Министерство науки и образования РФ
Федеральное государственное автономное образовательное
учреждение высшего профессионального образования
«Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В. И. Ульянова (Ленина)»
(СПбГЭТУ «ЛЭТИ»)

Факультет компьютерных технологий и информатики

Кафедра вычислительной техники

Пояснительная записка к курсовой работе по теме «Графы» по дисциплине «Алгоритмы и структуры данных» Вариант 24

Выполнил студент гр.9308: Хамитов А.К.

Проверил: Колинько П.Г.

Санкт-Петербург, 2020 г.

Оглавление

Введение

Исследование алгоритмов, реализуемых с помощью графов.

1. Задание.

Построение эйлерова цикла в ориентированном графе.

2. Формализация задания

Эйлеров цикл - замкнутый путь, проходящий через каждое ребро графа ровно по одному разу.

Рассмотрим общий случай – ориентированный граф с петлями.

Ориентированный граф содержит эйлеров цикл тогда и только тогда, когда он сильно связан или среди его компонент сильной связности только одна содержит ребра и в вершину входит столько же ребер, сколько из нее выходит.

Исходя из условий эйлеровости графа, для решения поставленной задачи реализовано поочередная проверка: на равенство суммы входящих и выходящих ребер и на сильно связность. Такой порядок связан с сложностью каждой из верификаций: первая работает быстрее, а значит, если тест провален – второй проходить нет необходимости.

Поиск самого эйлерова цикла осуществляется через алгоритм Хирхольцера, что с подвигает нас на еще один вопрос: возможно ли проверку и построение объединить?

Во-первых, стоит заметить, что действительно случайно сгенерированный ориентированный граф будет иметь низкий шанс быть эйлеровым.

Действительно, при первой верификации, рассматривая граф в виде матрицы смежности, сумма элементов i-ой стройки должно равняться сумме элементов i-ого столбца для каждого целого i меньшего кол-ва вершин в графе. Предполагаемая вероятность $1/n^2$, где n – кол-во вершин в графе, что еще требует формального доказательства. С вероятностью сильно связности дела обстоят сложнее, но возможный способ вычисления, и интересный, по моему мнению, будет основываться на алгоритме Уоршелла. Также общую вероятность можно вычислять, не исходя из проверок, а напрямую: будет ли существовать эйлеров цикл, подсчитывая, например, вероятность существование определенного ребра по отдельности. Однако, если оставить формальности, из здравого смысла можно утверждать, что вероятность достаточно низкая, чтобы оправдать верификации перед использованием алгоритма Хирхольцера. Поэтому в программе также реализован генератор

эйлеровых графов, чтобы избежать длительного ожидания выпадения нужного случая и наглядной проверки, что алгоритм действительно работает.

Во-вторых, мной не придуман и не найден алгоритм, который бы не основывался при поиске эйлерова цикла, что граф эйлеров. Что в принципе и логично: если бы существовал алгоритм поиска эйлерова цикла за линейную сложность относительно количества ребер или вершин, или и того и другого вместе, без проверки на эйлеровость, тогда, не забыв, что суждения про существования эйлерова цикла необходимы и достаточны, все примененные мной алгоритмы в программе не имели столь важную значимость в теории графов.

Тогда остается вариант с объединённым алгоритмом, который имеет приблизительно такую же сложность, что все остальные вместе взятые, что вполне возможно, но тогда не имеющий смысла.

Поговорим про детали реализации:

Количество входящих (in_deg) и выходящих (out_deg) ребер вершины отслеживается на этапе ввода или генерации графа. Они имеют особую важность в реализации, потому что будут использоваться на протяжении выполнения всей программы: проверки на сильно связность и нахождении эйлерова цикла.

Построения графа осуществляется в виде матрицы смежности из-за удобства ввода и генерировании графа. Двумерный массив займет узкое место в памяти, а после этапа входных данных произойдет ее высвобождение. Дальше происходит переход матрицы смежности в списки смежности.

Отчасти вышеперечисленное и второстепенные моменты описаны краткими комментариями в исходном тексте программы.

