МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теоретических основ компьютерной безопасности и криптографии

**Гамильтоновость и эйлеровость**

НАУЧНО-ИССЛЕДОВСКАЯ РАБОТА

студента 5 курса 531 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Енца Михаила Владимировича

|  |  |  |
| --- | --- | --- |
| Научный руководитель  д.ф.-м.н., доцент | \_\_\_\_\_\_\_\_\_\_ | М. Б. Абросимов |
|  | подпись, дата |  |

Саратов 2019

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc10389587)

[1 Необходимые определения 4](#_Toc10389588)

[2 Гамильтоновость и эйлеровость графа 5](#_Toc10389589)

[2.1 Гамильоновость 5](#_Toc10389590)

[2.1.1 Алгоритм поиска всех гамильтоновых циклов 5](#_Toc10389591)

[2.2 Эйлеровость 6](#_Toc10389592)

[3 Выполнение работы 7](#_Toc10389593)

[4 Результаты исследования 9](#_Toc10389594)

[ЗАКЛЮЧЕНИЕ 20](#_Toc10389595)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 21](#_Toc10389596)

[Приложение А. Листинг программы 22](#_Toc10389597)

[Приложение Б. Каталог графов 33](#_Toc10389598)

# ВВЕДЕНИЕ

Целью данной работы является поиск и подсчет количества гамильтоновых циклов в графе и проверка графа на эйлеровость во всех двусвязных графах с количеством вершин от трех до десяти включительно. Для поиска гамильтоновых циклов и для проверки графов на эйлеровость разработаем программу на языке C++, которая в основе своей будет использовать обход графа в глубину, а также критерии для проверки графа на эйлеровость. Полученные результаты будут записаны в файл и приведены в данной работе.

Основным результатом и интересом работы будут таблицы с количеством графов для каждого класса графа. В качестве дополнения будут приведены иллюстрации графов, представляющих из себя интересные случаи. Графы будут сгенерированы с помощью генератора , входящего в состав программного комплекса .

# 1 Необходимые определения

*Неориентированным графом* (далее будем называть графом) называется пара , где – симметричное и антирефлексивное отношение на множестве вершин **,** называемое отношением смежности.

Если , то говорят, что вершины и *смежные* и эти вершины соединены ребром . Если , то вершины и *несмежные*. При этом и это одно и то же ребро, которое обозначают . Говорят, что ребро *инцидентно* каждой из вершин и и эти вершины называются концевыми вершинами или концами ребра . Два ребра называются *смежными*, если они имеют общую концевую вершину.

Степенью вершины в графе будем называть количество вершин в , смежных с . Вершина, не смежная ни с одной другой вершиной, называется *изолированной*, а вершина, смежная со всеми остальными вершинами, называется *полной*. Вершина называется *четной* или *нечетной* в зависимости от четности или нечетности своей степени.

Две вершины и называются *связными*, если в графе существуют путь из в .

*Компонентой связности* называется класс эквивалентности относительно связности.

*Связный граф* – это граф, в котором между любыми несовпадающими вершинами существует путь.

*Двусвязный граф* – это связный и неделимый граф, в том смысле, что удаление любого ребра не приведет к потере связности. [1]

# 2 Гамильтоновость и эйлеровость графа

# 2.1 Гамильоновость

Цикл или цепь, содержащие все вершины графа, называется *гамильтоновым*. Граф, содержащие гамильтонов цикл, также называется *гамильтоновым*. Для проверки гамильтоновости произвольного графа нет эффективных условий, приведем несколько достаточных условий гамильтоновости.

Теорема (Хватал, 1972). Пусть граф с вектором степеней () и . Если для любого верна импликация

то граф гамильтонов.

Теорема (Оре, 1960). Если в связном n-вершинном графе для любых двух несмежных вершин и выполняется неравенство , то это граф гамильтонов. [1]

# 2.1.1 Алгоритм поиска всех гамильтоновых циклов

Опишем алгоритм, основанный на поиске в глубину. [2]

Запустим обход в глубину из произвольной вершины графа и обозначим её через . На каждом уровне рекурсии мы имеем текущую вершину , относительно которой будем рассматривать смежные вершины.

Пусть мы находимся в обходе в глубину, пометим вершину как использованную. Просматривая смежные вершины вершины , выбираем не помеченную ранее и запускаем обход в глубину от неё. Также в процессе погружения в рекурсию запоминаем все помеченные вершины. Если количество запомненных ранее вершин равно количеству вершин в графе, то гамильтонов путь найден, сохраняем его и возвращаемся на уровень выше.

Алгоритм. Поиск всех гамильтоновых циклов.

Вход. Массив – список помеченных вершин, изначально он пустой, – текущий путь, изначально состоит из единственной вершины .

Выход. Массив paths – список всех гамильтоновых путей в графе .

1. Если размер равен количеству вершин в графе , то добавляем в и поднимаемся на уровень рекурсии выше;
2. В массив добавляем ;
3. Для всех , выполняем шаг 4;
4. если , то добавляем вершину в и запускаем этот алгоритм c . После выхода из рекурсии удаляем вершину из ;
5. Удаляем из вершину ;
6. Выходим из текущего уровня рекурсии.

# 2.2 Эйлеровость

Путь, который содержит все ребра графа, называется эйлеровым. Циклический путь, который содержит все ребра графа, называется эйлеровым циклом, а граф с таким путем – эйлеровым. Следующие теоремы дают эффективные критерии для проверки эйлеровости.

Теорема (Эйлер, 1736). Связный граф тогда и только тогда является эйлеровым, когда все его вершины четны. [1]

# 3 Выполнение работы

Для генерации всевозможных графов с заданным количеством вершин используется генератор , входящий в состав программного комплекса . [4]

Данный генератор включает в себя множество параметров генерации неориентированных графов. В данной работе рассмотрены всевозможные неориентированные двусвязные графы в формате . Для генерации использовался параметр, отвечающий за количество вершин графа, и параметр , задающий генерацию двусвязных графов.

