# Міністерство науки і освіти України

Харківський національний університет радіоелектроніки

Кафедра Інформаційних управляючих систем

КУРСОВА РОБОТА

з дисципліни

ТЕХНОЛОГІЇ ОБ’ЄКТНО ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

на тему:

«Розробка об’єктно-орієнтованої програми для обробки графів»

|  |  |
| --- | --- |
| Роботу виконав | Керівник |
| ст. гр.\_ІТУ-19-2 \_ | \_доцент\_ \_Білова Тетяна Георгіївна\_  (посада) (ПІБ) |
| \_Куренков Богдан Михайлович\_  (ПІБ) | До захисту |
| “\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ р. | “\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ р. |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (підпис) | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (підпис) |

Захист курсової роботи

\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ Голова комісії \_\_\_\_\_\_\_ \_\_\_Білова Т.Г.\_\_\_

(дата) (оцінка) (підпис) (ПІБ)

Члени комісії \_\_\_\_\_\_\_ Севостьянова К.А.

(підпис) (ПІБ)

\_\_\_\_\_\_\_ \_\_Малькова І.А.\_\_

(підпис) (ПІБ

Харків 2020

# Харківський національний університет радіоелектроніки

Факультет Комп’ютерних наук

Кафедра Інформаційних управляючих систем

Спеціальність 122 – Комп’ютерні науки

Освітня програма Інформаційні технології управління

**ЗАВДАННЯ**

**на курсову роботу з дисципліни «Технології об’єктно-орієнтованого програмування»**

студенту\_\_Куренову Богдану Михайловичу\_\_

(прізвище, ім’я та по батькові)

групи \_ІТУ-19-2\_\_

1. Тема роботи «Розробка об’єктно-орієнтованої програми для обробки графів»

2. Строк захисту студентом закінченої роботи 11.06.2020

3. Вихідні дані до роботи: опис предметної області, структура \_Edge, клас \_Graph.

4. Зміст пояснювальної записки (перелік питань, що повинні бути розроблені) введення, опис призначення та можливостей мови програмування, опис вхідної та вихідної інформації програми, опис використаних у програмі класів та інших засобів мови програмування, схема алгоритму роботи програми, текст програми з коментарями, висновки.

5. Перелік графічного матеріалу (с точним зазначенням обов’язкових креслень) діаграма класів, алгоритм програми, екранні форми

6. Дата видачі завдання 19.03.2020

# **КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Назва етапів курсової роботи | Термін виконання | Примітка |
| 1 | Отримання завдання на курсову роботу | 19.03.20 |  |
| 2 | Аналіз предметної області | 22.03.20 |  |
| 3 | Побудова діаграми класів обліку номерів | 1.04.20 |  |
| 4 | Побудова діаграми класів графічного інтерфейсу | 14.04.20 |  |
| 5 | Побудова схеми алгоритму роботи програми | 24.03.20 |  |
| 6 | Розробка програми | 1.05.20 |  |
| 7 | Налагодження програми | 20.05.20 |  |
| 8 | Тестування програми | 25.05.20 |  |
| 9 | Оформлення пояснувальної записки | 05.06.20 |  |
| 10 | Захист курсової роботи | 11.06.20 |  |

Керівник\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(Білова Т.Г.)

Студент\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(Куренков Б.М.)

# **РЕФЕРАТ**

Пояснювальна записка до курсової роботи містіть: 33 стор., 13 рис., 1 додаток, 2 джерела.

Мета роботи - розробка об’єктно-орієнтованої програми для обробки графів, що орієнтована під платформу Windows.

Було проведено опис вхідної та вихідної інформації задачі, проаналізований вибір програмних засобів розробки програмного забезпечення програми, розроблені основні алгоритми для різних задач.

Програма допомогає швидко та комфортно отримати необхідні дані про граф для використання у геоінформаційних задачах.

Областю застосування проектованого програмного забезпечення є геоінформаційні системи.

ГРАФ, МАТРИЦЯ ВАГ, МІНІМАЛЬНИЙ ШЛЯХ, МІНІМАЛЬНИЙ ОСТОВ, КЛАС, С++, ООП, СХЕМА АЛГОРИТМУ, ДІАГРАМА КЛАСІВ, ІНТЕРФЕЙС.

# **ЗМІСТ**

Перелік умовних позначень, символів, одиниць, скорочень і термінів……………………………………………………………………………….6

Введення……………………………………………………………………………...7

1.Аналіз предметної області і постановка задач курсової роботи………………………...………………………………………………………9

2.Опис призначення і можливостей мови програмування……………………….11

3.Опис вхідний і вихідний інформації програми………………………………….13

4.Опис використаних в програмі класів та інших засобів мови програмування……………………………………………..…………………….....14

5.Текст програми з коментарями………………………………….……………….17

Висновки……………………………..……………………………………..……….28

Перелік посилань………………………………………………………….………..29

Додаток А Приклад виконання програми………………………………………….30

# **ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ**

ООП **–** об’єктно-орієнтованне програмування.

Граф **–** це сукупність об’єктів із зв’язками між ними. Об’єкти розглядаються як вершини, а зв’язки – як дуги, або ребра.

Ребро – ненаправлений зв’язок між вершинами графа.

Дуга – направлений зв’язок між вершинами графа.

Об’єкт – структура даних, що містить опис властивостей зовнішнього об’єкта програмування.

Метод – функція, яка працює з об’єктом.

Клас – опис структури об’єкта і методів роботи з ним.

Подія – метод, що спрацьовує при контакті користувача з елементами графічного інтерфейсу при певних обставинах.

Лог – список подій, що кояться в програмі.

Інцидентність – термін, що демонструє зв’зок між двома вершинами.

Підграф – граф, що містить деяку підмножину вершин даного графа і всі ребра, інцидентні даній підмножині.

Мінімальний остов графа – шлях, яким слід йти за для проходу через усі вершини графа, витративші мінімальні зусилля.

Компонента зв’язності графа – це підграф, в якому будь-які дві вершини зв’язані одна з одною шляхами, і вони не зв’язані з ніякими додатковими вершинами.

Петля у графі – ребро, що виходить деякої вершини та повертається у неї ж.

**ВВЕДЕННЯ**

Теорiя графiв - роздiл математики, в якому вивчаються графи та їх властивостi. Графи звичайно зображують у виглядi точок (вершин), з’єднаних вiдрiзками (ребрами). Графи можна задавати графiчно, у виглядi спискiв ребер, за допомогою матриць iнцидентностi та сумiжностi, що значно спрощує створення математичних моделей на основi вiзуального представлення. За допомогою графiв можна вiзуально представити об’єкти та вiдношення, або зв’язки, мiж ними. Все в реальному світі пов’язано: міста можна представити як мережу вулиць, залiзничну та польотну мережу. Сторiнки в інтернет пов’язанi посиланнями. Рiзнi компоненти електричного та електронного ланцюга або мікросхема комп’ютера пiдключенi. Шляхи спалахiв захворювань утворюють мережу. Аналiз та оптимiзацiя мереж можуть бути виконані за допомогою теориії графiв. Граф є одним iз іструментів моделювання. Моделi рiзних об’єктiв у виглядi графiв можуть мати спільні властивостi. Загальнi властивостi певних груп графiв є предметом вивчення у теориії графiв.

