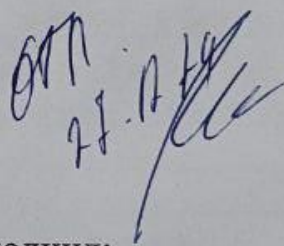


Министерство науки и высшего образования Российской Федерации  
Пензенский государственный университет  
Кафедра «Вычислительная техника»

## ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе  
по курсу «Логика и основы алгоритмизации  
в инженерных задачах»  
на тему «Реализация алгоритма нахождения  
Эйлеровых циклов»



Выполнил:

студент группы 23ВВВ2

Пырков Д. А.

Принял:

к. т. н. доцент

Юрова О. В.

к. т. н. проф.

Митрохин М. А.

Пенза 2024

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет Вычислительной техники

Кафедра "Вычислительная техника"

"УТВЕРЖДАЮ"

Зав. кафедрой ВТ

« \_\_\_\_ » \_\_\_\_ 20 \_\_\_\_

ЗАДАНИЕ

на курсовое проектирование по курсу

"Логика и основы алгоритмизации в инженерных задачах".  
Студенту Пурмову Денису Алексеевичу Группа 23ВВВ2  
Тема проекта Реализация алгоритма нахождения  
Эйлеровых циклов

Исходные данные (технические требования) на проектирование

Разработка алгоритмов и программного обеспечения  
в соответствии с данным заданием курсового  
проекта. Настоятельная записка должна содержать:

1. Постановку задачи;
2. Теоретическую часть задания;
3. Описание алгоритма поставленной задачи;
4. Пример ручного расчёта задачи и вычислений  
(на небольшом участке работы алгоритма);
5. Описание самой программы;
6. Тесты;
7. Список литературы;
8. Листинг программы;
9. Результаты работы программы.



## Объем работы по курсу

### 1. Расчетная часть

Точный расчет работы алгоритма.

### 2. Графическая часть

Схема алгоритма в формате блок-схем.

### 3. Экспериментальная часть

Тестирование программы;

Результаты работы программы на тестовых данных.

### Срок выполнения проекта по разделам

- 1 Исследование теоретической части курсового
- 2 Разработка алгоритмов программы
- 3 Разработка программы
- 4 Тестирование и завершение разработки программы
- 5 Оформление пояснительной записки
- 6
- 7
- 8

Дата выдачи задания "06" сентября

Дата защиты проекта " " "

Руководитель Митрохин М.А.

Задание получил "06" сентября 2024 г.

Студент Жуков Денис Алексеевич Жуков

# Содержание

<b>Введение.....</b>	<b>5</b>
<b>1. Постановка задачи.....</b>	<b>6</b>
<b>2. Теоретическая часть задания.....</b>	<b>7</b>
<b>3. Описание программы .....</b>	<b>8</b>
<b>3.1 Общая структура программы .....</b>	<b>8</b>
<b>3.2 Описание алгоритма поиска Эйлера цикла.....</b>	<b>9</b>
<b>4. Руководство пользователя .....</b>	<b>12</b>
<b>5. Тестирование .....</b>	<b>16</b>
<b>6. Ручной расчёт задачи .....</b>	<b>22</b>
<b>Заключение .....</b>	<b>24</b>
<b>Список используемых источников .....</b>	<b>25</b>
<b>Приложение А.....</b>	<b>26</b>

## Введение

Эйлеров цикл в графах — это фундаментальное понятие теории графов. Эйлеров цикл представляет собой маршрут, который проходит через каждое ребро графа ровно один раз и возвращается в исходную вершину. Эта концепция находит широкое применение в различных областях, включая проектирование маршрутов, оптимизацию сетей и решение логистических задач.

Нахождение эйлеровых циклов является классической алгоритмической задачей. В отличие от поиска гамильтоновых циклов, для эйлеровых циклов существуют эффективные алгоритмы решения, работающие за полиномиальное время. Наиболее известным является алгоритм Флёрри, а также алгоритм Хиерхольцера, которые позволяют найти эйлеров цикл в графе, если он существует.

Эйлеровы циклы имеют особое значение в практических приложениях, например, при проектировании печатных плат, планировании маршрутов уборочной техники или оптимизации движения транспорта. В отличие от простых алгоритмов обхода графа, алгоритмы поиска эйлеровых циклов направлены на нахождение специального маршрута, охватывающего все рёбра графа ровно по одному разу.

В качестве среды разработки была выбрана среда Microsoft Visual Studio 2022, язык программирования – Си.

Целью данной курсовой работы является разработка программы на языке Си, который обладает необходимой эффективностью и гибкостью. Именно с его помощью в данном курсовом проекте реализуется алгоритм поиска эйлеровых циклов в графах.

## **1. Постановка задачи**

Требуется разработать программу, которая найдет эйлеров цикл в графе.

Исходный граф в программе должен задаваться матрицей смежности, причем должна быть возможность задать ее различными способами, как случайно, так и из файла. Пользователь должен вводить в программу либо количество вершин, либо название файла, где записана матрица смежности и название файла, где сохраняются результаты. Если эйлеров цикл не будет существовать, программа покажет сообщение об этом и предложит сделать граф эйлеровым. После обработки этих данных на экран выводится матрица смежности и результат алгоритма поиска эйлеровых циклов. Необходимо предусмотреть различные исходы поиска, чтобы программа не выдавала ошибок и работала правильно. Устройство ввода – клавиатура и мышь. Необходимо меню для удобной работы пользователя с алгоритмом.

Минимальные системные требования для запуска программы:

### **1. Операционная система:**

- Windows 7 SP1 или новее;
- Windows Server 2012 R2 или новее.

### **2. Процессор:**

- 1.8 GHz или быстрее;
- Поддержка x86 или x64 архитектуры.

### **3. Оперативная память (RAM):**

- Минимум 512 MB;
- Рекомендуется 1 GB.

### **4. Дисковое пространство:**

- Около 50-100 MB для самого приложения;
- Дополнительное место для данных программы.

## 2. Теоретическая часть задания

Эйлеров цикл в графе представляет собой путь, который проходит через каждое ребро графа ровно один раз и возвращается в начальную вершину. Граф  $G$  также задается множеством вершин и ребер, но ключевое внимание уделяется именно ребрам.

При представлении графа матрицей смежности для поиска Эйлерова цикла важно учитывать степени вершин - количество ребер, инцидентных каждой вершине.

Основное условие существования Эйлерова цикла:

- Для неориентированного графа: все вершины должны иметь четную степень;
- Для ориентированного графа: количество входящих ребер должно равняться количеству исходящих для каждой вершины.

Алгоритм поиска Эйлерова цикла (алгоритм Флёрри) работает так:

1. Начинаем с произвольной вершины;
2. Идем по любому не пройденному ребру;
3. Удаляем пройденное ребро из графа;
4. Повторяем процесс, пока не пройдем все ребра.

В отличие от поиска Гамильтонова цикла, нахождение Эйлерова цикла является полиномиальной задачей, то есть может быть решена достаточно быстро даже для больших графов.

Граф называется эйлеровым, если в нем существует эйлеров цикл, и полуэйлеровым, если в нем существует эйлеров путь (проходящий по всем ребрам, но не обязательно возвращающийся в начальную вершину).