3. Обоснование выбора способа представления графа в памяти ЭВМ

Граф в памяти представлен в виде списка смежности. На это есть свои причины (в нижеперечисленном n и m – кол-во вершин и ребер соответственно):

Во-первых, скорее всего меньшее кол-во используемый памяти. Скорее всего, потому что, несмотря на меньшее кол-во используемой памяти n + n(n-1)/2 в худшем случае (полный граф), по сравнению с матрицей смежности, которой необходимо n^2, массив хранит данные последовательно, в отличие от списков. Поэтому не совсем ясно, где стоит отчертить границы, когда стоит использовать то или иное представление, однако в нашем случае мы тестируем программу для кол-ва ребер, которое в основном достаточно меньше, чем кол-во вершин, чтобы использовать списки смежности.

Во-вторых, алгоритм Косарайю и Хирхольцера основываются на обходе графа, что имеет временную сложность O(n^2) для матрицы смежности и O(n+m) для списков. Второе представление данных в памяти выигрывает. Но возможно, что есть алгоритмы, которые могут работать быстрее для матрицы смежности. Однако такие алгоритмы мной не придуманы и не найдены, но кажется интересным проверка на сильно связность через алгоритм Уоршела.

4. Временная сложность функций

1. **Ввод графа**

   Сложность: O(*n*^2), где n — количество вершин графа.

2. **Генерация случайного графа**

   Сложность: O(*n*^2), где n — количество вершин графа.

3. **Переход к спискам смежности.**

   Сложность: O(*n*^2), где n — количество вершин графа.

4. **Проверка на равенство кол-ва входящих и выходящих ребер у каждой вершины графа.**

   Сложность: O(*n*), где n — количество вершин графа.

```
bool GR::in_out_deg() {
    for(int i = 0; i < n; i++)
        if(out_deg[i] != in_deg[i]) return false;

    return true;
}
```

Рисунок 1. Функция in_out_deg

5. **Алгоритм Косарайю для проверки на сильно связность.**

Сложность: O(*n*^2) для насыщенных графов и O(n+m) для разреженных.

Алгоритм производит 2 поиска в глубину в худшем случае по O(n+m) и 2 раза пробегается по вектору *vis* в худшем случае за O(n).

```
// лямбда-выражение - краткая форма записи анонимных функторов (
visit = [&](int u) {    // [&] - Позволяет нам захватить перемен
    if (!vis[u]) {
        vis[u] = true;
        for (auto v : LIST[u]) {
            visit(v);
            t[v].push_back(u); // Меняем направление, т.е. строи
        }
    }
};
while(!out_deg[i]) i++; // Доходим до вершины с ненулевой степен
visit(i);

for (int j = 0; j < size; ++j)
    if(!vis[j] && out_deg[j]) return false;    // Если найдется

// Аналогично проходимся по транспонированному графу
std::function<void(int)> assign;
assign = [&](int u) {
    if (vis[u]) {
        vis[u] = false;
        for (auto v : t[u]) {
            assign(v);
        }
    }
};
assign(i);

for (i = 0; i < size; ++i)
    if(vis[i] && out_deg[i]) return false;
```

Рисунок 2. Функция kosaraju

6. Алгоритм Хирхольцера для построения эйлерова цикла

Сложность: O(m), где m –кол-во ребер графа.

Алгоритм проходится по каждому ребру ровно по одному разу.

```
while (!curr_path.empty())
{
    // Если есть инцидентные ребра, то
    if (out_deg[curr_v])
    {
        // Добавляем в путь очередную вершину
        curr_path.push(curr_v);
        // Достаем следующую вершину смежную с curr_v
        next_v = LIST[curr_v].back();
        // и удаляем ребро ведущее к next_v
        out_deg[curr_v]--;
        LIST[curr_v].pop_back();
        // Переходим к следующей вершине
        curr_v = next_v;
    }

    // Если нет, то
    else
    {
        // Добавляем вершину в эйлеров цикл
        circuit.push_back(curr_v);

        // Достаем последнюю вершину в пути и раз мы ее рассматриваем на следующем шаге, то удалим ее из стека
        curr_v = curr_path.top();
        curr_path.pop();
    }
}
```

Рисунок 3. Отрывок функции Heirholzer_printCurcuit

5. Контрольные примеры

Контрольные примеры представлены в таблице 1.