Теорiю графiв широко застосовують у логiстицi. lнтелектуальнi транспорты системи можуть працювати, збираючи дан про мiсцезнаходження вiд навiгаторiв автомобілів i передавати iнформацiю водiям, де i як швидко їздити, щоб зменшити загальне перевантаження. Теорiя графiв вже використовуеться авiакомпанiями, якi хочуть з’єднати велику клькість мiст найефектившшим чином, створити систему перемiщення великої кiлькостi пасажирiв з найменшою кількістю можливих поїздок. Дана проблема схожа за своею суттю на задачу про комiвояжера. У той же час, авiадиспетчери повиннi переконатися, що сотні лiтакiв знаходяться в потрiбному мiсцi в потрiбний час i запобiгти можливим аварiям. Виршення цього завдання не було б можливим без комп’ютерiв i теорії графiв.

Теорiя графiв також грає важливу роль в аналiзi та вiзуалiзацiї еволюції тварин i мов, контролю натовпу i поширення захворювань

Вирішення геоінформаційних задач за допомогою графів та алгоритмів на них є актуальною на сьогоднішній день задачею.

Метою курсової роботи є розробка об’єктно-орієнтованої програми для обробки графів з графічним інтерфейсом для платформи Windows з використанням програмного забезпечення Microsoft Visual Studio.

Задачами дослідження є аналіз предметної області, об’єкта декомпозиція, визначення системи класів та відношення між ними, розробка графічного інтерфейсу.

**1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАДАЧ КУРСОВОЇ РОБОТИ**

У програмі слід реалізувати такі задачі, як: додавання дуг до графу, видалення дуг із графу, підрахунок кількосних характеристик графу (кількість вершин, кількість ребер, кількість компонент зв’язності), побудова матриці ваг графу (для матричної ілюстрації графу), побудова матриці найкоротших шляхів графу, пошук та побудова остову мінімальної ваги графу.

Потрібно виділити такі елементи як дуга та граф.

З операцій потрібно виділити наступні:

* додавання дуги до графу;
* видалення дуги до графу;
* пдрахунок кількості вершин у графі;
* підрахунок кількості ребер у графі;
* підрахунок кількості компонент зв’зності у графі;
* підрахунок матриці ваг графу;
* пошук найкоротших шляхів на графі за допомогою алгоритми Дейкстри;
* пошук мінімального остову графу.

З елементів графічного інтерфейсу потрібно виділити:

* поля для вводу даних про граф;
* кнопка додавання ребра до графу;
* кнопка видалення ребра з графу;
* поле для виводу інформації про граф;
* поле для виводу матриці ваг графу;
* поле для виводу матриці найкоротших шляхів графу;
* поле для виводу мінімального остову графу;
* поле для виводу помилок у разі їх виникнення.

Основними задачами курсової роботи є:

1. розробка ієрархії класів, їх методів та алгоритмів , які реалізують програму;
2. розробка графічного інтерфейсу з використанням Windows Forms.

**2 ОПИС ПРИЗНАЧЕННЯ І МОЖЛИВОСТЕЙ МОВИ ПРОГРАМУВАННЯ**

Мови програмування - це формальні штучні мови. Як і природні мови, вони мають алфавіт, словниковий запас, граматику і синтаксис, а також семантику.

Алфавіт - дозволений до використання набір символів, за допомогою якого можуть бути утворені слова і величини даного мови.

Синтаксис - система правил, що визначають допустимі конструкції мови програмування з букв алфавіту.

Семантика - система правил однозначного тлумачення кожної мовної конструкції, що дозволяють виробляти процес обробки даних.

Мови програмування, що імітують природні, що володіють укрупненими командами, орієнтовані «на людину», називають мовами високого рівня. Чим вище рівень мови, тим ближче структури даних і конструкції, які використовуються в програмі, до понять початкового завдання. Особливості конкретних комп’ютерних архітектур в них не враховуються, тому вихідні тексти програм легко переносяться на інші платформи, мають транслятори цієї мови. Розробляти програми на мовах високого рівня за допомогою зрозумілих і потужних команд значно простіше. Число помилок, що допускаються в процесі програмування, набагато менше. В даний час налічується кілька сотень таких мов (без урахування їх діалектів).

Таким чином, мови програмування високого рівня, орієнтовані на рішення великих змістовних прикладних задач, є апаратно-незалежними і вимагають використання відповідних програм-перекладачів для перетворення тексту програми в машинний код, який в результаті і обробляється процесором. У 90-х з’явилися мови, що дозволяють створювати веб-сценарії. Зараз мови програмування теж розвиваються - в напрямках безпеки, модульної організації коду, інтеграції з базами даних і надійності.

С++ - компільований строго типізований мова програмування загального призначення. Підтримує різні парадигми програмування: процедурну, узагальнену, функціональну; найбільшу увагу приділено підтримці об’єктно-орієнтованого програмування.

Об’єктно-орієнтоване програмування ввібрало в себе кращі ідеї структурного програмування і комбінує їх з новими потужними концепціями, що дозволяють побачити задачу програмування в новому світлі. Об’єктно-орієнтоване програмування дозволяє легко розкласти завдання на підгрупи взаємодіючих частин. Потім можна перетворити ці підгрупи в одиниці, звані об’єктами.

Основними поняттями ООП є клас, метод та об’єкти класів.

Всі об’єктно-орієнтовані мови мають три загальні концепції: інкапсуляцію, поліморфізм і успадкування.

Інкапсуляція – це механізм, який пов’язує разом код і дані і який зберігає їх від зовнішнього впливу і від неправильного використання. Ми можемо визначити частину коду і даних як власність об’єкта, яка недоступна ззовні.

Спадкування являє собою процес, завдяки якому один об’єкт може успадковувати, набувати властивостей від іншого об’єкта. Це властивість підтримує концепцію класифікації, що навіть обумовлюється його важливість. Ця концепція лежить в основі класифікації знань. Без використання класифікації кожен об’єкт мав би визначати всі свої характеристики явно. На основі класифікації об’єкт потребує тільки у визначенні таких якостей, які відрізняють його від інших об’єктів цього класу. Завдяки механізму успадкування об’єкт може характеризуватися в рамках класифікації спільних та власних.

Поліморфізм – це властивість, яке дозволяє одне і те ж ім’я використовувати для вирішення двох або більше схожих, але технічно різних завдань. Метою поліморфізму, стосовно об’єктно-орієнтованого програмування, є використання одного імені для завдання загальних для класу дій. Виконання кожного конкретного дії буде визначатися типом даних.

Windows Forms – інтерфейс програмування додатків (API), що відповідає за графічний інтерфейс користувача і є частиною Microsoft. NET Framework. Даний інтерфейс спрощує доступ до елементів інтерфейсу Microsoft Windows за рахунок створення обгортки для існуючого Win32 API в керованому коді.