### 3. Описание программы

Для написания программы был использован язык программирования С. Язык программирования С - универсальный язык программирования, который пользуется особой популярностью, благодаря сочетанию возможностей языков программирования высокого и низкого уровней.

Проект был создан в виде консольного приложения C++.

#### 3.1 Общая структура программы

Функция `main()` представляет собой основной цикл программы для работы с графами и поиска эйлеровых циклов. Она работает следующим образом:

1. Инициализация:
  - Устанавливается русская локализация;
  - Объявляются переменные для размера графа, выбора пользователя и вероятности создания ребра.
2. Основной цикл (`while(1)`):
  - Очищает экран;
  - Выводит меню с 4 опциями:
    1. Создание случайного неориентированного графа;
    2. Создание случайного ориентированного графа;
    3. Загрузка графа из файла;
    4. Выход.
3. Для каждого варианта (`case 1-3`):
  - Запрашивает необходимые параметры (размер графа, вероятность рёбер или имя файла);
  - Создает или загружает граф
  - Проверяет, является ли граф эйлеровым;
  - Предлагает сделать граф эйлеровым, если он таковым не является;
  - Ищет эйлеров цикл;
  - Сохраняет результаты в файл;



- Выводит результаты на экран.

Особенности для разных случаев:

- Case 1: работа с неориентированным графом;
- Case 2: работа с ориентированным графом;
- Case 3: загрузка графа из файла и определение его типа.

4. После каждой операции:

- Освобождает память;
- Ждёт нажатия клавиши Esc для возврата в меню;
- Очищает экран.

5. Защита от ошибок:

- Проверяет корректность ввода;
- Обрабатывает ошибки открытия файлов.

6. Выход:

- Программа завершается при выборе опции 4;
- При некорректном вводе возвращает в меню.

### 3.2 Описание алгоритма поиска Эйлера цикла

Алгоритм поиска Эйлера цикла работает следующим образом:

1. Функция `findEulerianCycle` инициирует поиск Эйлера цикла. Она выполняет следующие задачи:

- Проверка существования Эйлера цикла через функцию `isEulerian`;
- Создание копии исходного графа (`temp_graph`), чтобы не модифицировать оригинал;
- Вызов функции `findEulerPath` для поиска цикла.

2. Основной алгоритм реализован в функции `findEulerPath`. Он использует модифицированный алгоритм Флёрри и работает следующим образом:

```
void findEulerPath(int_fast8_t** graph_pointer, int size, int
start, Path* result, bool directed) {
    short stack[2048];    // Стек для хранения вершин
```

```

int top = -1;          // Указатель на вершину стека
stack[++top] = start; // Помещаем начальную вершину в стек
while (top >= 0) {     // Пока стек не пуст
    int current = stack[top]; // Текущая вершина
    bool found = false;     // Флаг наличия непосещенного ребра
    // Ищем непосещенное ребро из текущей вершины
    for (int i = 0; i < size; i++) {
        if (graph_pointer[current][i] > 0) {
            stack[++top] = i; // Добавляем вершину в стек
            graph_pointer[current][i]--; // Удаляем ребро
            if (!directed) {
                graph_pointer[i][current]--; } // Для
неориентированного графа
            found = true;
            break;
        }
    }
    // Если нет непосещенных ребер
    if (!found) {
        result->path[result->size++] = stack[top]; //
Добавляем вершину в результат
        top--; // Удаляем вершину из стека
    }
}
}

```

Алгоритм работает следующим образом:

1. Использует стек для отслеживания текущего пути;
2. Начинает с заданной вершины start;
3. Для каждой текущей вершины:
  - Ищет непосещенное ребро ( $\text{graph\_pointer}[\text{current}][i] > 0$ );
  - Если находит:
    1. Добавляет новую вершину в стек;
    2. Удаляет использованное ребро из графа.
  - Если не находит:
    1. Добавляет вершину в результирующий путь;
    2. Возвращается назад по стеку.
4. В результате формируется путь в обратном порядке (от конца к началу).

Важные особенности:

- Алгоритм модифицирует граф, удаляя пройденные ребра;

- Для неориентированного графа удаляются ребра в обоих направлениях;
- Результирующий путь формируется в обратном порядке;
- Алгоритм гарантированно находит Эйлеров цикл, если он существует (что проверяется функцией `isEulerian`).

## 4. Руководство пользователя

Программа предназначена для поиска Эйлеровых циклов в графах. При запуске программы пользователю предоставляется меню с четырьмя опциями (Рисунок 1).

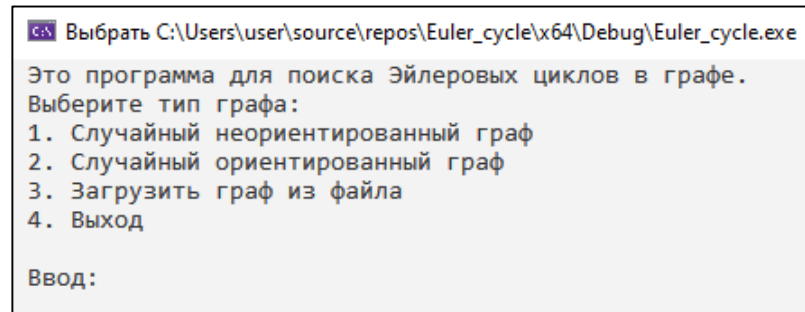


Рисунок 1 – Меню программы

Работа с неориентированным графом (опция 1): при выборе первой опции необходимо ввести количество вершин графа (число должно быть больше 1). Затем нужно указать вероятность создания ребра в процентах (от 0 до 100). После этого программа отобразит матрицу смежности созданного графа. Если граф не является Эйлеровым, программа предложит сделать его таковым (введите 1 для подтверждения или 0 для отказа). В конце потребуются ввести имя файла для сохранения результатов (Рисунок 2).

```
Выбрать C:\Users\user\source\repos\Euler_cycle\x64\Debug\Euler_cycle.exe

Это программа для поиска Эйлеровых циклов в графе.
Выберите тип графа:
1. Случайный неориентированный граф
2. Случайный ориентированный граф
3. Загрузить граф из файла
4. Выход

Ввод: 1
Введите количество вершин в неориентированном графе: 7
Введите вероятность создания ребра в неориентированном графе (от 0 до 100): 45
Матрица смежности неориентированного графа:

0 0 1 1 0 1 0
0 0 0 1 0 1 1
1 0 0 0 0 0 0
1 1 0 0 1 1 1
0 0 0 1 0 1 0
1 1 0 1 1 0 1
0 1 0 1 0 1 0

Граф не является Эйлеровым. Хотите сделать его Эйлеровым? (0 - нет, 1 - да): 1

Преобразованная матрица смежности:

0 0 1 0 0 1 0
0 0 0 1 0 0 1
1 0 0 0 0 0 1
0 1 0 0 1 1 1
0 0 0 1 0 1 0
1 0 0 1 1 0 1
0 1 1 1 0 1 0

Введите название файла для сохранения: 1.txt

Эйлеров цикл существует: 1-6 6-7 7-4 4-6 6-5 5-4 4-2 2-7 7-3 3-1

Нажмите Esc чтобы вернуться в меню.
```