Таблица 1: Контрольные примеры

№	Входные данные		Результат
	Количество вершин	Список смежности	Эйлеров цикл
1	5	a: a - - - - b: - - - - e c: - - - - - d: - - - d - e: - - - - -	The graph is not euler
2	8	a: - b - - e - - - b: a b - d - - - - c: - b - - - - - - d: - - - - - - - h e: - - c - - - - - f: - - - - - - - - g: - - - - - - - - h: a - - - - - - -	a -> e -> c -> b - > d -> h -> a -> b -> b -> a
3	5	a: - b - - - b: - - c d - c: a - - - - d: - - - - e e: - b - - -	a -> b -> d -> e - > b -> c -> a
4	0		The graph has zero vertices
5	3	a: a b - - - b: - - c d e c: - - - - e	The graph is not euler

		d: - - - - e e: - - - - e	

6. Результаты проверки алгоритма

Результаты проверки алгоритма представлены на рисунках.

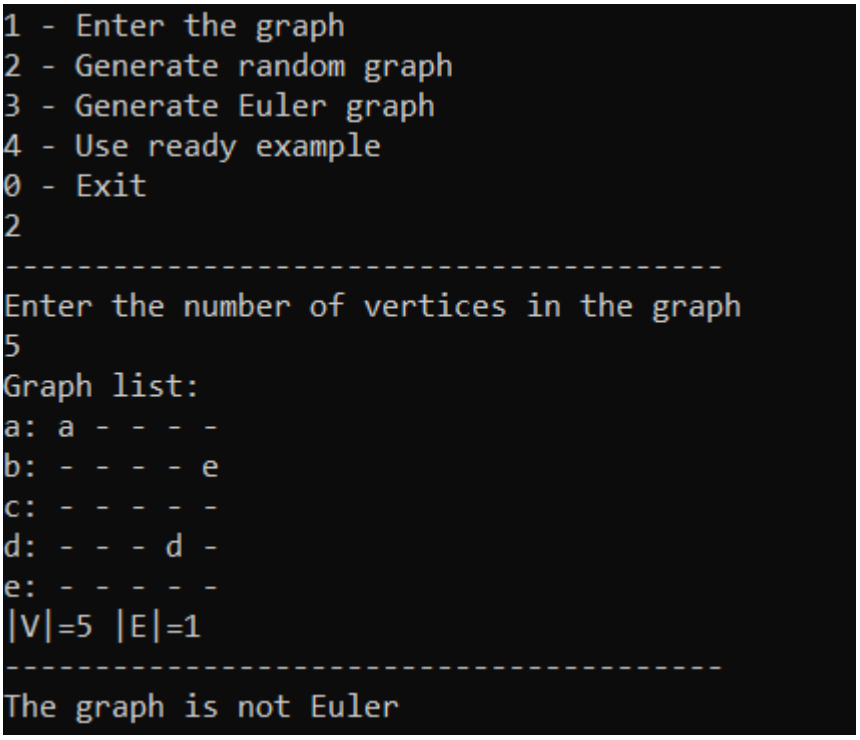

Рисунок 4. Тест 1

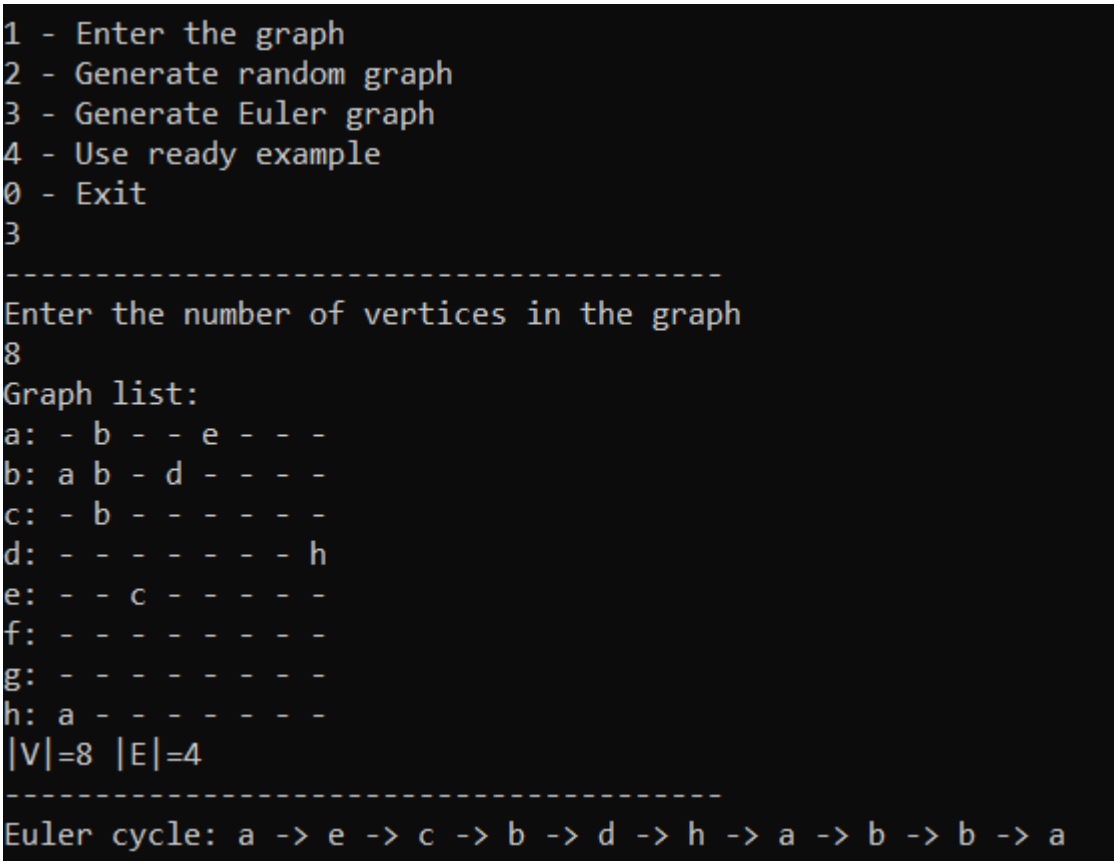

Рисунок 5. Тест 2

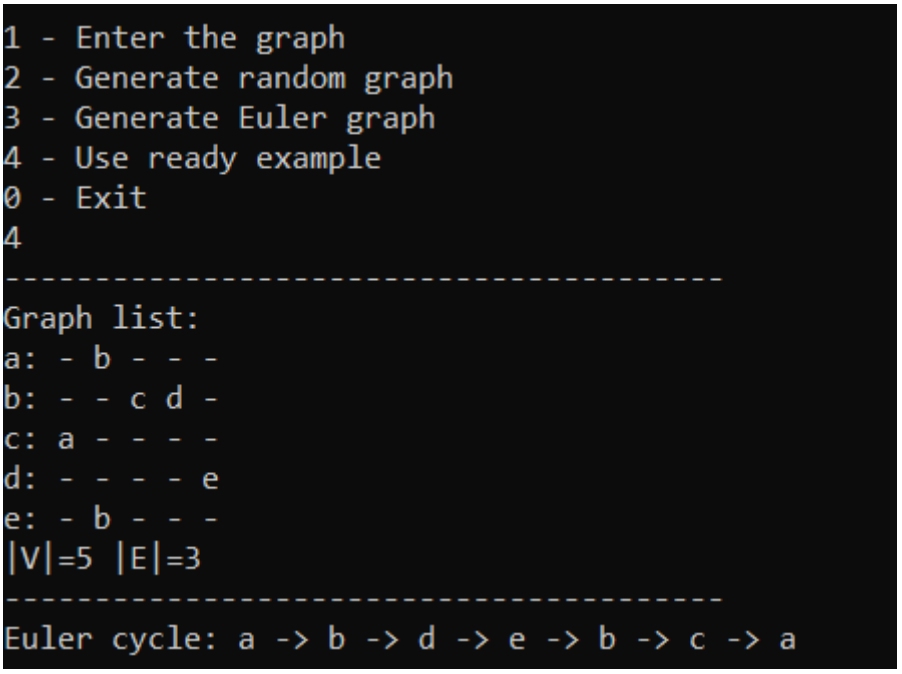

Рисунок 6. Тест 3