**3 ОПИС ВХІДНИЙ І ВИХІДНИЙ ІНФОРМАЦІЇ ПРОГРАМИ**

Для початку роботи програми користувачу виконати наступні дії:

1. заповнити поля «Начало», «Конец», «Вес» інформацією про дугу;
2. натиснути кнопку «Добавить дугу»;

У процесі роботи програми користувачу потрібно буде додавати чи видаляти дуги за допомогою вищезазначених дій.

Процеси оновлення інформації та повідомлення користувача про некоректні дії запрограмовані на автоматичну роботу при зміні інформації про граф.

Після натискання кнопки закриття екрану користувач потрапляє на робочий стіл свого комп’ютера

**4 ОПИС ВИКОРИСТАНИХ В ПРОГРАМІ КЛАСІВ ТА ІНШИХ ЗАСОБІВ МОВИ ПРОГРАМУВАННЯ**

В ході об’єктної декомпозиції були виділені наступні класи: \_Graph, \_Edge

\_ Edge – структура, яка представляє собою однозв’язний список дуг.

Поля структури:

- int id\_from – поле структури, яке являє собою початок дуги.

- int id\_to – поле структури, яке являє собою кінець дуги.

- int weight – поле структури, яке являю собою вагу дуги.

- \_Edge\* next – поле структури, що вказує на наступний елемент структури.

\_Graph – клас-контейнер для обробки дуг.

Закриті поля класу \_Graph:

* \_Edge\* head – поле класу, яке являє собою вказівник на однозв’язний список дуг.

Методи \_ Graph:

* int Add(int id\_from, int id\_to, int weight) – метод додає дугу до графа та повертає результат операції - цифру (додана, недодана, перезаписана).
* int Pop(int id\_from, int id\_to) – метод видаляє дугу з зазначеними параметрами з однозв’язного списку у класі-контейнері та повертає результат операції - цифру (видалена, не знайдена, список пустий).
* int\*\* get\_weight\_matrix() – метод підраховує матрицю ваг графа та повертає її у вигляді двомірного масиву.
* int get\_count\_of\_unique() – метод підраховує та повертає кількість вершин графа.
* int\* get\_array\_of\_unique() – метод підраховує та повертає масив вершин графа у вигляді одномірного масиву.
* void dfs(int source, bool\* used, int\*\* matrix, int size) – допоміжний метод – алгоритм пошука в глибину (потрібен для пошуку кількості компонент зв’язності графа)
* int get\_connectivity\_component\_count() – метод підраховує кількість компонент зв’язності графа та повертає їх кількість.
* int\*\* get\_minimal\_ways() – метод підраховує найкоротші відстані на графі (реалізований алгоритм Дейкстри) та повертає матрицю мінімальних шляхів як двомірний массив.
* int\*\* get\_minimal\_ostov() – метод знаходить остов мінімальної ваги (реалізований алгоритм Прима) починаючи з першої вершини та повертає результат – множину інциденцій з тих дуг графа, котрі входять до остову у вигляді двомірного масиву.
* int get\_count\_of\_edges() – метод підраховує та повертає кількість дуг графа.

Діаграма розроблених класів представлена на рисунку 4.1

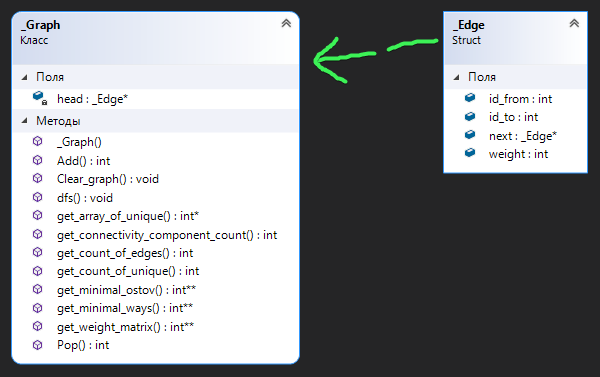
****

Рисунок 4.1– UML – діаграма класів

Графічний інтерфейс налічує на головній формі текстові поля «Начало», «Конец», «Вес», які являють собою текстові поля для вводу даних; кнопки «Добавить дугу», «Удалить дугу», «Очистить граф», які являють собою методи додавання та видалення інформації графа; закриті текстові поля «Инфо поле», «Минимальний остов», «Лог», «Матрица весов графа», «Матрица кратчайших путей», які являють собою інформаційні поля з результатами обробки графа відповідними алгоритмами.

Були використані обробники подій :

* only\_num\_KeyPress – при вводі у текстові поля забороняє користувачу вводити що завгодно, окрім цифр.
* deny\_paste\_KeyDown – коли користувач працює з текстовими полями, забороняє йому вставляти інформацію в них.
* add\_edge\_Click – клікаючи на кнопку «Добавить дугу», зчитує інформацію з текстових полів та додає дугу до графа, якщо це можливо. Інформаційні поля програми оновлюються за допомогою допоміжної функції info\_update().
* delete\_edge\_Click – клікаючи на кнопку «Удалить дугу», зчитує інформацію з текстових полів та видаляє дугу з графа, якщо це можливо. Інформаційні поля програми оновлюються за допомогою допоміжної функції info\_update().
* clear\_graph\_button\_Click – клікаючи на кнопку «Очистить граф», зчитує інформацію з текстових полів та перетворює нинішній граф у пустий. Інформаційні поля програми очищуються.

Допоміжні функції:

* void info\_update() – оновлює інформацію в деяких закритих текстових полях (необхідна для подій add\_edge\_Click та delete\_edge\_Click).

Приклад інтерфейсу можна побачити у додатку А.

**5 ТЕКСТ ПРОГРАМИ З КОМЕНТАРЯМИ**

**Graph.h**

#pragma once

// структура, которая представляет однозвязный список дуг

struct \_Edge {

int id\_from;

int id\_to;

int weight;

\_Edge\* next;

};

// класс-контейнер для оброботки дуг

class \_Graph {

\_Edge\* head;

public:

// для очистки графа

void Clear\_graph() {

while (head)

this->Pop(head->id\_from, head->id\_to);

}

// конструктор по умолчанию

\_Graph() : head(nullptr) {};

// для добавления дуги в граф

int Add(int id\_from, int id\_to, int weight) {

\_Edge\* tmp = head;

\_Edge\* tmp\_prev = tmp;

if (head)

while (tmp) {

if (tmp->id\_from == id\_from && tmp->id\_to == id\_to) {

if (tmp->weight > weight) {

tmp->weight = weight;

return 0;

}

return 1;

}

tmp\_prev = tmp;

tmp = tmp->next;

}

tmp = new \_Edge;

tmp->id\_from = id\_from;

tmp->id\_to = id\_to;

tmp->weight = weight;

tmp->next = nullptr;

if (!head)

head = tmp;

else

tmp\_prev->next = tmp;

return 2;