Например, для генерации всевозможных двусвязных графов с количеством вершин 11 в файл 11v.txt, необходимо перейти в папку, содержащую генератор и в командной строке запустить команду

|  |
| --- |
| ./geng -C -g 11 > 11v.txt |

В результате получим файл с двусвязными графами, отсортированными в порядке уплотнения. Так, первый графа — это граф, содержащий ребро, а последний граф в файле содержит ребро.

Листинг программы, которая принимает на вход графы в формате , а затем применяет к ним алгоритм для поиска всех гамильтоновых циклов и проверяет их на эйлеровость, представлен в приложении А. Программу необходимо запускать для каждого набора графов отдельно. Результатом работы программы являются файлы с графами, разделенными на классы: гамильтонов и эйлеров, гамильтонов и не эйлеров, не гамильтонов и эйлеров, не гамильтонов и не эйлеров. Также подсчитывается общая статистика гамильтоновых циклов в формате <количество циклов>:<количество графов с таким количеством циклов>.

Программа поддерживает 2 режима работы однопоточный и многопоточный, количество потоков определяется по количеству ядер машины, на которой выполняется программа.

Для запуска программы в многопоточном режиме, с поданным на вход файлом «11v.txt» с графами в формате *graph6*, нужно передать следующие аргументы:

|  |
| --- |
| -f 11v.txt -m -t g6 |

Содержимое файлов с результатами приведено в следующем разделе.

# 4 Результаты исследования

По таблице 1 видно, что графы с количеством вершин 10 стали пределом для обработки на персональном компьютере.

Таблица 1 – Свод информации о данных, необходимых для работы

|  |  |  |
| --- | --- | --- |
| Количество вершин | Количество всевозможных двусвязных неориентированных графов | Время работы разработанной программы, секунд |
| 3 | 1 | 0.062 |
| 4 | 3 | 0.048 |
| 5 | 10 | 0.062 |
| 6 | 56 | 0.195 |
| 7 | 468 | 0.055 |
| 8 | 7123 | 0.852 |
| 9 | 194066 | 46.054 |
| 10 | 9743542 | 3982 (1.1 ч) |
| 11 | 900969091 | Больше 38 ч. |

Для оценки асимптотики необходимо использовать формулу для подсчета сочетаний. В случае поиска всех гамильтоновых путей, в худшем случае, необходимо выбрать ребро из : .

Так, для 11 вершинного графа: операций на обработку одного графа. Умножив на количество графов, получим: 26352129089924768130‬ операций. Грубо усреднив сложность, поделив пополам, также с учетом того, что программа для обработки многопоточная, поделим на 4 потока получим 3294016136240596016 операций, что слишком много для вычисления всех 11-вершинных графов на персональном компьютере.

Так, экспериментальным путем было установлено, что чтобы вычислить последние 50 11-вершинных графов в 4 потока на персональном компьютере, требуется 50 секунд.

Программа была запущена для первых 300000000 графов из набора одиннадцативершинных графов. Программа отработала за 136219 секунд (~38 часов). Так как плотность графов увеличивается, то запускать программу для следующего блока графов не представляется возможным на персональном компьютере.

В таблице 2 показано соотношение классов графов для различных наборов графов. В каждой ячейке таблицы указано количество графов из набора, принадлежащих одному из классов.

Таблица 2

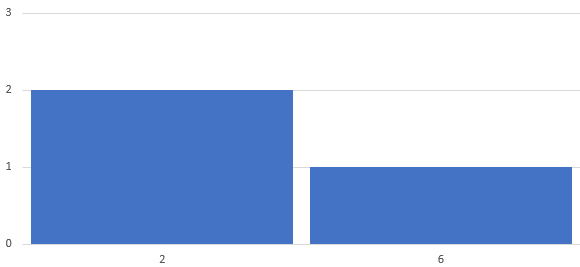
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество вершин в наборе/класс графов | Гамильтоновы и эйлеровы | Гамильтоновы и не эйлеровы | Не гамильтоновы и эйлеровы | Не гамильтоновы и не эйлеровы |
| 3 | 1 | 0 | 0 | 0 |
| 4 | 1 | 2 | 0 | 0 |
| 5 | 2 | 6 | 1 | 1 |
| 6 | 5 | 43 | 2 | 6 |
| 7 | 21 | 362 | 9 | 76 |
| 8 | 120 | 6076 | 42 | 885 |
| 9 | 1312 | 175771 | 336 | 16647 |
| 10 | 26525 | 9278593 | 3529 | 434895 |

Следующие диаграммы показывают соотношение гамильтоновости и эйлеровости на выбранных наборах графов.

В следующих гистограммах по оси Y указано количество графов, а по оси X количество гамильтоновых циклов.