Рисунок 2 – Работа с неориентированным графом

Работа с ориентированным графом (опция 2): процесс аналогичен работе с неориентированным графом, но создаётся ориентированный граф. Последовательность действий та же: ввод количества вершин, вероятности создания ребра, возможность сделать граф Эйлеровым и сохранение результатов в файл (Рисунок 3).



```
Выбрать C:\Users\user\source\repos\Euler_cycle\x64\Debug\Euler_cycle.exe

Это программа для поиска Эйлеровых циклов в графе.
Выберите тип графа:
1. Случайный неориентированный граф
2. Случайный ориентированный граф
3. Загрузить граф из файла
4. Выход

Ввод: 2
Введите количество вершин в ориентированном графе: 7
Введите вероятность создания ребра в ориентированном графе (от 0 до 100): 55
Матрица смежности ориентированного графа:

0 0 1 0 1 0 0
0 0 0 0 0 0 0
1 0 0 0 0 0 0
0 0 1 0 1 1 0
1 1 0 1 0 0 1
0 0 1 1 0 0 1
0 1 0 0 0 1 0

Граф не является Эйлеровым. Хотите сделать его Эйлеровым? (0 - нет, 1 - да): 1
Преобразованная матрица смежности:

0 0 1 0 1 0 0
0 0 0 1 1 0 0
1 0 0 0 1 1 0
0 0 1 0 1 1 0
1 1 0 1 0 0 1
0 0 1 1 0 0 1
0 1 0 0 0 1 0

Введите название файла для сохранения: 2.txt

Эйлеров цикл существует: 1-5 5-4 4-6 6-7 7-6 6-3 3-6 6-4 4-5 5-7 7-2 2-4 4-3 3-5 5-2 2-5 5-1 1-3 3-1

Нажмите Esc чтобы вернуться в меню.
```

Рисунок 3 – Работа с ориентированным графом

Загрузка графа из файла (опция 3): при выборе этой опции нужно указать имя файла, содержащего матрицу смежности графа. Программа определит тип графа (ориентированный или неориентированный) автоматически. Как и в предыдущих случаях, будет предложено сделать граф Эйлеровым при необходимости. Результаты сохраняются в указанный пользователем файл (Рисунок 4).

```
Выбрать C:\Users\user\source\repos\Euler_cycle\x64\Debug\Euler_cycle.exe
Это программа для поиска Эйлеровых циклов в графе.
Выберите тип графа:
1. Случайный неориентированный граф
2. Случайный ориентированный граф
3. Загрузить граф из файла
4. Выход

Ввод: 3
Введите название файла для чтения матрицы смежности: 1.txt

Матрица смежности графа из файла:

0 0 1 0 0 1 0
0 0 0 1 0 0 1
1 0 0 0 0 0 1
0 1 0 0 1 1 1
0 0 0 1 0 1 0
1 0 0 1 1 0 1
0 1 1 1 0 1 0

Введите название файла для сохранения результатов: 3.txt

Эйлеров цикл существует: 1-6 6-7 7-4 4-6 6-5 5-4 4-2 2-7 7-3 3-1

Нажмите Esc чтобы вернуться в меню.
```

**Рисунок 4 – Загрузка графа из файла**

Выход из программы (опция 4): завершает работу программы.

Дополнительная информация: после каждой операции для возврата в главное меню необходимо нажать клавишу Esc. При некорректном вводе данных программа выдаст соответствующее сообщение об ошибке. Результаты работы программы включают матрицу смежности графа и найденный Эйлеров цикл (если он существует).

## 5. Тестирование

В качестве среды разработки была выбрана программа Microsoft Visual Studio 2022. Программа обладает всеми средствами необходимыми при разработке и отладке программы.

Тестирование проводилось в рабочем порядке, в процессе разработки, после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем, связанных с вводом данных, изменением дизайна выводимых данных, алгоритмом программы, взаимодействием функций.

Для полного тестирования программы было проведено 8 тестов.

### Тест №1. Работа меню

Предусловие: программа запущена.

Тестирование: проверка отображения главного меню с четырьмя пунктами.

Ожидаемый результат: на экране отображается меню с опциями выбора типа графа и выхода.

Запускаем программу из Microsoft Visual Studio 2022 (Рисунок 5).

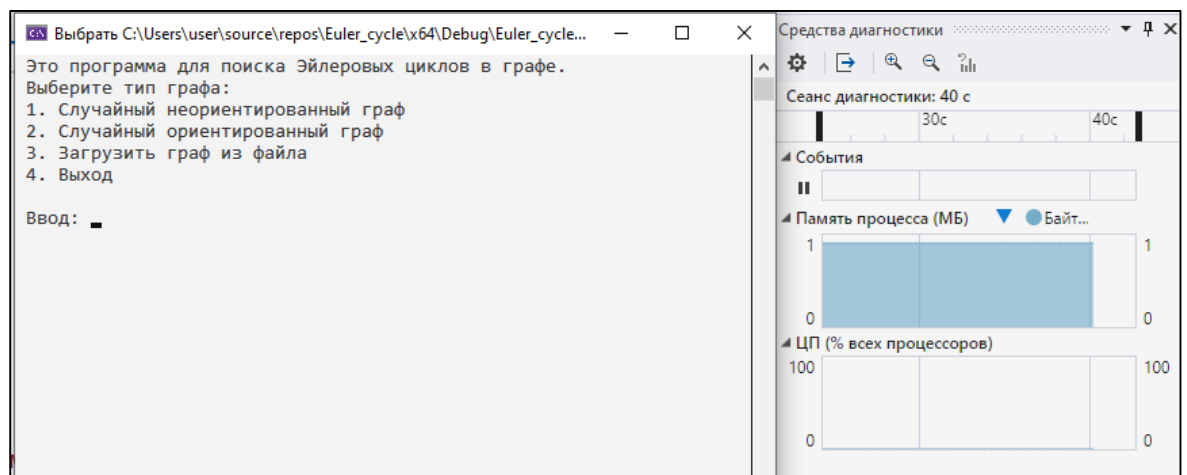


Рисунок 5 – Запуск программы

Программа успешно запустилась. Ожидаемые результаты совпали с наблюдаемыми.

### Тест №2. Выбор функции

Предусловие: отображено главное меню.

Тестирование: ввод числа от 1 до 4 для выбора соответствующей функции.

Ожидаемый результат: программа переходит к выбранной функции или завершает работу.

После каждого из четырёх запусков программы были введены числа от 1 до 4 соответственно (Рисунок 6).