}

// для удаления дуги из графа

int Pop(int id\_from, int id\_to) {

if (!head) return 0;

int count = 0;

\_Edge\* tmp = head;

while (tmp->next) {

if (tmp->id\_from == id\_from && tmp->id\_to == id\_to)

break;

tmp = tmp->next;

++count;

}

if (!count) {

if (!tmp->next) {

if (tmp->id\_from != id\_from || tmp->id\_to != id\_to)

return 2;

delete head;

head = nullptr;

return 1;

}

else {

if (tmp->id\_from != id\_from || tmp->id\_to != id\_to)

return 2;

head = tmp->next;

tmp->next = nullptr;

delete tmp;

return 1;

}

}

else {

tmp = head;

\_Edge\* tmp\_prev = tmp;

while (tmp->next) {

if (tmp->id\_from == id\_from && tmp->id\_to == id\_to)

break;

tmp\_prev = tmp;

tmp = tmp->next;

}

if (tmp->id\_from == id\_from && tmp->id\_to == id\_to) {

tmp\_prev->next = tmp->next;

tmp->next = nullptr;

delete tmp;

return 1;

}

return 2;

}

}

// возвращает матрицу весов графа

int\*\* get\_weight\_matrix() {

\_Edge\* tmp = head;

int count\_of\_elems = 0;

while (tmp) {

tmp = tmp->next;

++count\_of\_elems;

}

int\* all\_vertexes = new int[count\_of\_elems \* 2];

tmp = head;

for (int i = 0; i < count\_of\_elems; ++i) {

all\_vertexes[2 \* i] = tmp->id\_from;

all\_vertexes[2 \* i + 1] = tmp->id\_to;

tmp = tmp->next;

}

for (int i = 0, tmp\_swap; i < count\_of\_elems \* 2; ++i) {

for (int j = i; j < count\_of\_elems \* 2; ++j) {

if (all\_vertexes[i] > all\_vertexes[j]) {

tmp\_swap = all\_vertexes[i];

all\_vertexes[i] = all\_vertexes[j];

all\_vertexes[j] = tmp\_swap;

}

}

}

int count\_of\_unique = 0;

for (int i = 0; i < count\_of\_elems \* 2 - 1; ++i)

if (all\_vertexes[i] != all\_vertexes[i + 1])

++count\_of\_unique;

if (count\_of\_unique)

++count\_of\_unique;

if (count\_of\_unique) {

int\*\* weight\_matrix = new int\* [count\_of\_unique];

for (int i = 0; i < count\_of\_unique; ++i)

weight\_matrix[i] = new int[count\_of\_unique];

for (int i = 0; i < count\_of\_unique; ++i)

for (int j = 0; j < count\_of\_unique; ++j)

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

int\* ids = new int[count\_of\_unique];

ids[0] = all\_vertexes[0];

for (int i = 0, j = 0; j < count\_of\_unique - 1; ++i) {

if (all\_vertexes[i] != all\_vertexes[i + 1])

ids[++j] = all\_vertexes[i + 1];

}

tmp = head;

for (int i = 0, id\_from = 0, id\_to = 0; i < count\_of\_elems; ++i, id\_from = 0, id\_to = 0) {

for (int j = 0; j < count\_of\_unique; ++j) {

if (ids[j] == tmp->id\_from)

id\_from = j;

}

for (int j = 0; j < count\_of\_unique; ++j) {

if (ids[j] == tmp->id\_to)

id\_to = j;

}

weight\_matrix[id\_from][id\_to] = tmp->weight;

//weight\_matrix[id\_to][id\_from] = tmp->weight;

tmp = tmp->next;

}

return weight\_matrix;

}

else

return nullptr;

}

// возвращает массив вершин графа

int\* get\_array\_of\_unique() {

int\* array\_of\_unique;

int count\_of\_unique = this->get\_count\_of\_unique();

array\_of\_unique = new int[count\_of\_unique];

\_Edge\* tmp = head;

int count\_of\_elems = 0;

while (tmp) {

tmp = tmp->next;

++count\_of\_elems;

}

int\* all\_vertexes = new int[count\_of\_elems \* 2];

tmp = head;

for (int i = 0; i < count\_of\_elems; ++i) {

all\_vertexes[2 \* i] = tmp->id\_from;

all\_vertexes[2 \* i + 1] = tmp->id\_to;

tmp = tmp->next;

}

for (int i = 0, tmp\_swap; i < count\_of\_elems \* 2; ++i) {

for (int j = i; j < count\_of\_elems \* 2; ++j) {

if (all\_vertexes[i] > all\_vertexes[j]) {

tmp\_swap = all\_vertexes[i];

all\_vertexes[i] = all\_vertexes[j];

all\_vertexes[j] = tmp\_swap;

}

}

}

/\*int count\_of\_unique = 0;

for (int i = 0; i < count\_of\_elems \* 2 - 1; ++i)

if (all\_vertexes[i] != all\_vertexes[i + 1])

++count\_of\_unique;

if (count\_of\_unique)

++count\_of\_unique;\*/

if (count\_of\_unique) {

int\*\* weight\_matrix = new int\* [count\_of\_unique];

for (int i = 0; i < count\_of\_unique; ++i)

weight\_matrix[i] = new int[count\_of\_unique];

for (int i = 0; i < count\_of\_unique; ++i)

for (int j = 0; j < count\_of\_unique; ++j)

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

array\_of\_unique[0] = all\_vertexes[0];

for (int i = 0, j = 0; j < count\_of\_unique - 1; ++i) {

if (all\_vertexes[i] != all\_vertexes[i + 1])

array\_of\_unique[++j] = all\_vertexes[i + 1];

}

}

return array\_of\_unique;

}

// возвращает кол-во вершин графа

int get\_count\_of\_unique() {

\_Edge\* tmp = head;

int count\_of\_elems = 0;

while (tmp) {

tmp = tmp->next;

++count\_of\_elems;

}

int\* all\_vertexes = new int[count\_of\_elems \* 2];

tmp = head;

for (int i = 0; i < count\_of\_elems; ++i) {

all\_vertexes[2 \* i] = tmp->id\_from;

all\_vertexes[2 \* i + 1] = tmp->id\_to;

tmp = tmp->next;

}

for (int i = 0, tmp\_swap; i < count\_of\_elems \* 2; ++i) {

for (int j = i; j < count\_of\_elems \* 2; ++j) {

if (all\_vertexes[i] > all\_vertexes[j]) {

tmp\_swap = all\_vertexes[i];

all\_vertexes[i] = all\_vertexes[j];

all\_vertexes[j] = tmp\_swap;

}

}

}

int count\_of\_unique = 0;

for (int i = 0; i < count\_of\_elems \* 2 - 1; ++i)

if (all\_vertexes[i] != all\_vertexes[i + 1])

++count\_of\_unique;

if (count\_of\_unique)

++count\_of\_unique;

return count\_of\_unique;

}

// поиск в глубину в графе

void dfs(int source, bool\* used, int\*\* matrix, int size) {

used[source] = 1;