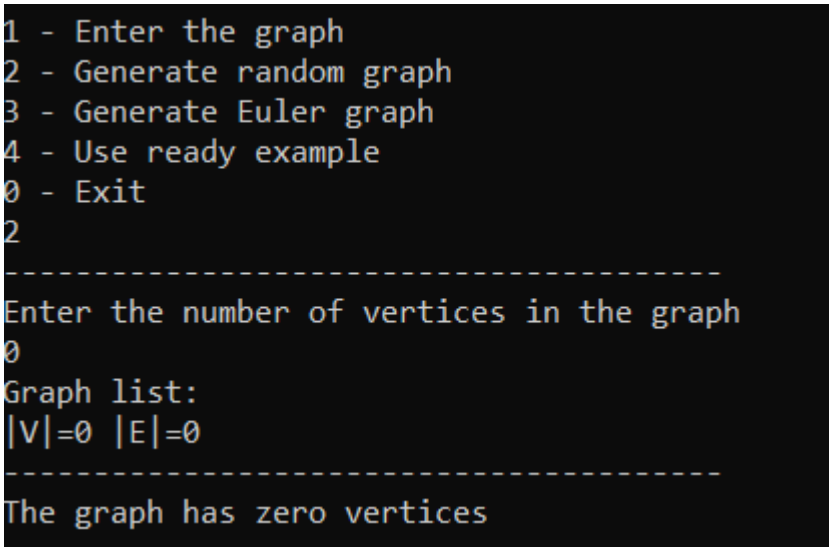

Рисунок 7. Тест 4

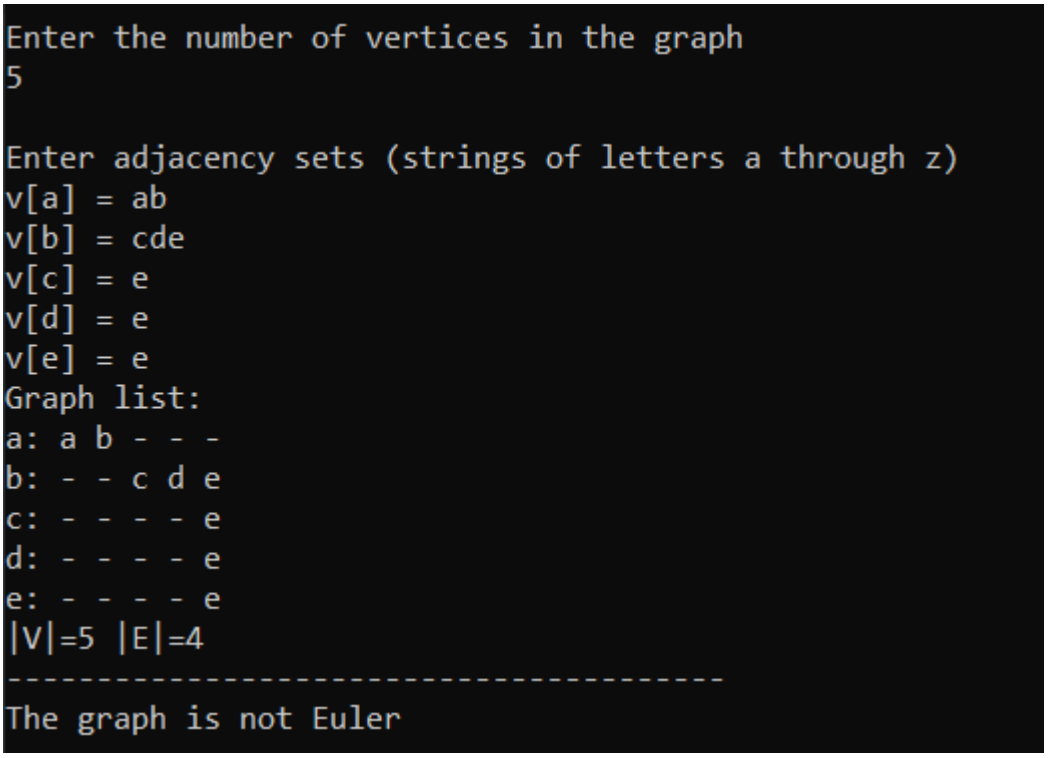

Рисунок 8. Тест 5

Вывод

Исследован и реализован алгоритм построения эйлерова цикла в ориентированном графе с петлями. Использованы алгоритмы Косарайю и Хирхольцера, а также поверхностно рассмотрен вопрос о вероятности случайного генерировании графа и выбран оптимальный способ осуществления поставленной задачи. При тестировании программы ошибки не обнаружены.