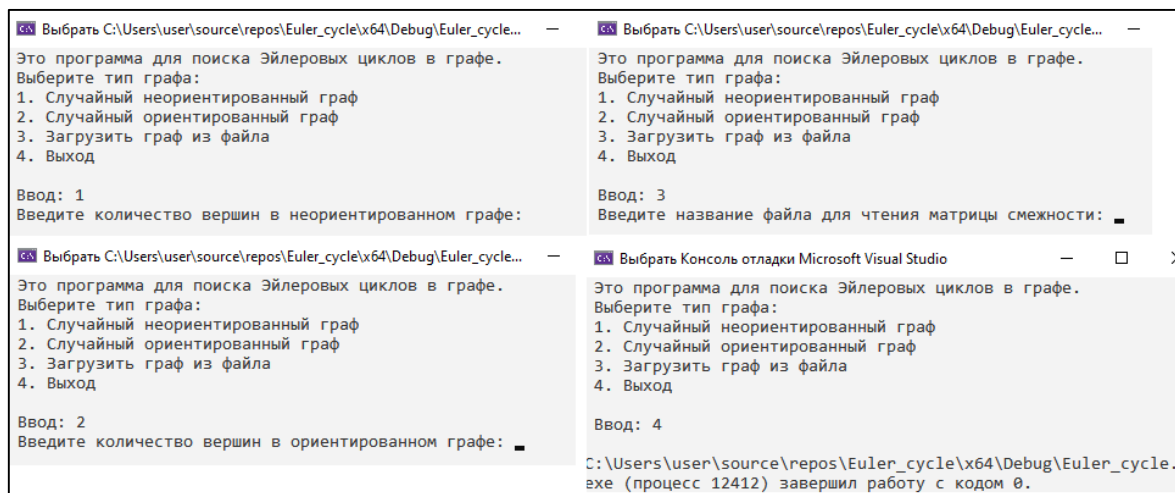


Рисунок 6 – Выбор всех функций программы

Программа переходит к выбранной функции и ожидает дальнейшего ввода. Ожидаемые результаты совпали с наблюдаемыми.

### Тест №3. Создание неориентированного графа

Предусловие: выбран пункт 1.

Тестирование: ввод количества вершин и вероятности создания ребра.

Ожидаемый результат: создается и отображается матрица смежности неориентированного графа.

После ввода 1 было введено количество вершин (7) и вероятность создания ребра в графе (Рисунок 7).

```
Выбрать C:\Users\user\source\repos\Euler_cycle\x64\Debug\Euler_cycle.exe
Это программа для поиска Эйлеровых циклов в графе.
Выберите тип графа:
1. Случайный неориентированный граф
2. Случайный ориентированный граф
3. Загрузить граф из файла
4. Выход

Ввод: 1
Введите количество вершин в неориентированном графе: 7
Введите вероятность создания ребра в неориентированном графе (от 0 до 100): 45
Матрица смежности неориентированного графа:

0 0 1 1 0 1 0
0 0 0 1 0 1 1
1 0 0 0 0 0 0
1 1 0 0 1 1 1
0 0 0 1 0 1 0
1 1 0 1 1 0 1
0 1 0 1 0 1 0
```

Рисунок 7 – Создание неориентированного графа

Программа корректно создала и отобразила матрицу смежности неориентированного графа. Ожидаемые результаты совпали с наблюдаемыми.

#### Тест №4. Создание ориентированного графа

Предусловие: выбран пункт 2.

Тестирование: ввод количества вершин и вероятности создания ребра.

Ожидаемый результат: создается и отображается матрица смежности ориентированного графа.

После ввода 2 было введено количество вершин (6) и вероятность создания ребра в графе (Рисунок 8).

```
Выбрать C:\Users\user\source\repos\Euler_cycle\x64\Debug\Euler_cycle.exe
Это программа для поиска Эйлеровых циклов в графе.
Выберите тип графа:
1. Случайный неориентированный граф
2. Случайный ориентированный граф
3. Загрузить граф из файла
4. Выход

Ввод: 2
Введите количество вершин в ориентированном графе: 6
Введите вероятность создания ребра в ориентированном графе (от 0 до 100): 55
Матрица смежности ориентированного графа:

0 1 0 0 0 0
1 0 1 1 0 0
1 1 0 0 0 1
1 1 1 0 0 0
0 0 1 1 0 1
1 1 1 1 1 0
```

Рисунок 8 – Создание ориентированного графа



Программа корректно создала и отобразила матрицу смежности ориентированного графа. Ожидаемые результаты совпали с наблюдаемыми.

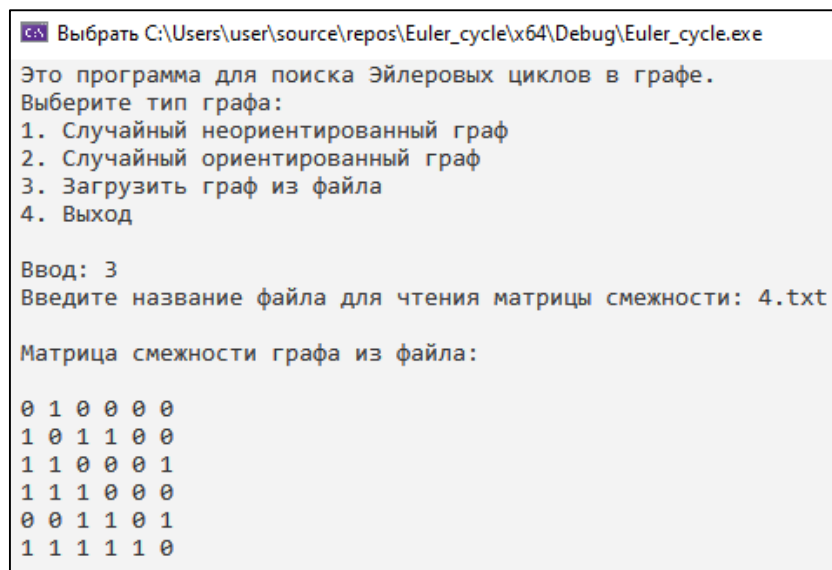
### **Тест №5. Загрузка графа из файла**

Предусловие: выбран пункт 3.

Тестирование: ввод имени существующего файла с матрицей смежности.

Ожидаемый результат: загружается и отображается матрица смежности из файла.

После ввода 3 было введено название файла (Рисунок 9).



```
Выбрать C:\Users\user\source\repos\Euler_cycle\x64\Debug\Euler_cycle.exe
Это программа для поиска Эйлеровых циклов в графе.
Выберите тип графа:
1. Случайный неориентированный граф
2. Случайный ориентированный граф
3. Загрузить граф из файла
4. Выход

Ввод: 3
Введите название файла для чтения матрицы смежности: 4.txt

Матрица смежности графа из файла:

0 1 0 0 0 0
1 0 1 1 0 0
1 1 0 0 0 1
1 1 1 0 0 0
0 0 1 1 0 1
1 1 1 1 1 0
```

**Рисунок 9 – Загрузка графа из файла**

Программа корректно считала и отобразила матрицу смежности графа из файла. Ожидаемые результаты совпали с наблюдаемыми.

### **Тест №6. Преобразование неэйлерова графа в эйлеровый**

Предусловие: граф создан и не является эйлеровым.

Тестирование: подтверждение преобразования графа в эйлеров.

Ожидаемый результат: отображается преобразованная матрица смежности эйлерова графа.

После создания графа любым способом введём 1, чтобы подтвердить преобразование (Рисунок 10).

```

Матрица смежности неориентированного графа:
0 0 1 1 1 1
0 0 1 0 1 0
1 1 0 0 0 1
1 0 0 0 1 0
1 1 0 1 0 1
1 0 1 0 1 0

Граф не является Эйлеровым. Хотите сделать его Эйлеровым? (0 - нет, 1 - да): 1

Преобразованная матрица смежности:
0 0 1 1 1 1
0 0 1 0 1 0
1 1 0 1 0 1
1 0 1 0 1 1
1 1 0 1 0 1
1 0 1 1 1 0

```

**Рисунок 10 – Преобразование неэйлерова графа в эйлеровый**

Программа корректно преобразовала неэйлеровый граф в эйлеровый и отобразила его матрицу смежности. Ожидаемые результаты совпали с наблюдаемыми.