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

if (used[i] == 0 && (matrix[source][i]/\* || matrix[i][source]\*/))

dfs(i, used, matrix, size);

}

}

// возвращает кол-во компонент связности графа

int get\_connectivity\_component\_count() {

int count = 0;

int size = this->get\_count\_of\_unique();

int\*\* matrix = this->get\_weight\_matrix();

bool\* used = new bool[size];

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

used[i] = 0;

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

if (used[i] == 0) {

this->dfs(i, used, matrix, size);

++count;

}

}

return count;

}

// возвращает матрицу кратчайших расстояний

int\*\* get\_minimal\_ways() {

int\*\* weight\_matrix = this->get\_weight\_matrix();

int\* array\_of\_unique = this->get\_array\_of\_unique();

int count\_of\_unique = this->get\_count\_of\_unique();

int\*\* distances = new int\*[count\_of\_unique];

if (count\_of\_unique)

for (int k = 0; k < count\_of\_unique; ++k) {

int\* distance = new int[count\_of\_unique];

bool\* visited = new bool[count\_of\_unique];

for (int i = 0; i < count\_of\_unique; ++i) {

distance[i] = 1000000;

visited[i] = false;

}

distance[k] = 0;

for (int count = 0, index, min = 1000000; count < count\_of\_unique - 1; ++count, min = 1000000) {

for (int i = 0; i < count\_of\_unique; ++i)

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

min = distance[i];

index = i;

}

visited[index] = true;

for (int i = 0; i < count\_of\_unique; ++i)

if (!visited[i] && weight\_matrix[index][i] && distance[index] != 1000000 && distance[index] + weight\_matrix[index][i] < distance[i])

distance[i] = distance[index] + weight\_matrix[index][i];

}

distances[k] = distance;

}

else

distances = nullptr;

return distances;

}

// возвращает множество инциденций минимального остова

int\*\* get\_minimal\_ostov() {

int\*\* weight\_matrix = this->get\_weight\_matrix();

int\* array\_of\_unique = this->get\_array\_of\_unique();

int count\_of\_unique = this->get\_count\_of\_unique();

if (!count\_of\_unique) return nullptr;

bool\* visited = new bool[count\_of\_unique];

for (int i = 0; i < count\_of\_unique; ++i)

visited[i] = false;

visited[0] = true;

int\*\* minimal\_ostov = new int\* [count\_of\_unique - 1];

for (int i = 0; i < count\_of\_unique - 1; ++i)

minimal\_ostov[i] = new int[3];

for (int l = 0; l < count\_of\_unique - 1; ++l) {

int min\_target = -1, min\_source = -1;

for (int i = 0; i < count\_of\_unique; ++i)

if (visited[i])

for (int j = 0; j < count\_of\_unique; ++j)

if (!visited[j] && weight\_matrix[i][j] > 0 && (min\_source == -1 || weight\_matrix[i][j] < weight\_matrix[min\_source][min\_target]))

min\_source = i, min\_target = j;

visited[min\_target] = true;

minimal\_ostov[l][0] = min\_source;

minimal\_ostov[l][1] = min\_target;

minimal\_ostov[l][2] = weight\_matrix[min\_source][min\_target];

}

return minimal\_ostov;

}

// возвращает кол-во рёбер

int get\_count\_of\_edges() {

int count = 0;

\_Edge\* tmp = this->head;

while (tmp) {

++count;

tmp = tmp->next;

}

return count;

}

};

**Graph\_WF.h**

//объект, представляющий граф

\_Graph\* graph;

//создание объекта graph

graph = new \_Graph();

// запрещает пользователю вводить что-либо, кроме цифр

private: System::Void only\_num\_KeyPress(System::Object^ sender, System::Windows::Forms::KeyPressEventArgs^ e) {

if (!Char::IsDigit(e->KeyChar) && e->KeyChar != 0x08)

e->Handled = true;

}

// запрещает пользователю вставлять информацию через комбинацию клавиш Ctrl+V

private: System::Void deny\_paste\_KeyDown(System::Object^ sender, System::Windows::Forms::KeyEventArgs^ e) {

((TextBox^)sender)->ContextMenu = nullptr;

((TextBox^)sender)->ContextMenuStrip = nullptr;

}

//вспомогательная функция для событий add\_edge\_Click и delete\_edge\_Click. обновляет нек-рые информационные поля

private: void info\_update() {

String^ info, ^ weight\_matrix, ^ minimal\_ways, ^ minimal\_ostov;

info += "Кол-во вершин: " + graph->get\_count\_of\_unique().ToString() + "\r\n";

info += "Кол-во дуг: " + graph->get\_count\_of\_edges().ToString() + "\r\n";

int\*\* weights = graph->get\_weight\_matrix();

int count\_of\_unique = graph->get\_count\_of\_unique();

int\* array\_of\_unique = graph->get\_array\_of\_unique();

int\*\* distances = graph->get\_minimal\_ways();

if (graph->get\_connectivity\_component\_count() == 1) {

int\*\* min\_ostov = graph->get\_minimal\_ostov();

int min\_ostov\_value = 0;

for (int i = 0; i < count\_of\_unique - 1; ++i) {

minimal\_ostov += (array\_of\_unique[min\_ostov[i][0]]).ToString() + " -> " + (array\_of\_unique[min\_ostov[i][1]]).ToString() + ", ";// << " = " << min\_ostov[i][2];

min\_ostov\_value += min\_ostov[i][2];

}

minimal\_ostov += "\r\nВес мин остова: " + min\_ostov\_value.ToString();

}

else {

minimal\_ostov = "Вес мин остова определить нельзя, ибо компонента связности не одна!";

}

weight\_matrix += " ";

minimal\_ways += " ";

for (int i = 0; i < count\_of\_unique; ++i) {

weight\_matrix += array\_of\_unique[i].ToString() + " ";

minimal\_ways += array\_of\_unique[i].ToString() + " ";

}

weight\_matrix += "\r\n";

minimal\_ways += "\r\n";

for (int i = 0; i < count\_of\_unique; ++i, weight\_matrix += "\r\n", minimal\_ways += "\r\n") {

weight\_matrix += array\_of\_unique[i].ToString() + " ";

minimal\_ways += array\_of\_unique[i].ToString() + " ";

for (int j = 0; j < count\_of\_unique; ++j) {

weight\_matrix += weights[i][j].ToString() + " ";

distances[i][j] == 1000000 ?

minimal\_ways += "0 " :

minimal\_ways += distances[i][j].ToString() + " ";

}

}

info += "Кол-во компонент связности: " + graph->get\_connectivity\_component\_count().ToString();

info\_textBox->Text = info;

weight\_matrix\_textBox->Text = weight\_matrix;

minimal\_ways\_textBox->Text = minimal\_ways;

min\_ostov\_textBox->Text = minimal\_ostov;

}

// для добавления дуги в граф

private: System::Void add\_edge\_Click(System::Object^ sender, System::EventArgs^ e) {

