

std::cout << "Вершины (0 <= var <= " << vertices - 1 << "): " << std::endl;

int i;

bool error = false;

while (true)

{

std::cin >> i;

if (i == -1) break;

if (!(i >= 0 && i <= vertices - 1))

{

std::cout << "Ошибочные данные. Такой вершины не сущестует." << std::endl;

error = true;

break;

}

if (error) break;

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

{

if (v\_index[j] == i)

{

std::cout << "Ошибочные данные. Вершина повторяется." << std::endl;

error = true;

break;

}

}

if (error) break;

v\_index = (int \*)realloc((void \*)v\_index, (++count) \* sizeof(int));

v\_index[count - 1] = i;

}

if (error) continue;

if (count < 2){

std::cout << "Ошибочные данные. Мало вершин. Нужны минимум две." << std::endl;

continue;

}

\_Graph->graph\_contraction(v\_index, count);

std::cout << "Стягивание графа выполнено." << std::endl;

}

else if (str.compare("4") == 0)

{

int x, y;

int vertices = \_Graph->get\_vertices();

if (vertices <= 1)

{

std::cout << "В графе должно быть минимум 2 вершины для даннйо операции." << std::endl;

continue;

}

std::cout << "Command: Введите номер одной вершины (0 <= var <= " << vertices - 1 << "): ";

std::cin >> x;

if (x < 0 || x >= vertices)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

std::cout << "Command: Введите номер другой вершины (0 <= var <= " << vertices - 1 << "): ";

std::cin >> y;

if (y < 0 || y >= vertices || x == y)

{

if (x == y) std::cout << "Ошибочные входные данные. Вершины должны быть разными. Попробуйте снова." << std::endl;

else std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

\_Graph->vertex\_involving\_2(x, y);

std::cout << "Отождествление выполнено." << std::endl;

}

else if (str.compare("5") == 0)

{

int vertices = \_Graph->get\_vertices();

if (vertices > 0) std::cout << "Command: Введите номер удаляемой вершины (0 <= var <= " << vertices - 1 << "): ";

else

{

std::cout << "Граф пуст. В нем нечего дублировать." << std::endl;

continue;

}

int v;

std::cin >> v;

if (v >= 0 && v <= vertices - 1)

{

\_Graph->dupliacate\_vertices(v);

std::cout << "Дублирование выполнено." << std::endl;

}

else std::cout << "Неверные входные данные." << std::endl;

}

else if (str.compare("6") == 0)

{

int vertices = \_Graph->get\_vertices();

if (vertices > 0) std::cout << "Command: Введите номер удаляемой вершины (0 <= var <= " << vertices - 1 << "): ";

else

{

std::cout << "Граф пуст. В нем нечего размножать." << std::endl;

continue;

}

int v;

std::cin >> v;

if (v >= 0 && v <= vertices - 1)

{

\_Graph->vertices\_reproduction(v);

std::cout << "Размножение вершины выполнено." << std::endl;

}

else std::cout << "Неверные входные данные." << std::endl;

}

/\*----------------------------------\*/

/\*----------------------------------\*/

/\*----------------------------------\*/

else if (str.compare("7") == 0)

{

if (total < 3)

{

std::cout << "Для данной операции нужны 3 рабочих графа. Два для хранения исходных графов и один для сохранения туда результата операции." << std::endl;

std::cout << "На данный момент вы имеете всего " << total << " графов." << std::endl;

continue;

}

int g1, g2;

std::cout << "Не повторяйтесь и не вводите номер текущего графа, т.к. туда будет записан результат опреации" << std::endl;

std::cout << "Введите номер одного графа (0 <= var <= " << total - 1 << "): ";

std::cin >> g1;

if (g1 < 0 || g1 >= total || g1 == current)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

std::cout << "Command: Введите номер другого графа (0 <= var <= " << total - 1 << "): ";

std::cin >> g2;

if (g2 < 0 || g2 >= total || g2 == current || g2 == g1)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

if (Graph\_arr[g1].is\_oriented\_graph() != Graph\_arr[g2].is\_oriented\_graph())

{

std::cout << "Один граф ориентирован, другой нет. Так нельзя." << std::endl;

continue;

}

\_Graph->graphs\_union(&Graph\_arr[g1], &Graph\_arr[g2]);

std::cout << "Объединение выполнено." << std::endl;

}

else if (str.compare("8") == 0)

{

if (total < 3)

{

std::cout << "Для данной операции нужны 3 рабочих графа. Два для хранения исходных графов и один для сохранения туда результата операции." << std::endl;

std::cout << "На данный момент вы имеете всего " << total << " графов." << std::endl;

continue;