### **Тест №7. Работа алгоритма**

Предусловие: граф создан или загружен.

Тестирование: автоматический поиск эйлерова цикла.

Ожидаемый результат: отображается найденный эйлеров цикл или сообщение о его отсутствии.

Создадим граф любым способом и введём название файла для сохранения матрицы смежности и эйлерова цикла (Рисунок 11).

```

Преобразованная матрица смежности:
0 0 1 1 1 1
0 0 1 0 1 0
1 1 0 1 0 1
1 0 1 0 1 1
1 1 0 1 0 1
1 0 1 1 1 0

Введите название файла для сохранения: 1.txt

Эйлеров цикл существует: 1-6 6-5 5-4 4-6 6-3 3-4 4-1 1-5 5-2 2-3 3-1

Нажмите Esc чтобы вернуться в меню.

```

**Рисунок 11 – Результат нахождения эйлерова цикла**

Программа корректно нашла и отобразила найденный эйлеров цикл. Ожидаемые результаты совпали с наблюдаемыми.

### **Тест №8. Сохранение в файл результата**

Предусловие: алгоритм выполнил поиск цикла.

Тестирование: ввод имени файла для сохранения результата.

Ожидаемый результат: матрица смежности и результат поиска сохраняются в указанный файл.

Создадим граф любым способом и введём название файла для сохранения матрицы смежности и эйлерова цикла (Рисунок 12).

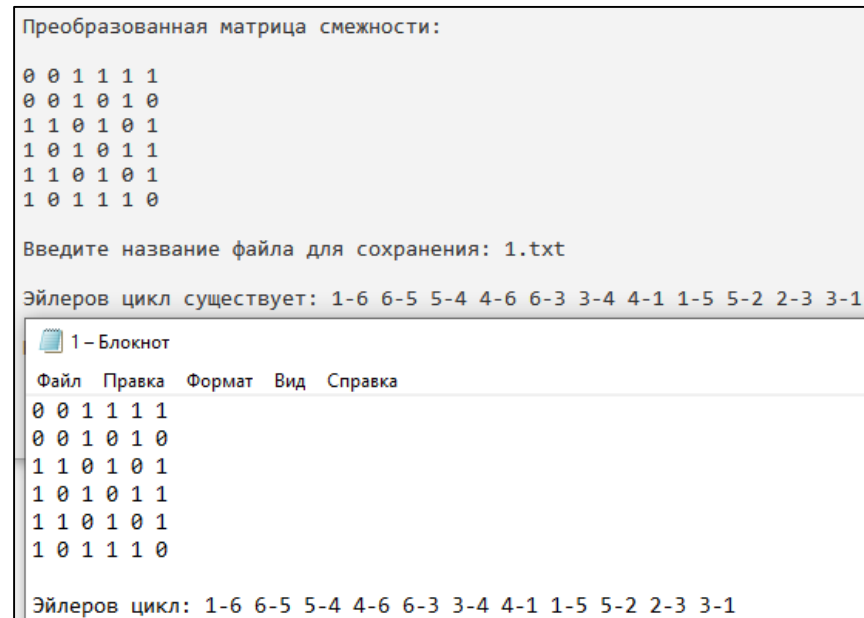


Рисунок 12 – Сохранение результатов в файл

Программа корректно записала в файл матрицу смежности и найденный эйлеров цикл. Ожидаемые результаты совпали с наблюдаемыми.

**Итоги тестирования:** все 8 тестов успешно пройдены, программа работает корректно.

## 6. Ручной расчёт задачи

Зададим матрицу смежности для графа с 7 вершинами и сохраним её в файле.

```
0 0 1 0 0 1 0
0 0 0 1 0 0 1
1 0 0 0 0 0 1
0 1 0 0 1 1 1
0 0 0 1 0 1 0
1 0 0 1 1 0 1
0 1 1 1 0 1 0
```

Проведём ручной расчет нахождения эйлерова цикла для данной матрицы смежности.

Сначала проверим степени вершин графа. Для существования эйлерова цикла все степени должны быть четными.

Подсчет степеней:

Вершина 1: 2 ребра

Вершина 2: 2 ребра

Вершина 3: 2 ребра

Вершина 4: 4 ребра

Вершина 5: 2 ребра

Вершина 6: 4 ребра

Вершина 7: 4 ребра

Так как граф является эйлеровым, начнем построение цикла с первой вершины:

1-6 (удаляем ребро)

6-7 (удаляем ребро)

7-4 (удаляем ребро)

4-6 (удаляем ребро)

6-5 (удаляем ребро)

5-4 (удаляем ребро)

4-2 (удаляем ребро)

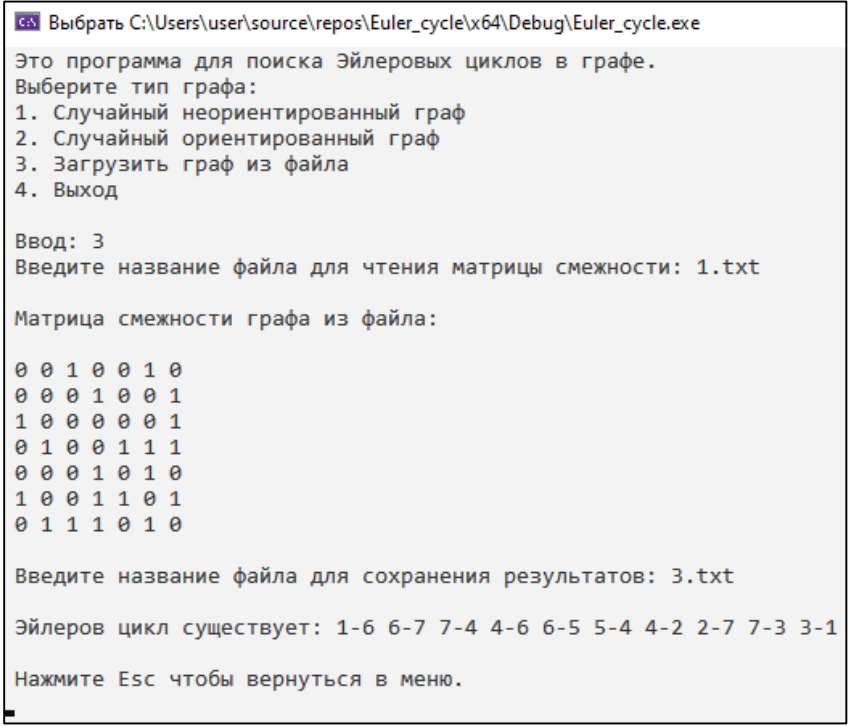
2-7 (удаляем ребро)

7-3 (удаляем ребро)

3-1 (удаляем ребро)

Полученный цикл: 1-6-7-4-6-5-4-2-7-3-1

Для проверки результатов используем функцию загрузки созданной матрицы смежности из файла (Рисунок 13).



```
Выбрать C:\Users\user\source\repos\Euler_cycle\x64\Debug\Euler_cycle.exe
Это программа для поиска Эйлеровых циклов в графе.
Выберите тип графа:
1. Случайный неориентированный граф
2. Случайный ориентированный граф
3. Загрузить граф из файла
4. Выход

Ввод: 3
Введите название файла для чтения матрицы смежности: 1.txt

Матрица смежности графа из файла:

0 0 1 0 0 1 0
0 0 0 1 0 0 1
1 0 0 0 0 0 1
0 1 0 0 1 1 1
0 0 0 1 0 1 0
1 0 0 1 1 0 1
0 1 1 1 0 1 0

Введите название файла для сохранения результатов: 3.txt

Эйлеров цикл существует: 1-6 6-7 7-4 4-6 6-5 5-4 4-2 2-7 7-3 3-1

Нажмите Esc чтобы вернуться в меню.
```

Рисунок 13 – Проверка программы на нахождение эйлерова цикла

Сравнивая с результатом работы программы, видим полное совпадение последовательности вершин в найденном эйлеровом цикле. Программа нашла тот же самый путь: 1-6-7-4-6-5-4-2-7-3-1.