4 вершины

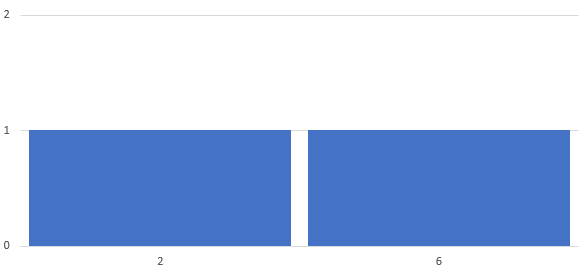
Общая статистика



Гамильтоновы и эйлеровы

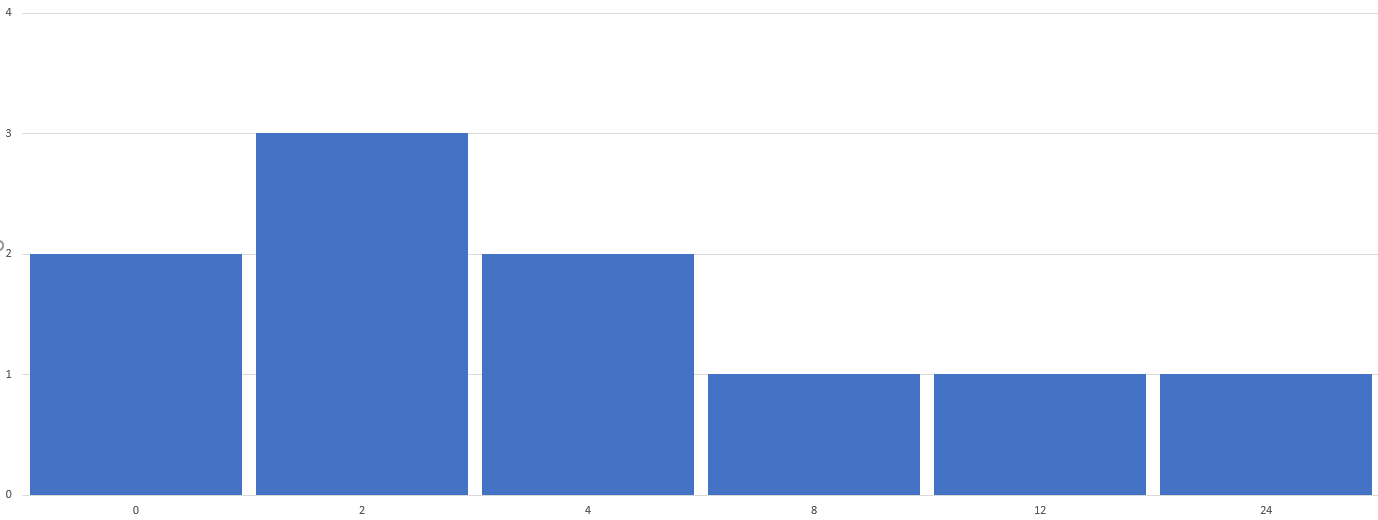


Гамильтоновы и не эйлеровы



5 вершин

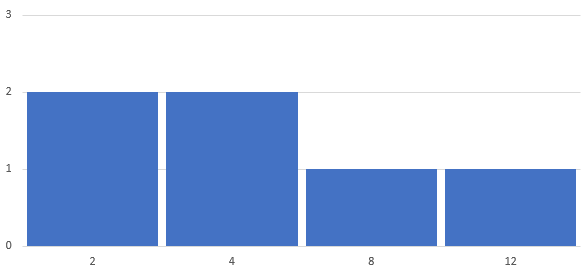
Общая статистика



Гамильтоновы и эйлеровы

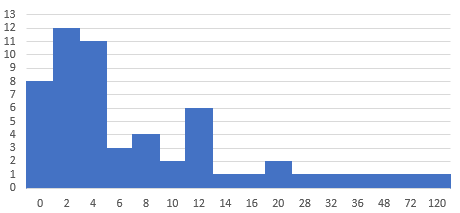


Гамильтоновы и не эйлеровы

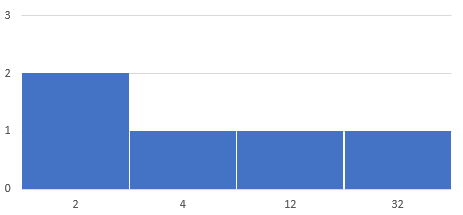


6 вершин

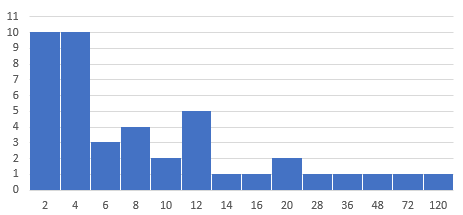
Общая статистика



Гамильтоновы и эйлеровы

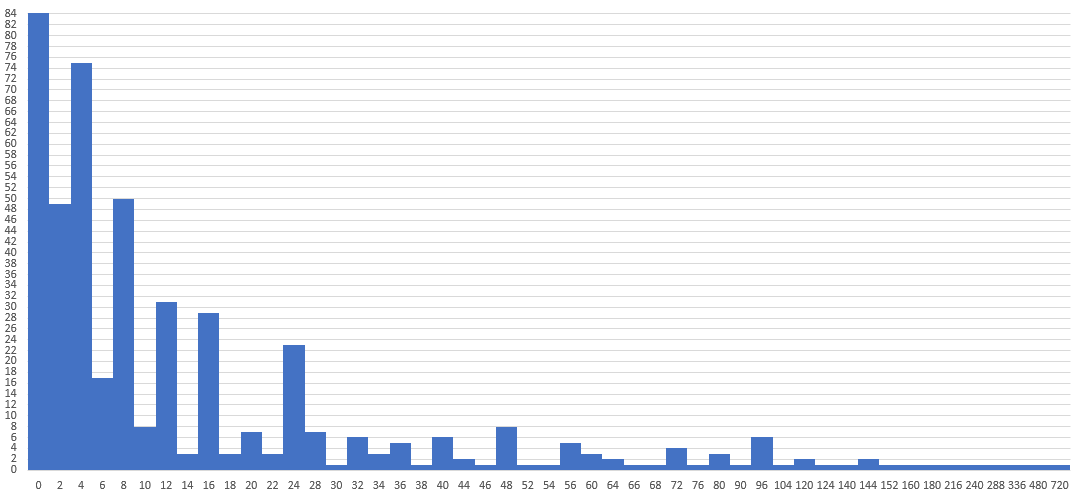


Гамильтоновы и не эйлеровы

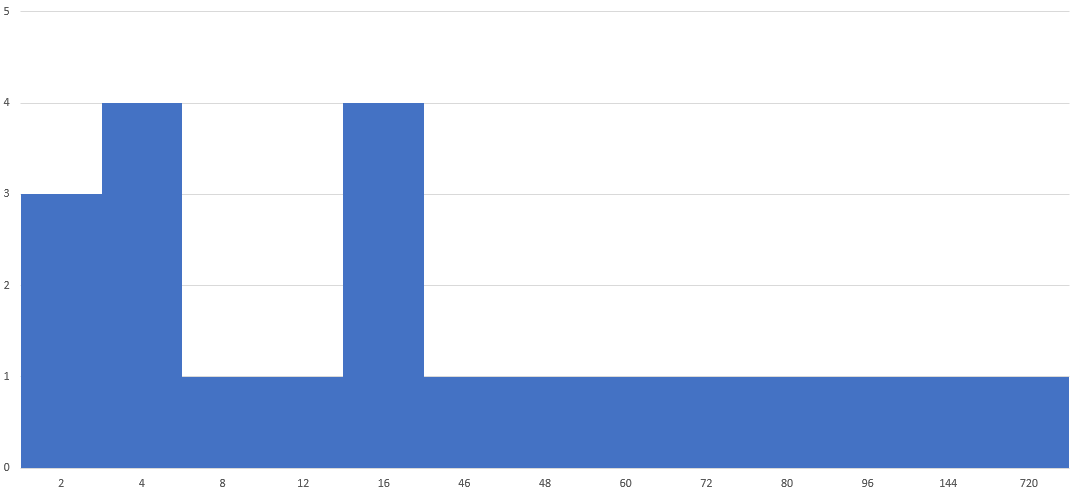


7 вершин

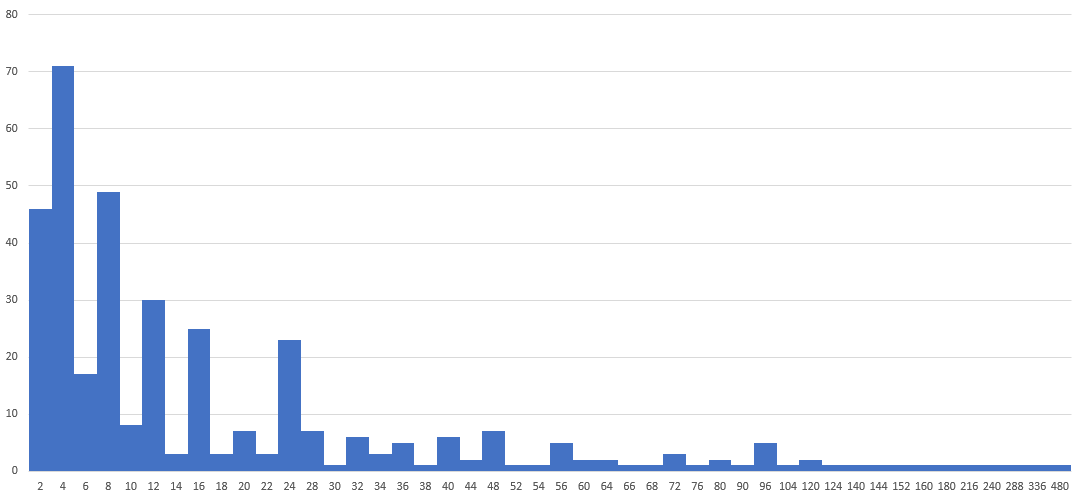
Общая статистика



Гамильтоновы и эйлеровы

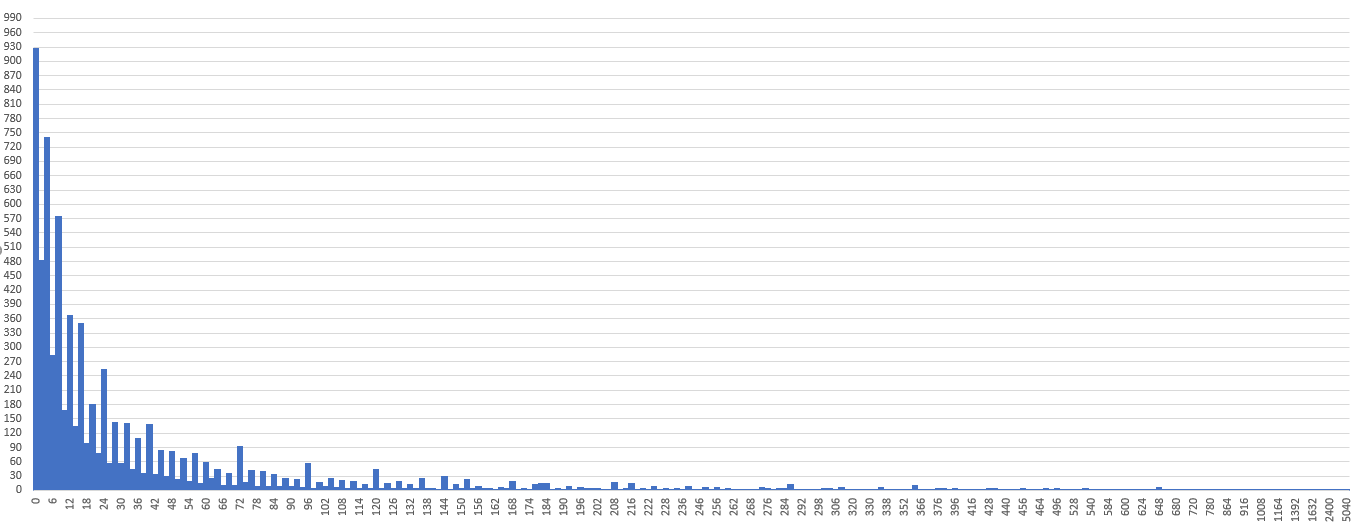


Гамильтоновы и не эйлеровы

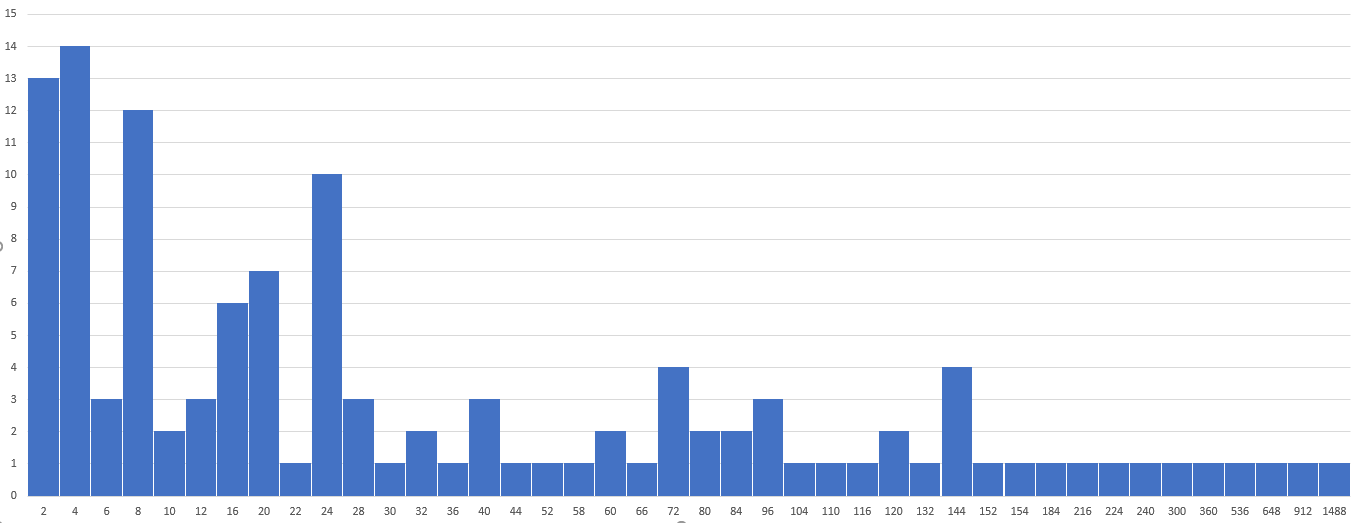


8 вершин

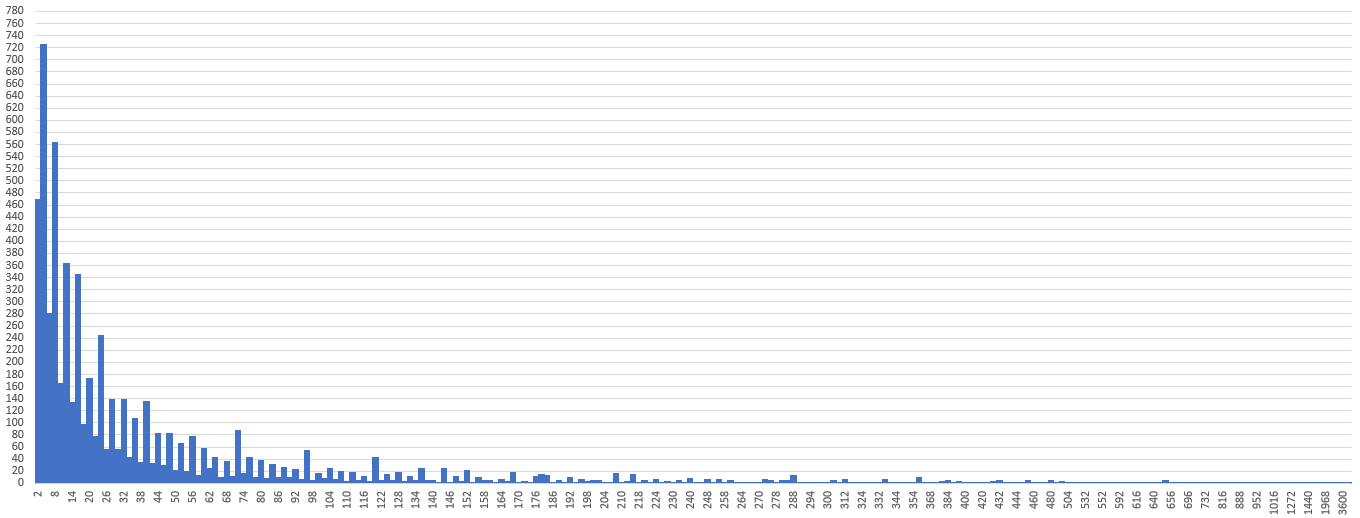