}

int g1, g2;

std::cout << "Не повторяйтесь и не вводите номер текущего графа, т.к. туда будет записан результат опреации" << std::endl;

std::cout << "Command: Введите номер одного графа (0 <= var <= " << total - 1 << "): ";

std::cin >> g1;

if (g1 < 0 || g1 >= total || g1 == current)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

std::cout << "Command: Введите номер другого графа (0 <= var <= " << total - 1 << "): ";

std::cin >> g2;

if (g2 < 0 || g2 >= total || g2 == current || g2 == g1)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

if (Graph\_arr[g1].is\_oriented\_graph() != Graph\_arr[g2].is\_oriented\_graph())

{

std::cout << "Один граф ориентирован, другой нет. Так нельзя." << std::endl;

continue;

}

\_Graph->graphs\_connection(&Graph\_arr[g1], &Graph\_arr[g2]);

std::cout << "Соединение выполнено." << std::endl;

}

else if (str.compare("9") == 0)

{

if (total < 3)

{

std::cout << "Для данной операции нужны 3 рабочих графа. Два для хранения исходных графов и один для сохранения туда результата операции." << std::endl;

std::cout << "На данный момент вы имеете всего " << total << " графов." << std::endl;

continue;

}

int g1, g2;

std::cout << "Не повторяйтесь и не вводите номер текущего графа, т.к. туда будет записан результат опреации" << std::endl;

std::cout << "Command: Введите номер одного графа (0 <= var <= " << total - 1 << "): ";

std::cin >> g1;

if (g1 < 0 || g1 >= total || g1 == current)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

std::cout << "Command: Введите номер другого графа (0 <= var <= " << total - 1 << "): ";

std::cin >> g2;

if (g2 < 0 || g2 >= total || g2 == current || g2 == g1)

{

std::cout << "Ошибочные входные данные. Попробуйте снова." << std::endl;

continue;

}

if (Graph\_arr[g1].is\_oriented\_graph() != Graph\_arr[g2].is\_oriented\_graph())

{

std::cout << "Один граф ориентирован, другой нет. Так нельзя." << std::endl;

continue;

}

\_Graph->product\_of\_graphs(&Graph\_arr[g1], &Graph\_arr[g2]);

std::cout << "Произведение графов выполнено." << std::endl;

}

else std::cout << "Wrong Command!" << std::endl;

}

}

Graph.h

#ifndef GRAPH\_H

#define GRAPH\_H

#include <string>

//#include <vector>

class Graph

{

private:

int \*\* adjacency\_matrix; // матрица смежности

int \*\* incidence\_matrix; // матрица инцидентности // строки = номера ребра // столбец = номера вершины

int \*\* adjacency\_list; // список смежности

int vertices; // вершин

int edges; // ребер

//std::vector<std::string> names;

//int \* names;

int al\_v;

int am\_v;

int im\_v;

int im\_e;

bool is\_orgraph;

/\*void init\_names();

void copy\_names(Graph \*);\*/

void init\_inc\_matrix();

void init\_adj\_list();

void init\_adj\_matrix();

void destroy\_adj\_list();

void destroy\_adj\_matrix();

void destroy\_inc\_matrix();

void reinit\_adj\_list();

void reinit\_adj\_matrix();

void reinit\_inc\_matrix();

void clear\_inc\_matrix(int, int);

void clear\_adj\_matrix();

void clear\_all();

int calculate\_vertices(std::string \*);

void calculate\_vertices\_and\_edges(std::string \*);

void calculate\_edges();

void fill\_adjacency\_list();

void convert\_inc\_into\_adj();

void convert\_adj\_into\_inc();

void convert\_adj\_into\_inc\_orgraph();

void convert\_adj\_into\_inc\_not\_orgraph();

void converting\_from\_im();

void converting\_from\_am();

public:

Graph();

~Graph();

void help();

void init();

void copy\_from(Graph \*);

void print\_adjacency\_matrix() const;

void print\_incidence\_matrix();

void print\_adjacency\_list();

void print\_out();

int get\_vertices() const;

int get\_edges();

bool is\_oriented\_graph();

void read\_adjacency\_matrix(std::string \*);

void read\_incidence\_matrix(std::string \*);

void get\_v\_degree(int, int \*);

int \* indegree\_and\_outcome();

int \* get\_degree\_sequence();

void add\_vortex();

void delete\_vortex(int);

void add\_edge(int, int);

void delete\_edge(int);

bool delete\_edge\_2(int, int);

void complement\_graph(); //Дополнение графа

void edge\_subdivision(int); //Подразбиение ребра

//void vertex\_involving(int, int); //Отождествление вершин

void vertex\_involving\_2(int, int); //Отождествление вершин

void graph\_contraction(int \*, int); //Стягивание графа

void dupliacate\_vertices(int); //Дублирование

void vertices\_reproduction(int); //Размножение

void graphs\_union(Graph \*, Graph \*);

void graphs\_connection(Graph \*, Graph \*);

void product\_of\_graphs(Graph \*, Graph \*);