Таким образом, ручной расчет подтвердил корректность работы программы для данного графа.



## **Заключение**

Таким образом, в процессе работы над данным проектом была разработана программа, реализующая алгоритм поиска эйлеровых циклов в графе в Microsoft Visual Studio 2022.

При выполнении данной курсовой работы были получены навыки разработки и тестирования программ и освоены методы работы с матрицами смежности графов. Углублены знания языка программирования Си.

Программа имеет небольшой, но достаточный для использования функционал возможностей.

### **Список используемых источников**

1. Р4. Уилсон Р. Введение в теорию графов. Пер. с англ. 1977. 208 с.
2. Прата С. Язык программирования C++. 2019 г.
3. Харви Дейтел, Пол Дейтел. Как программировать на C/C++. 2009 г.
4. Герберт Шилдт «Полный справочник по C++» - Вильямс, 2006 г.

# Приложение А

## Листинги программы

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <time.h>
#include <malloc.h>
#include <random>
#include <locale>
#include <Windows.h>

typedef struct {
    unsigned short path[2048];
    int size;
} Path;

int_fast8_t** create_graph(int size);
void initialize_unorient_graph(int_fast8_t** graph_pointer, int
size, float edge_probability);
void initialize_orient_graph(int_fast8_t** graph_pointer, int
size, float edge_probability);
void print_graph(int_fast8_t** graph_pointer, int size);
void free_graph(int_fast8_t** graph_pointer, int size);
bool findEulerianCycle(int_fast8_t** graph_pointer, int size, bool
directed, Path* result);
bool isEulerian(int_fast8_t** graph_pointer, int size, bool
directed);
void makeEulerian(int_fast8_t** graph_pointer, int size, bool
directed);

int main(void) {
    setlocale(LC_ALL, "Russian");
    int size, choice = 0;
    float edge_probability;
    char file_name[128];

    while (1) {
        system("cls");
        printf(" Это программа для поиска Эйлеровых циклов в
графе.\n");
        printf(" Выберите тип графа:\n");
        printf(" 1. Случайный неориентированный граф\n");
        printf(" 2. Случайный ориентированный граф\n");
        printf(" 3. Загрузить граф из файла\n");
        printf(" 4. Выход\n");
        printf("\n Ввод: ");
        scanf("%d", &choice);
        int_fast8_t** graph = NULL;

        switch (choice) {
            case 1:
            {
                printf(" Введите количество вершин в неориентированном
графе: ");
                if (!scanf("%d", &size) || size <= 1) {
```

```

        fprintf(stderr, " Ошибка ввода количества
вершин.");
        choice = 0;
        getchar();
        break;
    }

    graph = create_graph(size);
    fprintf(stdout, " Введите вероятность создания ребра в
неориентированном графе (от 0 до 100): ");
    if (!scanf("%f", &edge_probability) ||
edge_probability < 0 || edge_probability > 100) {
        fprintf(stderr, " Ошибка ввода вероятности.");
        choice = 0;
        getchar();
        break;
    }

    srand((unsigned int)time(NULL));
    initialize_unorient_graph(graph, size,
edge_probability);
    printf(" Матрица смежности неориентированного
графа:\n\n");
    print_graph(graph, size);

    int make_eulerian = 0;
    if (!isEulerian(graph, size, false)) {
        printf("\n Граф не является Эйлеровым. Хотите
сделать его Эйлеровым? (0 - нет, 1 - да): ");
        if (!scanf("%d", &make_eulerian) || make_eulerian
< 0 || make_eulerian > 1) {
            fprintf(stderr, " Неправильный ввод.");
            choice = 0;
            getchar();
            break;
        }
    }

    if (make_eulerian == 1) {
        makeEulerian(graph, size, false);
        printf("\n Преобразованная матрица
смежности:\n\n");
        print_graph(graph, size);
    }

    Path result1;
    printf("\n Введите название файла для сохранения: ");
    scanf("%s", file_name);

    FILE* file = fopen(file_name, "w");
    if (file == NULL) {
        printf("Ошибка открытия файла\n");
        break;
    }

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            fprintf(file, "%d ", graph[j][i]);
        }
        fprintf(file, "\n");
    }

```

```

    }

    if (findEulerianCycle(graph, size, false, &result1)) {
        fprintf(file, "\nЭйлеров цикл: ");
        for (int i = 1; i < result1.size; i++) {
            fprintf(file, "%d-%d ", result1.path[i - 1] +
1, result1.path[i] + 1);
        }
        fprintf(file, "\n");

        printf("\n Эйлеров цикл существует: ");
        for (int i = 1; i < result1.size; i++) {
            printf("%d-%d ", result1.path[i - 1] + 1,
result1.path[i] + 1);
        }
        printf("\n");
    }
    else {
        fprintf(file, "\nЭйлеров цикл не существует.\n");
        printf("\n Эйлеров цикл не существует.\n");
    }

    fclose(file);
    free_graph(graph, size);
    break;
}
case 2:
{
    printf(" Введите количество вершин в ориентированном
графе: ");

    if (!scanf(" %d", &size) || size <= 1) {
        fprintf(stderr, " Ошибка ввода количества
вершин.");

        choice = 0;
        getchar();
        break;
    }

    graph = create_graph(size);
    fprintf(stdout, " Введите вероятность создания ребра в
ориентированном графе (от 0 до 100): ");
    if (!scanf("%f", &edge_probability) ||
edge_probability < 0 || edge_probability > 100) {
        fprintf(stderr, " Ошибка ввода вероятности.");
        choice = 0;
        getchar();
        break;
    }

    srand((unsigned int)time(NULL));
    initialize_orient_graph(graph, size,
edge_probability);
    printf(" Матрица смежности ориентированного
графа:\n\n");

    print_graph(graph, size);

    int make_eulerian = 0;
    if (!isEulerian(graph, size, true)) {
        printf("\n Граф не является Эйлеровым. Хотите
сделать его Эйлеровым? (0 - нет, 1 - да): ");

```



```

        if (!scanf("%d", &make_eulerian) || make_eulerian
< 0 || make_eulerian > 1) {
            fprintf(stderr, " Неправильный ввод.");
            choice = 0;
            getchar();
            break;
        }
    }

    if (make_eulerian == 1) {
        makeEulerian(graph, size, true);
        printf("\n Преобразованная матрица
смежности:\n\n");
        print_graph(graph, size);
    }