String^ cpy, ^log;

if (!source\_textBox->Text->Length || !target\_textBox->Text->Length || !weight\_textBox->Text->Length) {

cpy = log\_textBox->Text;

log\_textBox->Text = "";

log\_textBox->Text += "Отказ! (пустые поля)\r\n";

log\_textBox->Text += cpy;

}

else if (source\_textBox->Text == target\_textBox->Text) {

cpy = log\_textBox->Text;

log\_textBox->Text = "";

log\_textBox->Text += "Отказ! (петля)\r\n";

log\_textBox->Text += cpy;

}

else if (weight\_textBox->Text == "0") {

cpy = log\_textBox->Text;

log\_textBox->Text = "";

log\_textBox->Text += "Отказ! (нулевой вес)\r\n";

log\_textBox->Text += cpy;

}

else {

String^ id\_from = source\_textBox->Text;

String^ id\_to = target\_textBox->Text;

String^ \_weight = weight\_textBox->Text;

int source, target, weight;

source = source.Parse(id\_from);

target = target.Parse(id\_to);

weight = weight.Parse(\_weight);

int res = graph->Add(source, target, weight);

log = "Дуга " + id\_from + " - > " + id\_to + " с весом " + \_weight;

switch (res) {

case 0:

log += " перезаписана!\r\n"; break;

case 1:

log += " не перезаписана!\r\n"; break;

case 2:

log += " добавлена!\r\n"; break;

}

info\_update();

cpy = log\_textBox->Text;

log\_textBox->Text = "";

log\_textBox->Text += log;

log\_textBox->Text += cpy;

}

}

// для удаления дуги из графа

private: System::Void delete\_edge\_Click(System::Object^ sender, System::EventArgs^ e) {

String^ cpy, ^ log;

if (!source\_textBox->Text->Length || !target\_textBox->Text->Length || !weight\_textBox->Text->Length) {

cpy = log\_textBox->Text;

log\_textBox->Text = "";

log\_textBox->Text += "Отказ! (пустые поля)\r\n";

log\_textBox->Text += cpy;

}

else if (source\_textBox->Text == target\_textBox->Text) {

cpy = log\_textBox->Text;

log\_textBox->Text = "";

log\_textBox->Text += "Отказ! (петля)\r\n";

log\_textBox->Text += cpy;

}

else {

String^ id\_from = source\_textBox->Text;

String^ id\_to = target\_textBox->Text;

int source, target;

source = source.Parse(id\_from);

target = target.Parse(id\_to);

int res = graph->Pop(source, target);

log = "Дуга " + id\_from + " - > " + id\_to;

switch (res) {

case 0:

log += " не удалёна, ведь граф уже пуст!\r\n"; break;

case 1:

log += " удалёна!\r\n"; break;

case 2:

log += " не найдена!\r\n"; break;

}

info\_update();

cpy = log\_textBox->Text;

log\_textBox->Text = "";

log\_textBox->Text += log;

log\_textBox->Text += cpy;

}

}

// для очистки графа от дуг

private: System::Void clear\_graph\_button\_Click(System::Object^ sender, System::EventArgs^ e) {

graph->Clear\_graph();

info\_textBox->Text = "";

log\_textBox->Text = "";

weight\_matrix\_textBox->Text = "";

minimal\_ways\_textBox->Text = "";

min\_ostov\_textBox->Text = "";

source\_textBox->Text = "";

target\_textBox->Text = "";

weight\_textBox->Text = "";

}

**ВИСНОВКИ**

В ході виконання курсової роботи було розроблено програму для програму для обробки графів з графічним інтерфейсом під популярну платформу Windows. При розробці було використано об’єктно-орієнтований підхід.

До переваг ООП можна віднести: можливість легкої модифікації (при грамотному аналізі і проектуванні), можливість відкату при наявності версій, більш легка розширюваність, «більш природна» декомпозиція програмного забезпечення, яка істотно полегшує його розробку, скорочення кількості міжмодульних викликів і зменшення обсягів інформації, що передається між модулями, збільшується показник повторного використання коду.

Був розроблений графічний інтерфейс та алгоритми які реалізують необхідні процеси. Також були вивчені базові поняття та синтаксис мови програмування С++ та його діалекту С++/CLR , який використовується у Microsoft Windows Forms, базові поняття та парадигми ООП.

**ПЕРЕЛІК ПОСИЛАНЬ**

1. Методичні вказівки до курсової роботи з дисципліни «Технології об’єктно-орієнтованого програмування» для студентів усіх форм навчання першого (бакалаврського) рівня вищої освіти спеціальності 122 – «Комп’ютерні науки» освітньої програми «Інформаційні технології управління» / Упоряд. Білова Т.Г. – Харків: ХНУРЕ, 2017. –33 с.

2. Бондаренко М.Ф. Комп’ютерна дискретна математика / М. Ф. Бондаренко, Н. В. Білоус, А. Г. Руткас. – Харків: Компанія СМІТ, 2008. – 480 с.

**ДОДАТОК А**

**ПРИКЛАД ВИКОНАННЯ ПРОГРАМИ**

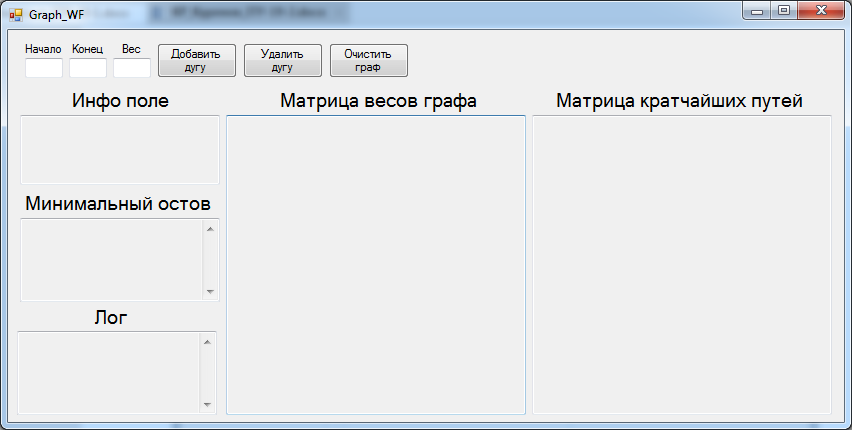
****

Рисунок А.1 – Демонстрація вікна початку роботи програми

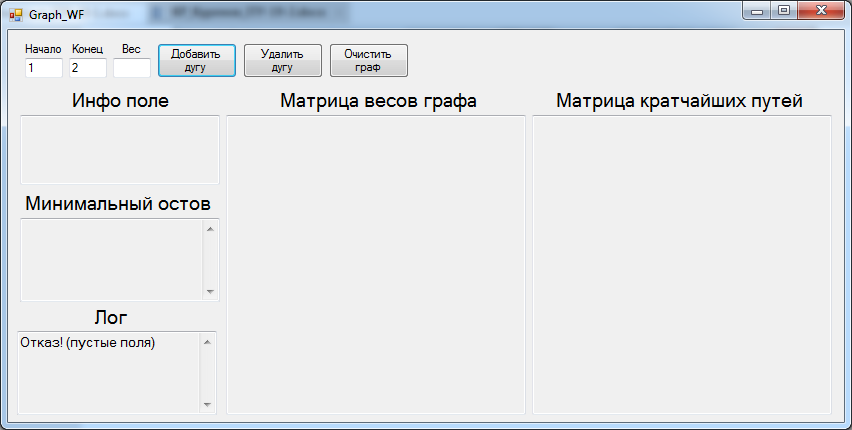


Рисунок А.2 – Демонстрація обробки некоректних дій (спроба додати дугу, незаповнивши необхідні поля)