Общая статистика



Гамильтоновы и эйлеровы

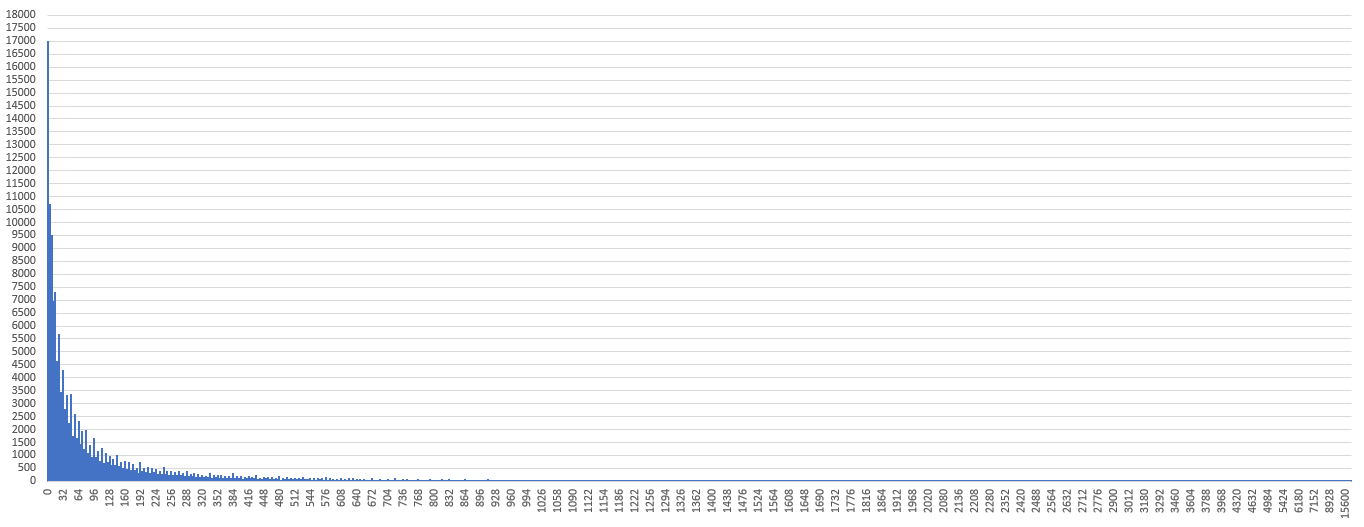


Гамильтоновы и не эйлеровы

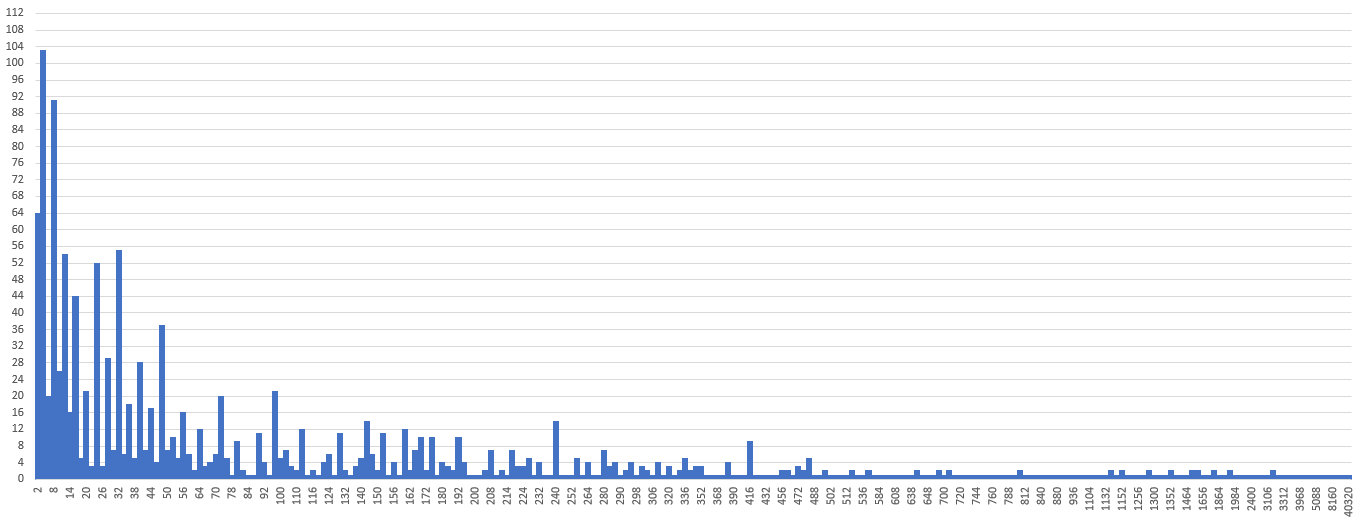


9 вершин

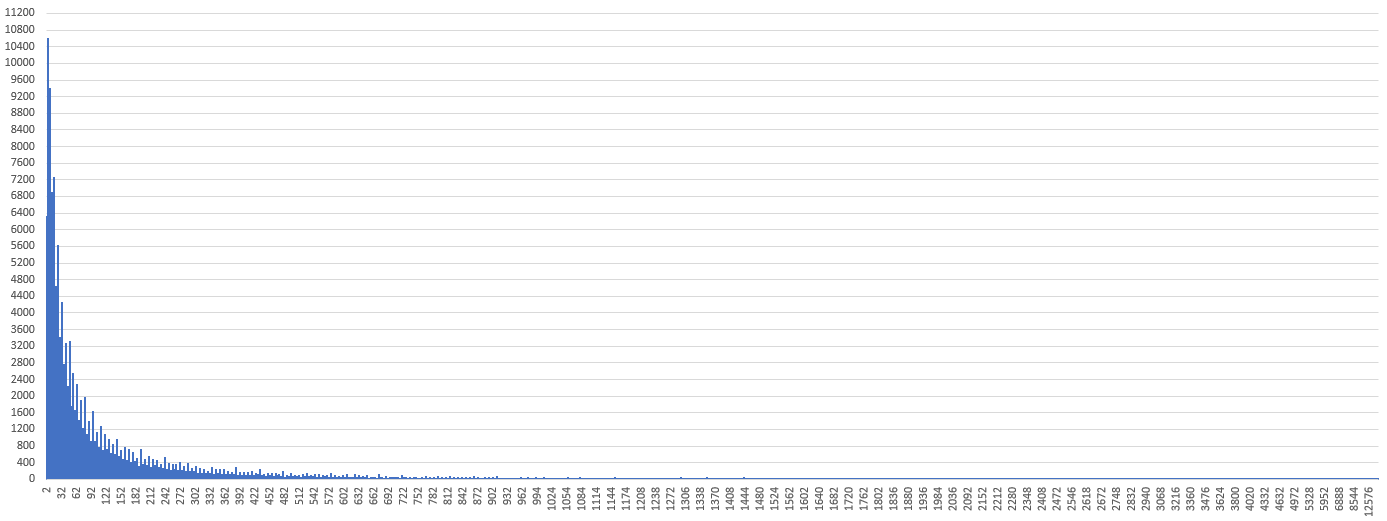
Общая статистика



Гамильтоновы и эйлеровы

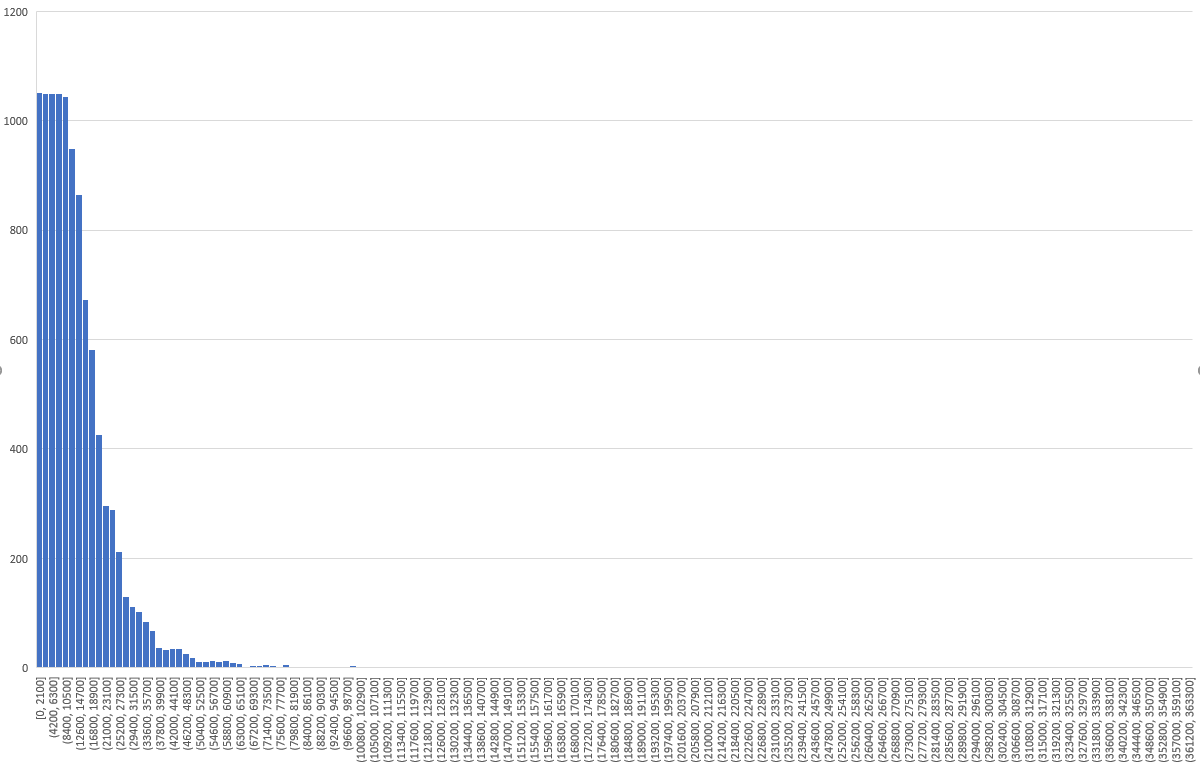


Гамильтоновы и не эйлеровы

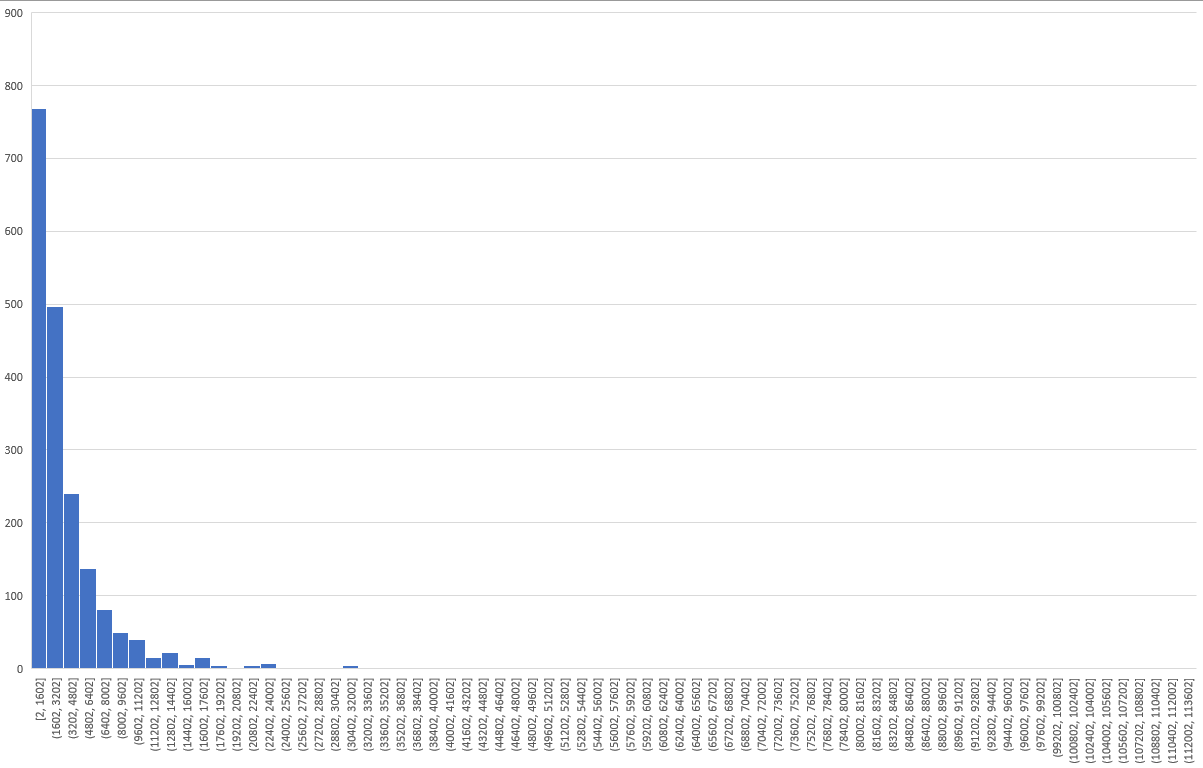


10 вершин

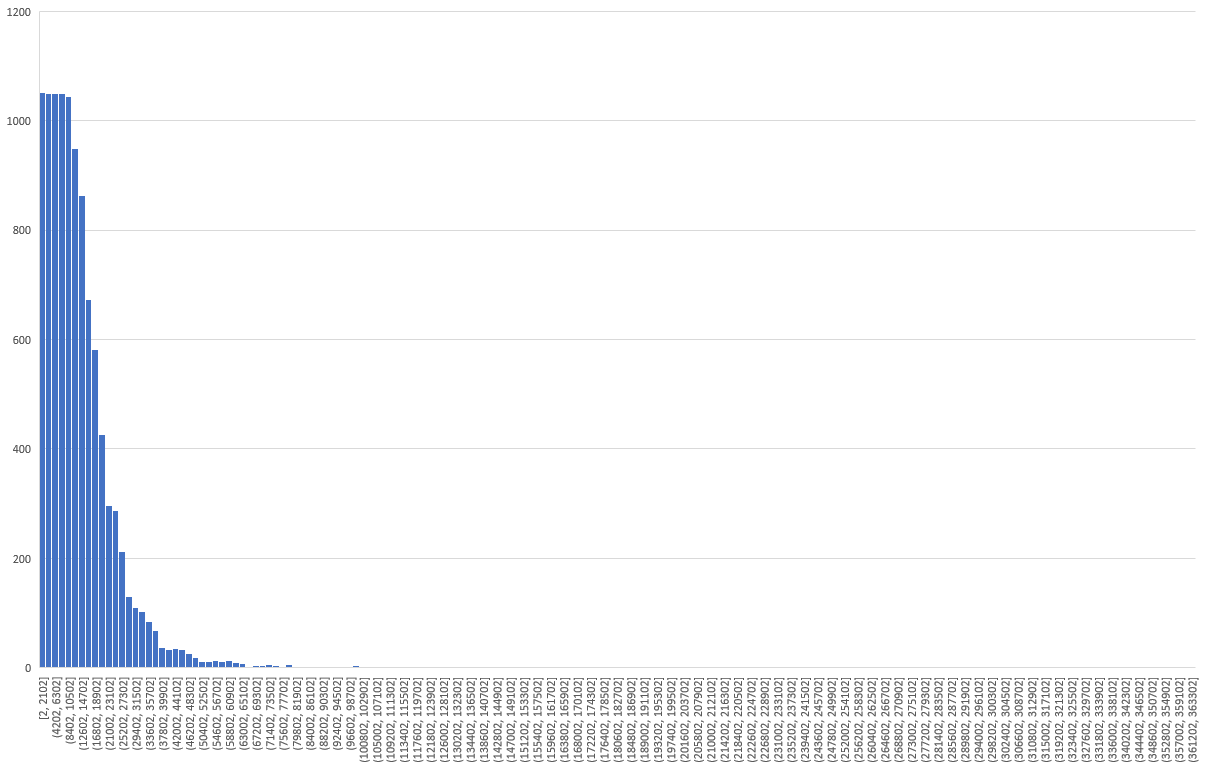
Общая статистика



Гамильтоновы и эйлеровы



Гамильтоновы и не эйлеровы



# ЗАКЛЮЧЕНИЕ

В ходе работы сгенерировали все связные неориентированные графы с количеством вершин от двух до десяти. Для всех этих графов подсчитали количество гамильтоновых циклов и проверили каждый граф на эйлеровость с помощью разработанной программы на языке C++. Проиллюстрировали графы, которые являются интересными в контексте соотношения количества ребер к количеству вершин.