Список используемых источников

1. Колинько П.Г. Пользовательские структуры данных / Методические указания по дисциплине «Алгоритмы и структуры данных» - Санкт-Петербург: СПбГЭТУ «ЛЭТИ», 2020.
2. Поздняков С.Н, Рыбин С.В Дискретная математика / Учебник для студ. Вузов — Издательский центр «Академия» 2008 — 448с
3. Роберт Седжвик. Алгоритмы на графах = Graph algorithms. — 3-е изд. — Россия, Санкт-Петербург: «ДиаСофтЮП», 2002. — С. 496. — ISBN 5-93772-054-7.
4. Hierholzer, Carl (1873), "Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren", Mathematische Annalen, **6** (1): 30–32, doi:10.1007/BF01442866.
5. Райгородский А. М. Модели случайных графов. — М.: МЦНМО,

Приложение 1 (Исходный текст программы)

```
#include <iostream>
#include <functional>
#include <vector>
#include <list>
#include <stack>
#include <ctime>


using namespace std;
const int MaxV = 26;

char Ch(int c) {    // Возвращает с-ый символ алфавита
    return c + 'a';
}

class GR {
    int n, m;                           // Кол-во вершин и ребер
    int in_deg[MaxV] = {0};             // Массив количества входящих ребер
    int out_deg[MaxV] = {0};            // Массив количества выходящих ребер
    vector < list < int >> LIST;        // Задаем вектор, элементы которого - списки

    //--приватные функции конструкторы и операторы для отслеживания их случайного вызова--
    GR(const GR &);
    GR(GR &&);

    GR operator=(const GR &) const = delete;
    GR operator=(GR &&) const = delete;
    public:
        GR(int, int, int &, int &);
        // Проверка на существование эйлерова цикла
        bool kosaraju();  // Проверка на сильно связность
        bool in_out_deg();  // Проверка на то, что для каждой вершины сумма входяших равна
сумме выходящих ребер
        void Hierholzer_printCurciut(); // Поиск эйлерова цикла через алгоритм Хирхольцера
        ~GR() =
            default; // указывает компилятору самостоятельно генерировать деструктор
};

GR::GR(int MaxV, int menu, int &v, int &e): n(0), m(0) {
    string s;
    int G[MaxV][MaxV];                  // Матрица смежности
    int start,
        k1,
        k2,
        counter;
    for (int i = 0; i < MaxV; ++i)
        for (int j = 0; j < MaxV; ++j) G[i][j] = 0;

    switch(menu)
    {
        case 1:
            // Ввод данных в матрицу смежности
```

```
            cout << "Enter the number of vertices in the graph\n";
            do {
                cin >> n;
                if(n < 0 || n > 26) cout << "The number of vertices range from 0 to 26";
            } while(n < 0 || n > 26);

            counter = 0;
            cout << "\nEnter adjacency sets (strings of letters a through z)\n";
            do {
                cout << "v[" << Ch(counter) << "] = ";
                // cout << "v[" << n << "]= ";
                cin >> s;
                for (auto i: s)                     // Эквивалентно for(auto s = string.beg
in(); s != string.end(); ++s)
                    if (isalpha(i)) {               // Игнорируем символы не принадлежащие
алфавиту
                        int j = (tolower(i) -
 'a'); // tolower возвращает строчный эквивалент символа
                        G[counter][j] = 1;          // Ориентированный граф
                        (in_deg[j])++;              // Увеличиваем количество входящих ребе
р в j на 1
                        (out_deg[n])++;
                    }
                counter++;
            } while (isalpha(s[0]) && counter < n);      // Пока первый символ введенной ст
роки не 0 и n не превышает MaxV
        break;
        case 2:
            // Генерация случайной матрицы смежности
            cout << "Enter the number of vertices in the graph\n";
            do {
                cin >> n;
                if(n < 0 || n > 26) cout << "The number of vertices range from 0 to 26";
            } while(n < 0 || n > 26);

            for (int i = 0; i < n; ++i)
                for (int j = i; j < n; ++j)
                    if (G[i][j] = rand()%2)
                    {
                        (in_deg[j])++;
                        (out_deg[i])++;
                    }
        break;
        case 3:
            // Генерация эйлерова графа