};

#endif

Graph.cpp

#include "Graph.hpp"

#include <iostream>

#include <fstream>

Graph::Graph()

{

init();

}

Graph::~Graph()

{

clear\_all();

}

int Graph::calculate\_vertices(std::string \* str)

{

std::ifstream fin(str->c\_str());

int count;

char c;

for (count = 0; ; count++)

{

c = fin.peek();

if (c == '\n') break;

fin >> c;

}

fin.close();

return count;

}

void Graph::calculate\_vertices\_and\_edges(std::string \* str)

{

std::ifstream fin(str->c\_str());

char c;

int i;

vertices = 0;

for (edges = 0;; edges++)

{

c = fin.peek();

if (c == '\n') {

vertices++;

break;

}

if (c == EOF)

{

if (edges != 0) vertices++;

fin.close();

return;

}

fin >> i;

}

while (true)

{

c = fin.peek();

if (c == '\n') vertices++;

if (c == EOF)

{

fin.close();

return;

}

fin >> i;

}

/\*char c;

for (edges = vertices = 0;; edges++)

{

c = fin.peek();

if (c == '\n') vertices++;

if (c == EOF) break;

fin >> c;

if (c == '-') edges--;

}

edges /= (++vertices);\*/

//fin.close();

}

/\*-----------reading-----------\*/

/\*------------begin------------\*/

void Graph::read\_adjacency\_matrix(std::string \* str)

{

clear\_all();

vertices = calculate\_vertices(str);

init\_adj\_matrix();

std::ifstream fin(str->c\_str());

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

{

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

{

fin >> adjacency\_matrix[i][j];

if (i == j && (adjacency\_matrix[i][j] % 2 == 1)) is\_orgraph = true;

if (i > j && (adjacency\_matrix[i][j] != adjacency\_matrix[j][i])) is\_orgraph = true;

}

}

fin.close();

calculate\_edges();

init\_inc\_matrix();

convert\_adj\_into\_inc();

init\_adj\_list();

fill\_adjacency\_list();

//init\_names();

}

void Graph::read\_incidence\_matrix(std::string \* str)

{

clear\_all();

calculate\_vertices\_and\_edges(str);

init\_inc\_matrix();

std::ifstream fin(str->c\_str());

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

{

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

{

fin >> incidence\_matrix[i][j];

if (incidence\_matrix[i][j] == -1 && is\_orgraph == false) is\_orgraph = true;

}

}

fin.close();

init\_adj\_matrix();

convert\_inc\_into\_adj();

init\_adj\_list();

fill\_adjacency\_list();

//init\_names();

}

/\*-----------reading-----------\*/

/\*-------------end-------------\*/

/\*-----------printing----------\*/

/\*------------begin------------\*/

void Graph::print\_adjacency\_matrix() const

{

/\*if (is\_adj\_matrix == false) {

std::cout << "Adjacency\_Matrix is not ready" << std::endl;

return;

}\*/

//printf("\nAdjacency\_Matrix\n");

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

{

//printf("%d\t|", names[i]);

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

{

printf("%d ", adjacency\_matrix[i][j]);

}

printf("\n");

}

}

void Graph::print\_incidence\_matrix()

{

//if (is\_inc\_matrix == false) convert\_adj\_into\_inc();

//printf("\nIncidence\_Matrix\n");

if (get\_edges() == -1) calculate\_edges();

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

{

//printf("%d\t|", names[i]);

for (int j = 0; j < im\_e; j++)

{

if (!is\_orgraph) printf("%d ", incidence\_matrix[i][j]);

else printf("%d\t", incidence\_matrix[i][j]);

}

printf("\n");

}

}

void Graph::print\_adjacency\_list()

{

//if (is\_adj\_list == false) fill\_adjacency\_list();

//printf("\nAdjacency\_List\n");

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

{

//printf("{%d", names[i]);

printf("{%d", i);

for (int j = 0; adjacency\_list[i][j] != -1; j++)

{

printf(" %d", adjacency\_list[i][j]);

}

printf("}\n");

}

}

/\*-----------printing----------\*/

/\*-------------end-------------\*/

/\*-----------clearing----------\*/

/\*------------begin------------\*/

void Graph::clear\_adj\_matrix()

{

for (int i = 0; i < am\_v; i++) for (int j = 0; j < am\_v; j++) adjacency\_matrix[i][j] = 0;

}

void Graph::clear\_inc\_matrix(int x, int y)