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Абросимов, М.Б., Долгов, А.А. Практические задания по графам, 2-е издание: Учеб. Пособие / М.Б. Абросимов, А.А. Долгов. – Саратов: Научная книга, 2009. – 76 с.

2 Поиск в глубину [Электронный ресурс] // Вики-конспекты Университета ИТМО, URL: https://neerc.ifmo.ru/wiki/index.php?title=Обход\_в\_глубину,\_цвета\_вершин (дата обращения: 02.03.2019). Загл. С экрана. Язык рус.

3 Csacademy [Электронный ресурс] // Online-editor URL: http://csacademy.com/app/graph\_editor (дата обращения: 15.05.2019). Загл. С экрана. Язык англ.

# Приложение А. Листинг программы

Файл main.cpp

#include <iostream>

#include "graph/Graph.h"

#include "data.h"

#include "task\_queue.h"

#include "utils/utils.h"

#include <thread>

#include <mutex>

#include <ctime>

#include <fstream>

#include <time.h>

#include <map>

#include <fileapi.h>

void producer(data\_t \* data) {

std::ifstream infile(data->config->filename);

if (!infile) {

std::cout << "param -f: filename is not correct" << std::endl;

raise(-1);

}

std::string s;

int cnt = 1;

for (infile >> s; !infile.eof(); infile >> s, cnt++) {

if (cnt % 1000000 == 0) {

break;

}

task\_t task = {

.task = s,

};

queue\_push(&data->queue, &task);

}

data->queue.stop\_push = 1;

{

while (!data->queue.q.empty()) {

}

//for (int i = 0; i < 4; i++)

data->queue.sem\_full.notify();

}

}

void set\_statistic(data\_t \*data, Graph \*graph, const std::string &s, int hams) {

data->hamilton\_cycles[hams]++;

if (graph->is\_euler())

data->count\_euler\_cycles++;

if (graph->is\_hamilton())

data->count\_hamilton\_cycles++;