            // Вместо того, чтобы строить граф по определенным признакам -
 мы будем строить эйлеров цикл
            cout << "Enter the number of vertices in the graph\n";
            do {
                cin >> n;
                if(n < 0 || n > 26) cout << "The number of vertices range from 0 to 26";
            } while(n < 0 || n > 26);

            counter = 0;
```

```
            if(n) {
                start = rand()%(n);
                k1 = start;
                k2 = rand()%(n);
                do
                {
                    if(!G[k1][k2]) {        // Мы не работаем с мультиграфом,
поэтому отслеживаем реальное кол-во out/in_deg (нужно для алгоритма Хирхольцера)
                        G[k1][k2] = 1;
                        k1 = k2;
                        (in_deg[k2])++;
                        (out_deg[k1])++;
                        k2 = rand()%(n);
                    }
                    if(counter++ > MaxV*MaxV) k2 = start;  // Как показывает практика, k2 м
ожет никогда не стать start, поэтому делаем это вручную
                }
                while(k1 != start);
            }
        break;
        case 4:
            // Заготовленный пример
            n = 5;
            G[0][1] = 1; G[1][2] = 1; G[1][3] = 1; G[2][0] = 1; G[3][4] = 1; G[4][1] = 1;
            for(int i = 0; i < n; i++)
            {
                for(int j = 0; j < n; j++)
                    in_deg[i] += G[i][j];
                out_deg[i] = in_deg[i];
            }
            // {0, 1, 0, 0, 0},
            // {0, 0, 1, 1, 0},
            // {1, 0, 0, 0, 0},
            // {0, 0, 0, 0, 1},
            // {0, 1, 0, 0, 0},
        break;
        default:
            cout << "menu range from 0 - 4!\n";
    }

    // Формирование и вывод списка смежности
    cout << "Graph list:";
    LIST.resize(n); // Изменение размера
    for (int i = 0; i < n; ++i) {
        int f = 0;
        cout << '\n' << Ch(i) << ": ";
        // cout << '\n' << i << ": ";
        for (int j = 0; j < n; ++j)
            if (G[i][j]) {
                ++f;
                LIST[i].push_back(j);
                cout << Ch(j) << ' ';
                //cout << j << ' ';
            }
```

```
        else cout << "- ";
        m += f;
    }
    cout << "\n|V|=" << n << " |E|=";  m ? cout << m / 2 << '\n': cout << m << '\n'; // На случай, если m = 1
    v = n; e = m;
}

bool GR::in_out_deg() {
    for(int i = 0; i < n; i++)
        if(out_deg[i] != in_deg[i]) return false;

    return true;
}

bool GR::kosaraju() {
    int i = 0;  // Вершина, степень которой больше нуля
    auto size = n;
    std::vector<bool> vis(size);           // По умолчанию все значения инициализирует как ложь
    vector < list < int >> t(size);         // Транспонированный граф

    // Раз разрешено пользоваться всей библиотекой STL, то решил изучить для себя лямбда выражения и обертки

    // std::function -
 полиморфная обертка, способная хранить, копировать и ссылаться на любой обьект. В том числе и лямбда-выражение.
    // Полиморфный - способный обрабатывать данные разных типов
    std::function<void(int)> visit; // Создаем обертку visit, которая ничего не возвращает и принимает в качестве параметра int

    // В глубину
    // Храним лямбда-выражение в обертке visit
    // Лямбда-выражение - краткая форма записи анонимных функторов (функтор -
 обьект, который можно использовать как функцию)
    visit = [&](int u) {    // [&] -
 Позволяет нам захватить переменную по ссылке. (int u) - используемая переменная в лямбда-выражении
        if (!vis[u]) {
            vis[u] = true;
            for (auto v : LIST[u]) {
                visit(v);
                t[v].push_back(u); // Меняем направление, т.е. строим транспонированный граф
            }
        }
    };
    while(!out_deg[i]) i++; // Доходим до вершины с ненулевой степенью
    visit(i);

    for (int j = 0; j < size; ++j)
        if(!vis[j] && out_deg[j]) return false;    // Если найдется не посещенная вершина, степень которой больше 0, то граф не сильно связный, а значит не содержит эйлерова цикла
```

```
    // Аналогично проходимся по транспонированному графу
    std::function<void(int)> assign;
    assign = [&](int u) {