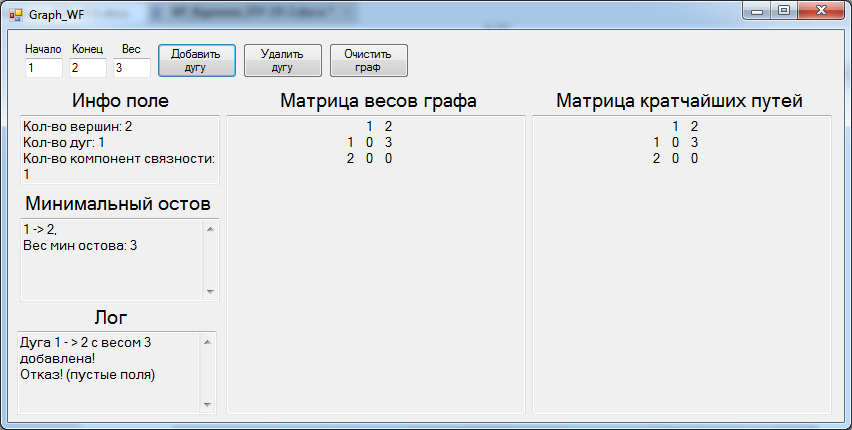
****

Рисунок А.3 – Демонстрація додавання дуги до графу

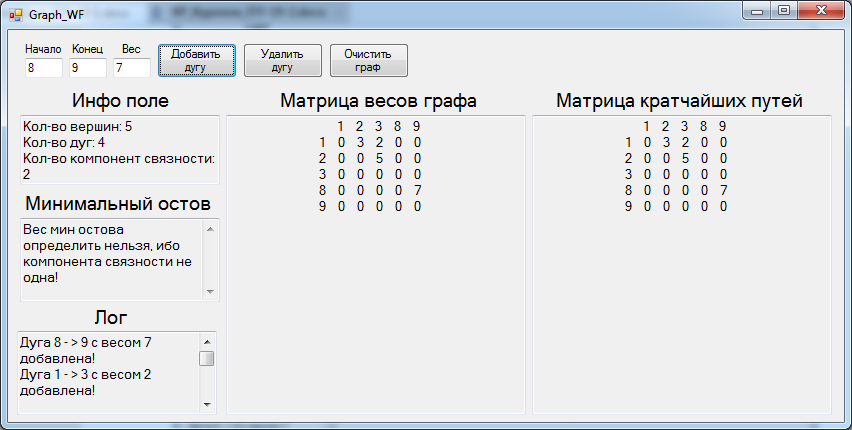


Рисунок А.4 – Демонстрація ситуації, коли кількость компонент зв’язності більше однієї

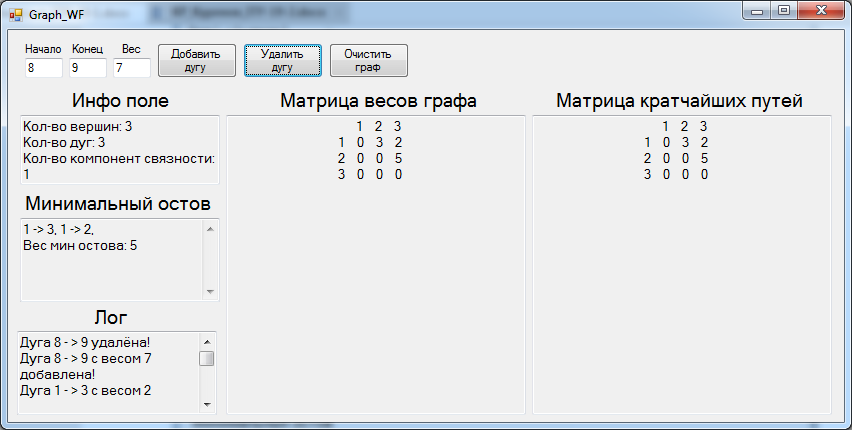
****

Рисунок А.5 – Демонстрація видалення дуги із графу

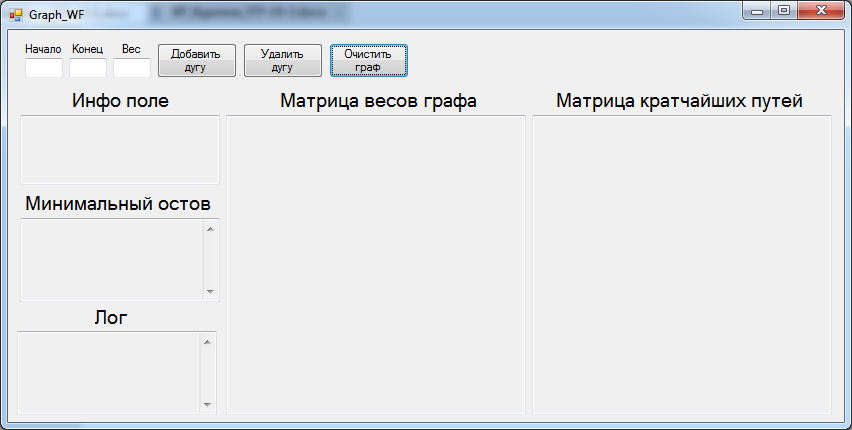
****

Рисунок А.6 – Демонстрація вікна програми після очищення графу за допомогою кнопки «Очистить граф»

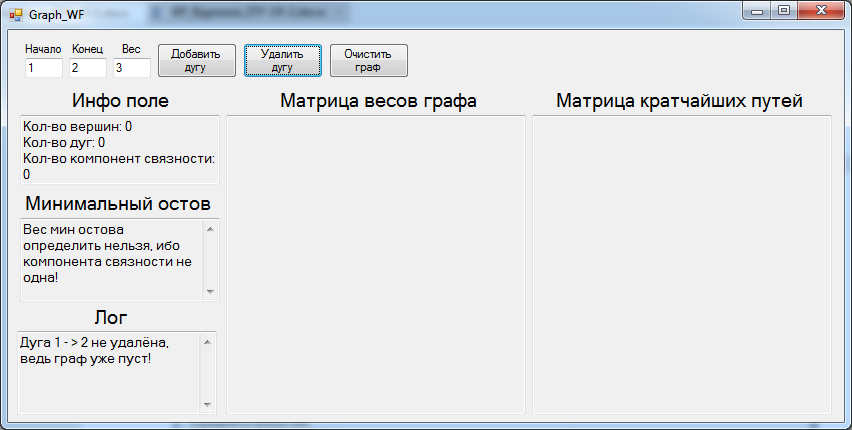
****

Рисунок А.7 – Демонстрація обробки некоректних дій (спроба видалити неіснуючу дугу)