{

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

{

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

{

incidence\_matrix[i][j] = 0;

}

}

}

void Graph::clear\_all()

{

if (vertices < 0) return;

for (int i = 0; i < vertices; i++) free(adjacency\_matrix[i]);

free(adjacency\_matrix);

for (int i = 0; i < vertices; i++) free(incidence\_matrix[i]);

free(incidence\_matrix);

for (int i = 0; i < vertices; i++) free(adjacency\_list[i]);

free(adjacency\_list);

adjacency\_matrix = NULL;

incidence\_matrix = NULL;

adjacency\_list = NULL;

vertices = 0;

edges = 0;

is\_orgraph = false;

//names.clear();

//if (names) free(names);

am\_v = 0;

al\_v = 0;

im\_v = 0;

im\_e = 0;

}

/\*-----------clearing----------\*/

/\*-------------end-------------\*/

/\*---------converting----------\*/

/\*-----------begin-------------\*/

void Graph::fill\_adjacency\_list()

{

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

{

int count = 0;

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

{

int temp = adjacency\_matrix[i][j];

if (temp > 0)

{

if (i == j && !is\_orgraph)

{

temp = temp >> 1;

}

temp += count;

adjacency\_list[i] = (int \*)realloc((void \*)adjacency\_list[i], (temp + 1) \* sizeof(int));

while (count < temp)

{

adjacency\_list[i][count] = j + 1;

count++;

}

adjacency\_list[i][count] = -1;

}

}

}

}

void Graph::convert\_inc\_into\_adj()

{

clear\_adj\_matrix();

int in, out;

bool loop;

int loops = 0;

for (int i = 0; i < vertices; i++) if (adjacency\_matrix) if (adjacency\_matrix[i][i] > 0) loops += (adjacency\_matrix[i][i] >> 1);

for (int j = 0; j < edges - loops; j++)

{

in = out = -1;

loop = false;

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

{

int temp = incidence\_matrix[i][j];

if (temp == 1)

{

if (is\_orgraph) out = i;

else if (in == -1) in = i;

else out = i;

}

else if (temp == -1) in = i;

else if (temp == 2)

{

loop = true;

if (is\_orgraph) adjacency\_matrix[i][i]++;

else adjacency\_matrix[i][i] += 2;

}

}

if (loop) continue;

if (is\_orgraph) adjacency\_matrix[out][in]++;

else

{

adjacency\_matrix[out][in]++;

adjacency\_matrix[in][out]++;

}

}

}

void Graph::convert\_adj\_into\_inc()

{

(is\_orgraph) ? convert\_adj\_into\_inc\_orgraph() : convert\_adj\_into\_inc\_not\_orgraph();

}

void Graph::convert\_adj\_into\_inc\_orgraph()

{

int y = 0;

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

{

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

{

int temp = adjacency\_matrix[i][j];

if (temp > 0)

{

temp += y;

if (i == j)

{

for (; y < temp; y++)

{

incidence\_matrix[i][y] = 2;

}

}

else

{

for (; y < temp; y++)

{

incidence\_matrix[i][y] = 1;

incidence\_matrix[j][y] = -1;

}

}

}

}

}

}

void Graph::convert\_adj\_into\_inc\_not\_orgraph()

{

int y = 0;

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

{

for (int j = i; j < vertices; j++)

{

int temp = adjacency\_matrix[i][j];

if (temp > 0)

{

if (i == j)

{

temp = temp >> 1;

temp += y;

while (y < temp)

{

incidence\_matrix[i][y] = 2;

y++;

}

}

else

{

temp += y;

while (y < temp)

{

incidence\_matrix[i][y] = incidence\_matrix[j][y] = 1;

y++;

}

}

}

}

}

}

void Graph::converting\_from\_im()

{

im\_v = vertices;

im\_e = edges;

reinit\_adj\_list();

reinit\_adj\_matrix();

convert\_inc\_into\_adj();

fill\_adjacency\_list();

}

void Graph::converting\_from\_am()

{

calculate\_edges();

reinit\_inc\_matrix();

convert\_adj\_into\_inc();

//print\_incidence\_matrix();

reinit\_adj\_list();

fill\_adjacency\_list();

}

/\*---------converting----------\*/

/\*------------end--------------\*/

/\*----------build-up-----------\*/

/\*-----------begin-------------\*/

void Graph::init()

{

adjacency\_matrix = NULL;

incidence\_matrix = NULL;

adjacency\_list = NULL;

vertices = 0;

edges = 0;

//names = NULL;

is\_orgraph = false;

al\_v = 0;

am\_v = 0;

im\_v = 0;

im\_e = 0;

}

void Graph::init\_adj\_matrix()