    Path result2;
    printf("\n Введите название файла для сохранения: ");
    scanf("%s", file_name);

    FILE* file = fopen(file_name, "w");
    if (file == NULL) {
        printf("Ошибка открытия файла\n");
        break;
    }

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            fprintf(file, "%d ", graph[j][i]);
        }
        fprintf(file, "\n");
    }

    if (findEulerianCycle(graph, size, true, &result2)) {
        fprintf(file, "\nЭйлеров цикл: ");
        for (int i = 1; i < result2.size; i++) {
            fprintf(file, "%d-%d ", result2.path[i - 1] +
1, result2.path[i] + 1);
        }
        fprintf(file, "\n");

        printf("\n Эйлеров цикл существует: ");
        for (int i = 1; i < result2.size; i++) {
            printf("%d-%d ", result2.path[i - 1] + 1,
result2.path[i] + 1);
        }
        printf("\n");
    }
    else {
        fprintf(file, "\nЭйлеров цикл не существует.\n");
        printf("\n Эйлеров цикл не существует.\n");
    }

    fclose(file);
    free_graph(graph, size);
    break;
}
case 3:
{

```

```

printf(" Введите название файла для чтения матрицы
смежности: ");
scanf("%s", file_name);

FILE* file = fopen(file_name, "r");
if (file == NULL) {
    printf(" Ошибка: файл не существует или не может
быть открыт.\n");
    break;
}

size = 0;
char line[1024];
if (fgets(line, sizeof(line), file)) {
    char* token = strtok(line, " \n");
    while (token != NULL) {
        size++;
        token = strtok(NULL, " \n");
    }
}

if (size < 2 || size > 1024) {
    printf(" Ошибка: некорректный размер матрицы.\n");
    fclose(file);
    break;
}

rewind(file);

graph = create_graph(size);
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        if (fscanf(file, "%hhhd", &graph[j][i]) != 1) {
            printf(" Ошибка чтения матрицы
смежности.\n");

            fclose(file);
            free_graph(graph, size);
            break;
        }
    }
}
fclose(file);

printf("\n Матрица смежности графа из файла:\n\n");
print_graph(graph, size);

bool is_directed = false;
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        if (graph[i][j] != graph[j][i]) {
            is_directed = true;
            break;
        }
    }
    if (is_directed) break;
}

int make_eulerian = 0;
if (!isEulerian(graph, size, is_directed)) {

```

```

        printf("\n Граф не является Эйлеровым. Хотите
сделать его Эйлеровым? (0 - нет, 1 - да): ");
        if (!scanf("%d", &make_eulerian) || make_eulerian
< 0 || make_eulerian > 1) {
            fprintf(stderr, " Неправильный ввод.");
            choice = 0;
            getchar();
            break;
        }
    }

    if (make_eulerian == 1) {
        makeEulerian(graph, size, is_directed);
        printf("\n Преобразованная матрица
смежности:\n\n");
        print_graph(graph, size);
    }

    Path result3;
    char output_file_name[128];
    printf("\n Введите название файла для сохранения
результатов: ");
    scanf("%s", output_file_name);

    FILE* output_file = fopen(output_file_name, "w");
    if (output_file == NULL) {
        printf(" Ошибка создания выходного файла.\n");
        break;
    }

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            fprintf(output_file, "%d ", graph[j][i]);
        }
        fprintf(output_file, "\n");
    }

    if (findEulerianCycle(graph, size, is_directed,
&result3)) {
        fprintf(output_file, "\nЭйлеров цикл: ");
        printf("\n Эйлеров цикл существует: ");
        for (int i = 1; i < result3.size; i++) {
            fprintf(output_file, "%d-%d ", result3.path[i
- 1] + 1, result3.path[i] + 1);
            printf("%d-%d ", result3.path[i - 1] + 1,
result3.path[i] + 1);
        }
        fprintf(output_file, "\n");
        printf("\n");
    }
    else {
        fprintf(output_file, "\nЭйлеров цикл не
существует.\n");
        printf("\n Эйлеров цикл не существует.\n");
    }

    fclose(output_file);
    free_graph(graph, size);
    break;
}

```

```

        case 4:
            return 0;

        default:
            printf(" Неправильный ввод.\n");
            choice = 0;
            getchar();
            break;
    }

    printf("\n Нажмите Esc чтобы вернуться в меню.\n");
    while (GetKeyState(VK_ESCAPE) >= 0) {
        Sleep(100);
    }
    system("cls");
}
return 0;
}

int_fast8_t** create_graph(int size) {
    int_fast8_t** array =
(int_fast8_t**)malloc(sizeof(int_fast8_t*) * size);
    for (int i = 0; i < size; i++) {
        array[i] = (int_fast8_t*)calloc(size,
sizeof(int_fast8_t));
    }
    if (array == NULL) {
        fprintf(stderr, " Ошибка создания массива.");
        exit(1);
    }
    return array;
}

void initialize_unorient_graph(int_fast8_t** graph_pointer, int
size, float edge_probability) {
    for (int i = 0; i < size; i++) {
        for (int j = i + 1; j < size; j++) {
            if ((float)rand() / (RAND_MAX / 100) <
edge_probability) {
                graph_pointer[i][j] = 1;
                graph_pointer[j][i] = 1;
            }
            else {
                graph_pointer[i][j] = 0;
                graph_pointer[j][i] = 0;
            }
        }
        graph_pointer[i][i] = 0;
    }
}

void initialize_orient_graph(int_fast8_t** graph_pointer, int
size, float edge_probability) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if ((float)rand() / (RAND_MAX / 100) <
edge_probability) {
                graph_pointer[i][j] = 1;
            }
            else {

```

```

        graph_pointer[i][j] = 0;
    }
}
graph_pointer[i][i] = 0;
}
}

void print_graph(int_fast8_t** graph_pointer, int size) {
    printf(" ");
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            fprintf(stdout, "%d ", graph_pointer[j][i]);
        }
        fprintf(stdout, "\n ");
    }
}

void free_graph(int_fast8_t** graph_pointer, int size) {
    for (int i = 0; i < size; i++) {
        free(graph_pointer[i]);
    }
    free(graph_pointer);
}

bool checkDegrees(int_fast8_t** graph_pointer, int size) {
    for (int i = 0; i < size; i++) {
        int degree = 0;
        for (int j = 0; j < size; j++) {
            degree += graph_pointer[i][j];
        }
        if (degree % 2 != 0) return false;
    }
    return true;
}

bool checkInOutDegrees(int_fast8_t** graph_pointer, int size) {
    for (int i = 0; i < size; i++) {
        int in_degree = 0, out_degree = 0;
        for (int j = 0; j < size; j++) {
            out_degree += graph_pointer[i][j];
            in_degree += graph_pointer[j][i];
        }
        if (in_degree != out_degree) return false;
    }
    return true;
}

bool isConnected(int_fast8_t** graph_pointer, int size) {
    bool visited[1024] = { false };
    int stack[1024], top = -1;

    stack[++top] = 0;
    visited[0] = true;

    while (top >= 0) {
        int vertex = stack[top--];
        for (int i = 0; i < size; i++) {
            if (graph_pointer[vertex][i] && !visited[i]) {
                stack[++top] = i;
                visited[i] = true;
            }
        }
    }
}

```