        if (vis[u]) {
            vis[u] = false;
            for (auto v : t[u]) {
                assign(v);
            }
        }
    };
    assign(i);

    for (i = 0; i < size; ++i)
        if(vis[i] && out_deg[i]) return false;

    return true;
}

// Алгоритм Хирхольцера
// Выбираем любую начальную вершину v и следуем по ребрам от этой вершины пока не вернемся к v.
// Невозможно застрять ни в одной вершине, кроме v, потому что четная степень всех вершин гарантирует,
// что, когда мы входим в другую вершину w, должно быть неиспользуемое ребро, выходящее из w.
// Сформированный таким образом путь является циклом, но он может не охватывать все ребра исходного графа.
// (1) Теперь добавляем в вектор вершины, у которых не осталось инцидентных ребер.
// (2) После этого возвращаемся к вершине u, у которой есть инцидентные ребра и следуем по ребрам из этой вершины пока не вернемся к u
// Повторяем шаги (1) и (2). Полученный вектор - будет упорядоченный эйлеров цикл, но почему?

// Формального доказательства я не нашел как и в принципе любого другого, но из здравого смысла можно додумать следующее:
// Первый получаемый цикл, если он не эйлеров - не проходит по всем ребрам, но если бы мы нашли цикл, который дополнил наш существующий,
// то мы бы приблизились к эйлерову, что собственно и делает наш алгоритм. И логичным утверждением будет, что дополняющий цикл,
// должен иметь пересечение с существующим через вершину, которая еще имеет неиспользуемые инцидентные ребра
void GR::Hierholzer_printCurciut() {
    // if (!LIST.size())
    // {
    //     cout << "graph empty";
    //     return; // Пустой граф
    // }

    int i = 0,  // Вершина с ненулевой степенью
        next_v,
        curr_v; // Текущая вершина
    // Текущий путь
    stack<int> curr_path;
```

```
    // Эйлеров цикл
    vector<int> circuit;

    while(!out_deg[i]) i++;
    curr_path.push(i);
    curr_v = i;

    while (!curr_path.empty())
    {
        // Если есть инцидентные ребра, то
        if (out_deg[curr_v])
        {
            // Добавляем в путь очередную вершину
            curr_path.push(curr_v);
            // Достаем следующую вершину смежную с curr_v
            next_v = LIST[curr_v].back();
            // и удаляем ребро ведущее к next_v
            out_deg[curr_v]--;
            LIST[curr_v].pop_back();
            // Переходим к следующей вершине
            curr_v = next_v;
        }

        // Если нет, то
        else
        {
            // Добавляем вершину в эйлеров цикл
            circuit.push_back(curr_v);

            // Достаем последнюю вершину в пути и раз мы ее рассматриваем на следующем шаге
, то удалим ее из стека
            curr_v = curr_path.top();
            curr_path.pop();
        }
    }

    // Вывод
    cout << "----------------------------------------\nEuler cycle: ";
    for (int i=circuit.size()-1; i>=0; i--)
    {
        cout << Ch(circuit[i]);
        // cout << circuit[i];
        if (i)
            cout<<" -> ";
    }
    cout << "\n----------------------------------------\n";
}

int main() {

    srand(time(nullptr));
    int menu,
        v, e;
    do
```

```
    {
        cout << "1 - Enter the graph\n";
        cout << "2 - Generate random graph\n";
        cout << "3 - Generate Euler graph\n";
        cout << "4 - Use ready example\n";
        cout << "0 - Exit\n";
        cin >> menu;
        cout << "----------------------------------------\n";
        if(menu < 5 && menu > 0) {
            GR graph(MaxV, menu, v, e);
            if(!v) cout << "----------------------------------------
\nThe graph has zero vertices\n----------------------------------------\n";
            else if(!e) cout << "----------------------------------------
\nThe graph has zero edges\n----------------------------------------\n";
            else
                if(graph.in_out_deg() && graph.kosaraju())
                    graph.Hierholzer_printCurciut();
                else cout << "----------------------------------------
\nThe graph is not Euler\n----------------------------------------\n";
        }
        else if(menu) cout << "Menu range from 0 to 4!\n";
    } while(menu);

    return 0;
}
```