{

adjacency\_matrix = (int \*\*)malloc(vertices \* sizeof(int \*));

for (int i = 0; i < vertices; i++) adjacency\_matrix[i] = (int \*)malloc(vertices \* sizeof(int));

am\_v = vertices;

//names = (int \*)malloc(vertices \* sizeof(int));

//for (int i = 0; i < vertices; i++) names[i] = i;

//for (int i = 0; i < vertices; i++) names.push\_back();

}

/\*void Graph::init\_names()

{

for (int i = 0; i < vertices; i++) names[i] = i + 1;

}\*/

void Graph::init\_inc\_matrix()

{

int loops = 0;

//for (int i = 0; i < vertices; i++) if (adjacency\_matrix) if (adjacency\_matrix[i][i] > 0) loops += (adjacency\_matrix[i][i] >> 1);

incidence\_matrix = (int \*\*)malloc(vertices \* sizeof(int \*));

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

{

//incidence\_matrix[i] = (int \*)malloc((edges - loops) \* sizeof(int));

incidence\_matrix[i] = (int \*)malloc(edges \* sizeof(int));

}

im\_v = vertices;

//im\_e = edges - loops;

im\_e = edges;

clear\_inc\_matrix(im\_v, im\_e);

}

void Graph::init\_adj\_list()

{

adjacency\_list = (int \*\*)malloc(vertices \* sizeof(int \*));

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

{

adjacency\_list[i] = (int \*)malloc(sizeof(int));

adjacency\_list[i][0] = -1;

}

al\_v = vertices;

}

void Graph::destroy\_adj\_list()

{

if (adjacency\_list == NULL) return;

for (int i = 0; i < al\_v; i++) free(adjacency\_list[i]);

free(adjacency\_list);

adjacency\_list = NULL;

}

void Graph::destroy\_adj\_matrix()

{

if (adjacency\_matrix == NULL) return;

for (int i = 0; i < am\_v; i++) free(adjacency\_matrix[i]);

free(adjacency\_matrix);

adjacency\_matrix = NULL;

}

void Graph::destroy\_inc\_matrix()

{

if (incidence\_matrix == NULL) return;

for (int i = 0; i < im\_v; i++) free(incidence\_matrix[i]);

free(incidence\_matrix);

incidence\_matrix = NULL;

}

void Graph::reinit\_adj\_list()

{

destroy\_adj\_list();

init\_adj\_list();

}

void Graph::reinit\_adj\_matrix()

{

destroy\_adj\_matrix();

init\_adj\_matrix();

}

void Graph::reinit\_inc\_matrix()

{

destroy\_inc\_matrix();

init\_inc\_matrix();

}

/\*----------build-up----------\*/

/\*------------end-------------\*/

int Graph::get\_vertices() const

{

return vertices;

}

int Graph::get\_edges()

{

calculate\_edges();

return edges;

}

void Graph::calculate\_edges()

{

edges = 0;

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

{

int j = (is\_orgraph) ? 0 : i;

for (; j < vertices; j++)

{

int temp = adjacency\_matrix[i][j];

if (temp > 0)

{

edges += (i == j && !is\_orgraph) ? (temp >> 1) : temp;

}

}

}

}

void Graph::get\_v\_degree(int v, int \*result)

{

result[0] = result[1] = 0;

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

{

result[0] += adjacency\_matrix[v][i];

if (is\_orgraph) result[1] += adjacency\_matrix[i][v];

}

}

int \* Graph::indegree\_and\_outcome()

{

int \* in\_out = (int \*)calloc(vertices \* 2, sizeof(int));

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

{

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

{

in\_out[i] += adjacency\_matrix[i][j];

in\_out[i + vertices] += adjacency\_matrix[j][i];

}

}

return in\_out;

}

int \* Graph::get\_degree\_sequence()

{

if (is\_orgraph) return indegree\_and\_outcome();

int \* sqnce = (int \*)malloc(vertices\*2\*sizeof(int));

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

{

sqnce[i] = 0;

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

{

sqnce[i] += adjacency\_matrix[i][j];

}

}

for (int i = 0; i < vertices; i++) sqnce[i + vertices] = i;

int max\_index = -1;

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

{

max\_index = i;

for (int j = i; j < vertices; j++)

{

if (sqnce[max\_index] < sqnce[j])

{

int temp = sqnce[max\_index];

sqnce[max\_index] = sqnce[j];

sqnce[j] = temp;

temp = sqnce[max\_index + vertices];

sqnce[max\_index + vertices] = sqnce[j + vertices];

sqnce[j + vertices] = temp;

}

}

}

return sqnce;

}

void Graph::help()