```

        }
    }

    for (int i = 0; i < size; i++) {
        if (!visited[i]) return false;
    }
    return true;
}

void dfs_helper(int v, int size, int_fast8_t** graph_pointer,
bool* visited, int* stack, int* stack_size) {
    visited[v] = true;
    for (int i = 0; i < size; i++) {
        if (graph_pointer[v][i] && !visited[i]) {
            dfs_helper(i, size, graph_pointer, visited, stack,
stack_size);
        }
    }
    stack[*stack_size] = v;
    (*stack_size)++;
}

void makeEulerian(int_fast8_t** graph_pointer, int size, bool
directed) {
    if (directed) {
        int* in_degree = (int*)calloc(size, sizeof(int));
        int* out_degree = (int*)calloc(size, sizeof(int));

        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (graph_pointer[i][j]) {
                    out_degree[i]++;
                    in_degree[j]++;
                }
            }
        }

        bool* visited = (bool*)calloc(size, sizeof(bool));
        int* stack = (int*)calloc(size, sizeof(int));
        int stack_size = 0;

        int start_vertex = 0;
        for (int i = 0; i < size; i++) {
            if (in_degree[i] != out_degree[i]) {
                start_vertex = i;
                break;
            }
        }

        dfs_helper(start_vertex, size, graph_pointer, visited,
stack, &stack_size);

        bool needs_connection = false;
        for (int i = 0; i < size; i++) {
            if (!visited[i]) {
                needs_connection = true;
                break;
            }
        }
    }
}

```

```

        if (needs_connection) {
            memset(visited, 0, size * sizeof(bool));
            stack_size = 0;

            // Повторный DFS для построения полного пути
            for (int i = 0; i < size; i++) {
                if (!visited[i]) {
                    dfs_helper(i, size, graph_pointer, visited,
stack, &stack_size);
                }
            }

            for (int i = 0; i < stack_size - 1; i++) {
                int v = stack[i];
                int u = stack[i + 1];
                if (!graph_pointer[v][u] && (in_degree[v] !=
out_degree[v] || in_degree[u] != out_degree[u])) {
                    graph_pointer[v][u] = 1;
                    out_degree[v]++;
                    in_degree[u]++;
                }
            }
        }

        for (int i = 0; i < size; i++) {
            if (in_degree[i] != out_degree[i]) {
                while (in_degree[i] > out_degree[i]) {
                    bool found = false;
                    for (int j = 0; j < size; j++) {
                        if (i != j && in_degree[j] < out_degree[j]
&& !graph_pointer[i][j]) {
                            graph_pointer[i][j] = 1;
                            out_degree[i]++;
                            in_degree[j]++;
                            found = true;
                            break;
                        }
                    }
                    if (!found) break;
                }
                while (in_degree[i] < out_degree[i]) {
                    bool found = false;
                    for (int j = 0; j < size; j++) {
                        if (i != j && in_degree[j] > out_degree[j]
&& !graph_pointer[j][i]) {
                            graph_pointer[j][i] = 1;
                            in_degree[i]++;
                            out_degree[j]++;
                            found = true;
                            break;
                        }
                    }
                    if (!found) break;
                }
            }
        }

        if (!isEulerian(graph_pointer, size, directed)) {

```

```

эйлеровости
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i != j && !graph_pointer[i][j] &&
in_degree[i] < out_degree[i] && in_degree[j] > out_degree[j]) {
                graph_pointer[i][j] = 1;
                out_degree[i]++;
                in_degree[j]++;
            }
        }
    }

    free(visited);
    free(stack);
    free(in_degree);
    free(out_degree);
}
else {
    int* degree = (int*)calloc(size, sizeof(int));

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (graph_pointer[i][j]) {
                degree[i]++;
            }
        }
    }

    for (int i = 0; i < size; i++) {
        if (degree[i] == 0) {
            int next = (i + 1) % size;
            graph_pointer[i][next] = graph_pointer[next][i] =
1;

            degree[i]++;
            degree[next]++;
        }
    }

    int max_degree = 0;
    for (int i = 0; i < size; i++) {
        if (degree[i] > max_degree) {
            max_degree = degree[i];
        }
    }

    for (int i = 0; i < size; i++) {
        if (degree[i] == max_degree && degree[i] % 2 != 0) {
            for (int j = 0; j < size; j++) {
                if (graph_pointer[i][j]) {
                    // Временно удаляем ребро
                    graph_pointer[i][j] = graph_pointer[j][i]
= 0;

                    // Проверяем, сохранилась ли связность
                    if (isConnected(graph_pointer, size)) {
                        degree[i]--;
                        degree[j]--;
                        if (degree[i] % 2 == 0) break;
                    }
                }
            }
        }
    }
}

```



```

        else {
            graph_pointer[i][j] =
graph_pointer[j][i] = 1;
        }
    }
}

for (int i = 0; i < size; i++) {
    if (degree[i] % 2 != 0) {
        for (int j = i + 1; j < size; j++) {
            if (degree[j] % 2 != 0 &&
!graph_pointer[i][j]) {
                graph_pointer[i][j] = graph_pointer[j][i]
= 1;
                degree[i]++;
                degree[j]++;
                break;
            }
        }

        if (degree[i] % 2 != 0) {
            for (int j = 0; j < size; j++) {
                if (i != j && !graph_pointer[i][j]) {
                    graph_pointer[i][j] =
graph_pointer[j][i] = 1;
                    degree[i]++;
                    degree[j]++;
                    break;
                }
            }
        }
    }

    free(degree);
}

bool isEulerian(int_fast8_t** graph_pointer, int size, bool
directed) {
    if (directed) {
        if (!checkInOutDegrees(graph_pointer, size)) return false;
    }
    else {
        if (!checkDegrees(graph_pointer, size)) return false;
    }

    if (!isConnected(graph_pointer, size)) return false;

    return true;
}

void findEulerPath(int_fast8_t** graph_pointer, int size, int
start, Path* result, bool directed) {
    short stack[2048];
    int top = -1;

```

```

    stack[++top] = start;

    while (top >= 0) {
        int current = stack[top];
        bool found = false;

        for (int i = 0; i < size; i++) {
            if (graph_pointer[current][i] > 0) {
                stack[++top] = i;
                graph_pointer[current][i]--;
                if (!directed) {
                    graph_pointer[i][current]--;
                }
                found = true;
                break;
            }
        }
        if (!found) {
            result->path[result->size++] = stack[top];
            top--;
        }
    }
}

bool findEulerianCycle(int_fast8_t** graph_pointer, int size, bool
directed, Path* result) {
    result->size = 0;

    if (!isEulerian(graph_pointer, size, directed)) {
        return false;
    }
    int_fast8_t** temp_graph =
(int_fast8_t**)malloc(sizeof(int_fast8_t*) * size);
    for (int i = 0; i < size; i++) {
        temp_graph[i] = (int_fast8_t*)malloc(sizeof(int_fast8_t) *
size);
        for (int j = 0; j < size; j++) {
            temp_graph[i][j] = graph_pointer[i][j];
        }
    }

    findEulerPath(temp_graph, size, 0, result, directed);
    for (int i = 0; i < size; i++) {
        free(temp_graph[i]);
    }
    free(temp_graph);

    return true;
}

```