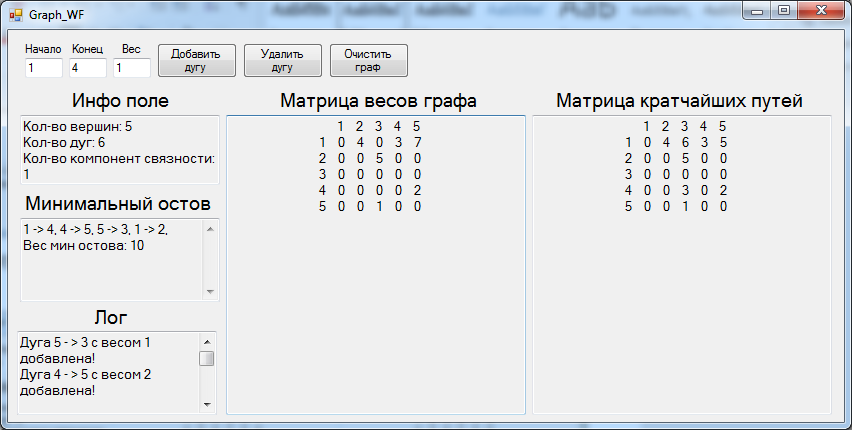


Рисунок А.8 – Демонстрація роботи програми на графі з більшим числом дуг

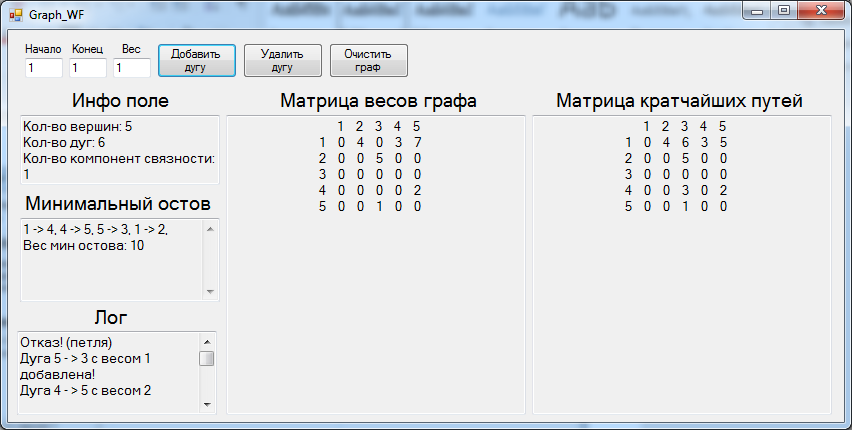


Рисунок А.9 – Демонстрація спроби додати петлю до графу

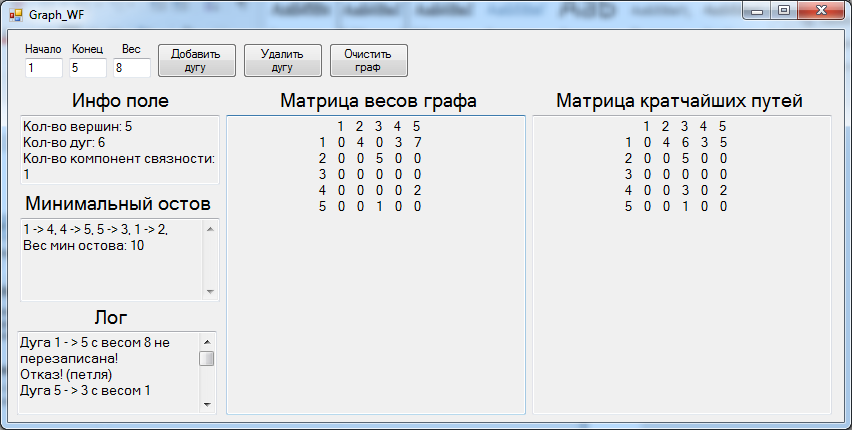


Рисунок А.10 – Демонстрація спроби перезаписати дугу з новою (більшою) вагою

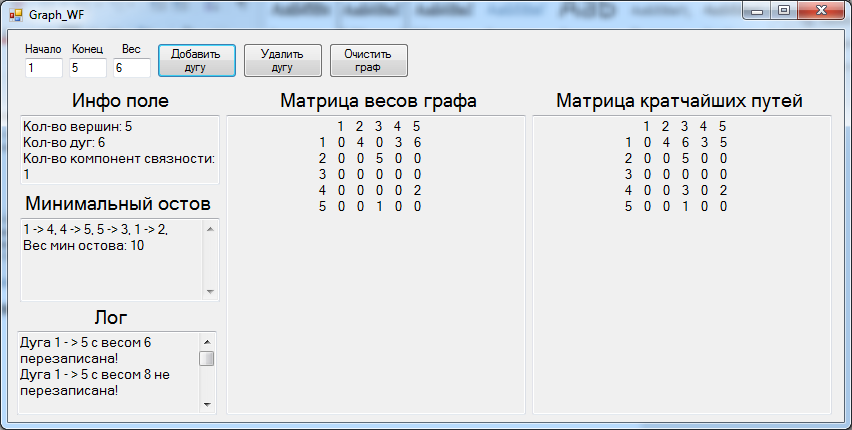


Рисунок А.11 – Демонстрація спроби перезаписати дугу з новою (меншою) вагою

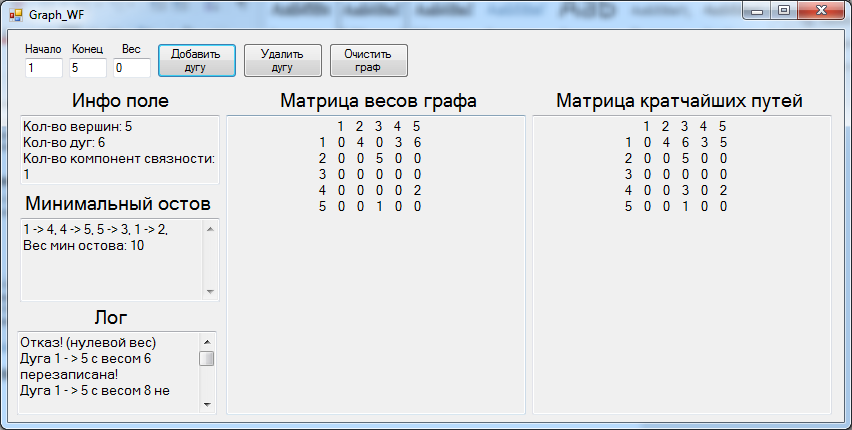


Рисунок А.12 – Демонстрація спроби додати дугу нульової ваги