{

std::cout << std::endl;

std::cout << "help\t-\tВыдать меню команд" << std::endl;

std::cout << "exit\t-\tВыход из программы" << std::endl;

std::cout << std::endl;

std::cout << "new\t-\tСоздать новый граф" << std::endl;

std::cout << "change (cng)\t-\tВыбрать граф для последующей работы" << std::endl;

std::cout << "copy (cpy)\t-\tСкопировать состояние из другого графа" << std::endl;

std::cout << std::endl;

std::cout << "read\_am\t-\tСчитать матрицу смежности из файла input.txt" << std::endl;

std::cout << "read\_im\t-\tСчитать матрицу инцидентности из файла input2.txt" << std::endl;

std::cout << std::endl;

std::cout << "get\_v\t-\tОпределение числа вершин" << std::endl;

std::cout << "get\_e\t-\tОпределение числа ребер" << std::endl;

std::cout << "get\_deg\t-\tОпределение степени вершины" << std::endl;

std::cout << "get\_deg\_sqnce\t-\tОпределение степенной последовательности графа" << std::endl;

std::cout << std::endl;

std::cout << "print\_am\t-\tОпределение матрицы смежности" << std::endl;

std::cout << "print\_im\t-\tОпределение матрицы инцидентности" << std::endl;

std::cout << "print\_al\t-\tОпределение списка смежности" << std::endl;

std::cout << "print\_out\t-\tВывод в файл output.txt как список смежностей" << std::endl;

std::cout << std::endl;

std::cout << "add\_v\t-\tДобавление вершин" << std::endl;

std::cout << "del\_v\t-\tУдаление вершин" << std::endl;

std::cout << "add\_e\t-\tДобавление ребра" << std::endl;

std::cout << "del\_e\t-\tУдаление ребра" << std::endl;

std::cout << std::endl;

std::cout << "1\t-\tОпределить дополнения графа" << std::endl;

std::cout << "2\t-\tПодразбиение ребра" << std::endl;

std::cout << "3\t-\tСтягивание графа" << std::endl;

std::cout << "4\t-\tОтождествление вершин" << std::endl;

std::cout << "5\t-\tДублирование вершин" << std::endl;

std::cout << "6\t-\tРазмножение вершин" << std::endl;

std::cout << "7\t-\tОбъединение (дизъюнктивное) графов" << std::endl;

std::cout << "8\t-\tСоединение графов" << std::endl;

std::cout << "9\t-\tПроизведение графов" << std::endl;

std::cout << std::endl;

}

bool Graph::is\_oriented\_graph()

{

return is\_orgraph;

}

void Graph::copy\_from(Graph \*G)

{

vertices = G->vertices;

reinit\_adj\_matrix();

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

{

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

{

adjacency\_matrix[i][j] = G->adjacency\_matrix[i][j];

}

}

//copy\_names(G);

converting\_from\_am();

}

/\*void Graph::copy\_names(Graph \*G)

{

int ver = G->vertices;

names = (int \*)malloc(ver \* sizeof(int));

for (int i = 0; i < ver; i++) names[i] = G->names[i];

}\*/

/\*------------------------------------------\*/

/\*------------------------------------------\*/

/\*------------------------------------------\*/

/\*------------------------------------------\*/

void Graph::add\_vortex()

{

vertices++;

incidence\_matrix = (int \*\*)realloc((void \*)incidence\_matrix, vertices \* sizeof(int \*));

incidence\_matrix[vertices - 1] = (int \*)calloc(edges, sizeof(int));

converting\_from\_im();

}

void Graph::delete\_vortex(int v)

{

//Удаление всех инцидентных ребер

for (int i = edges - 1; i >= 0; i--)

{

int temp = incidence\_matrix[v][i];

if (temp == 1 || temp == -1 || temp == 2)

{

//копирование (сдвиг) столбцов

delete\_edge(i);

}

}

//Удаление самой вершины

//

//копирование (сдвиг) строк

for (int k = v + 1; k < vertices; k++)

{

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

{

incidence\_matrix[k - 1][l] = incidence\_matrix[k][l];

}

}

//удаление лишнего (крайнего) столбца

vertices--;

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

{

incidence\_matrix = (int \*\*)realloc((void \*)incidence\_matrix, vertices \* sizeof(int \*));

}

converting\_from\_im();

}

void Graph::add\_edge(int out, int in)

{

edges++;

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

{

incidence\_matrix[i] = (int \*)realloc((void \*)incidence\_matrix[i], edges \* sizeof(int));

incidence\_matrix[i][edges - 1] = 0;

}

incidence\_matrix[out][edges - 1]++;

(is\_orgraph && in != out) ? incidence\_matrix[in][edges - 1]-- : incidence\_matrix[in][edges - 1]++;

converting\_from\_im();

}

void Graph::delete\_edge(int e)

{

//копирование (сдвиг) столбцов

for (int k = e + 1; k < edges; k++)

{

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

{

incidence\_matrix[l][k - 1] = incidence\_matrix[l][k];

}

}

//удаление лишнего (крайнего) столбца

edges--;

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

{

incidence\_matrix[k] = (int \*)realloc((void \*)incidence\_matrix[k], edges \* sizeof(int));

}

}

bool Graph::delete\_edge\_2(int x, int y)

{

if (!is\_orgraph)

{

if (adjacency\_matrix[x][y] && adjacency\_matrix[y][x])

{

adjacency\_matrix[x][y]--;

adjacency\_matrix[y][x]--;

}

else return true;

}

else

{

if (adjacency\_matrix[x][y]) adjacency\_matrix[x][y]--;

else return true;

}

edges--;

converting\_from\_am();

return false;

}

void Graph::print\_out()

{

std::ofstream fout("output.txt", std::ios\_base::trunc);

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

{

fout << "{" << i;

//printf("{%d", i);

for (int j = 0; adjacency\_list[i][j] != -1; j++)

{

fout << " " << adjacency\_list[i][j];

//printf(" %d", adjacency\_list[i][j]);

}

fout << "}";

//printf("}\n");

}

fout.close();

}

/\*------------------------------------------\*/

/\*------------------------------------------\*/

/\*------------------------------------------\*/

/\*------------------------------------------\*/

void Graph::complement\_graph()

{

/\*for (int i = 0; i < vertices; i++)

{

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

{

adjacency\_matrix[i][j] = 1;

}

adjacency\_matrix[i][i] = 0;

}

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

{

for (int j = 0; adjacency\_list[i][j] != -1; j++)

{

adjacency\_matrix[i][adjacency\_list[i][j]] = 0;

}

}\*/

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

{

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

{

int new\_val = (adjacency\_matrix[i][j] > 0) ? 0 : 1;

adjacency\_matrix[i][j] = adjacency\_matrix[j][i] = new\_val;

}

}

for (int i = 0; i < vertices; i++) adjacency\_matrix[i][i] = 0;

calculate\_edges();

converting\_from\_am();

}

void Graph::edge\_subdivision(int e)

{

int out, in;

out = in = -1;

bool loop = false;

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

{

int temp = incidence\_matrix[i][e];

if (temp == 2)

{

loop = true;

out = in = i;

break;

}

else if (temp == 1)

{

if (out != -1){

in = i;

break;

}

else out = i;

}

else if (temp == -1) in = i;

}

delete\_edge(e);

vertices++;

incidence\_matrix = (int \*\*)realloc((void \*)incidence\_matrix, vertices \* sizeof(int));

edges += 2;

incidence\_matrix[vertices - 1] = (int \*)malloc(edges \* sizeof(int));

for (int i = 0; i < edges - 2; i++) incidence\_matrix[vertices - 1][i] = 0;

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

{

incidence\_matrix[i] = (int \*)realloc((void \*)incidence\_matrix[i], edges \* sizeof(int));

incidence\_matrix[i][edges - 2] = incidence\_matrix[i][edges - 1] = 0;

}

int new\_var;

new\_var = (is\_orgraph) ? -1 : 1;

incidence\_matrix[out][edges - 2] = incidence\_matrix[vertices - 1][edges - 1] = 1;

incidence\_matrix[vertices - 1][edges - 2] = incidence\_matrix[in][edges - 1] = new\_var;

converting\_from\_im();

}

void Graph::vertex\_involving\_2(int x, int y)

{

add\_vortex();

int check;

if (adjacency\_matrix[x][x] || adjacency\_matrix[y][y])

{

add\_edge(vertices - 1, vertices - 1);

//adjacency\_matrix[vertices - 1][vertices - 1] = (is\_orgraph) ? 1 : 2;

check = adjacency\_matrix[vertices - 1][vertices - 1];

}

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

{

if (i != x && i != y)

{

if (adjacency\_matrix[i][x] || adjacency\_matrix[i][y]) add\_edge(i, vertices-1); //adjacency\_matrix[i][vertices - 1] = 1;

check = adjacency\_matrix[i][vertices - 1];

if (is\_orgraph)

{

if (adjacency\_matrix[x][i] || adjacency\_matrix[y][i]) add\_edge(vertices - 1, i); //adjacency\_matrix[vertices - 1][i] = 1;

check = adjacency\_matrix[vertices - 1][i];

}

}

}

//print\_adjacency\_matrix();

int max, min;

if (x > y)

{

max = x;

min = y;

}

else

{

max = y;

min = x;

}

//converting\_from\_am();

delete\_vortex(max);

//print\_adjacency\_matrix();

delete\_vortex(min);

//print\_adjacency\_matrix();

}

void Graph::graph\_contraction(int \* v\_index, int size)

{

int var\_1, var\_2;

var\_1 = v\_index[0];

var\_2 = v\_index[1];

for (int k = 2; k < size; k++)

{

vertex\_involving\_2(var\_1, var\_2);

//print\_incidence\_matrix();

//print\_adjacency\_matrix();

var\_1 = get\_vertices() - 1;

//int check;

int v\_k = v\_index[k];

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

{

if (v\_index[i] < v\_k) v\_index[k]--;

//check = v\_index[k];

}

var\_2 = v\_index[k];

}

vertex\_involving\_2(var\_1, var\_2);

//converting\_from\_im();

}

void Graph::dupliacate\_vertices(int x)

{

add\_vortex();

int loop = adjacency\_matrix[x][x];

if (is\_orgraph == false) loop = loop >> 1;

for (int i = 0; i < loop; i++) add\_edge(vertices - 1, vertices - 1);

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

{

if (i == x) continue;

int count = adjacency\_matrix[i][x];

if (count){

for (int j = 0; j < count; j++) add\_edge(i, vertices - 1);

}

if (is\_orgraph)

{

int count = adjacency\_matrix[x][i];

if (count){

for (int j = 0; j < count; j++) add\_edge(vertices - 1, i);

}

}

}

}

void Graph::vertices\_reproduction(int x)

{

dupliacate\_vertices(x);

add\_edge(x, vertices - 1);

if (is\_orgraph) add\_edge(vertices - 1, x);

}

/\*------------------------------------------\*/

/\*------------------------------------------\*/

/\*------------------------------------------\*/

/\*------------------------------------------\*/

void Graph::graphs\_union(Graph \* G1, Graph \* G2)

{

if (G1->vertices < G2->vertices)

{

Graph \* temp = G1;

G1 = G2;

G2 = temp;

}

vertices = G1->vertices;

reinit\_adj\_matrix();

//clear\_adj\_matrix();

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

{

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

{

adjacency\_matrix[i][j] = G1->adjacency\_matrix[i][j];

}

}

for (int i = 0; i < G2->vertices; i++)

{

for (int j = 0; j < G2->vertices; j++)

{

if (G2->adjacency\_matrix[i][j] > adjacency\_matrix[i][j]) adjacency\_matrix[i][j] = G2->adjacency\_matrix[i][j];

}

}

//print\_adjacency\_matrix();

converting\_from\_am();

}

void Graph::graphs\_connection(Graph \* G1, Graph \* G2)

{

int g1\_v, g2\_v;

g1\_v = G1->vertices;

g2\_v = G2->vertices;

if (g1\_v < g2\_v)

{

Graph \* temp = G1;

G1 = G2;

G2 = temp;

}

vertices = g1\_v + g2\_v;

reinit\_adj\_matrix();

for (int i = 0; i < g1\_v; i++)

{

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

{

adjacency\_matrix[i][j] = (j < G1->vertices) ? G1->adjacency\_matrix[i][j] : 1;

}

}

for (int i = g1\_v; i < vertices; i++)

{

for (int j = 0; j < g1\_v; j++)

{

adjacency\_matrix[i][j] = 1;

}

}

for (int i = g1\_v; i < vertices; i++)

{

for (int j = g1\_v; j < vertices; j++)

{

adjacency\_matrix[i][j] = G2->adjacency\_matrix[i - g1\_v][j - g1\_v];

}

}

converting\_from\_am();

}

void Graph::product\_of\_graphs(Graph \* G1, Graph \* G2)

{

if (G1->vertices < G2->vertices)

{

Graph \* temp = G1;

G1 = G2;

G2 = temp;

}

vertices = G1->vertices \* G2->vertices;

reinit\_adj\_matrix();

clear\_adj\_matrix();

for (int k = 0; k < G2->vertices; k++)

{

int x = 0;

for (int i = k \* G1->vertices; i < (k + 1) \* G1->vertices; i++, x++)

{

int y = 0;

for (int j = k \* G1->vertices; j < (k + 1) \* G1->vertices; j++, y++)

{

adjacency\_matrix[i][j] = G1->adjacency\_matrix[x][y];

}

}

}

for (int i = 0; i < G2->vertices; i++)

{

for (int j = 0; j< G2->vertices; j++)

{

int value = G2->adjacency\_matrix[i][j];

if (value > 0)

{

int start\_x = i \* G1->vertices;

int start\_y = j \* G1->vertices;

int end\_x = (i + 1) \* G1->vertices;

while (start\_x < end\_x)

{

adjacency\_matrix[start\_x][start\_y] += value;

start\_x++;

start\_y++;

}

}

}

}

converting\_from\_am();

}