if (graph->is\_hamilton() && graph->is\_euler()) {

data->h\_e.push\_back(s);

data->ham\_cyc\_h\_e[hams]++;

}

else if (!graph->is\_hamilton() && graph->is\_euler())

data->nh\_e.push\_back(s);

else if (graph->is\_hamilton() && !graph->is\_euler()) {

data->h\_ne.push\_back(s);

data->ham\_cyc\_h\_ne[hams]++;

}

else if (!graph->is\_hamilton() && !graph->is\_euler())

data->nh\_ne.push\_back(s);

}

void consumer(data\_t \* data, int num) {

for (;;) {

task\_t task;

if (!queue\_pop(&data->queue, &task)) {

break;

}

Graph graph(task.task);

int hams = int(graph.get\_hamilton\_paths().size());

graph.is\_euler(); // call firstly for save lock-time

{

std::lock\_guard<std::mutex> lock(data->queue.cnt\_elem\_lock);

set\_statistic(data, &graph, task.task, hams);

}

}

queue\_cancel(&data->queue);

std::cout << "thread " << num << " ended" << std::endl;

}

void thread\_mod(data\_t \* data) {

std::cout << "Thread mod start" << std::endl;

int nthreads = std::thread::hardware\_concurrency();

if(nthreads == 0)

nthreads = 2;

// nthreads = 4;

std::cout << "Count of threads: " << nthreads << std::endl;

std::vector<std::thread> threads;

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

std::thread thr(consumer, data, i);

threads.emplace\_back(std::move(thr));

}

producer(data);

data->queue.stop\_push = 1;

for(auto& thr : threads) {

thr.join();

}

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

}

void single\_mod(data\_t \* data) {

std::cout << "Single mod start" << std::endl;

std::ifstream infile(data->config->filename);

if (!infile) {

std::cout << "param -f: filename is not correct" << std::endl;

raise(-1);

}

if (data->config->input\_type == IT\_G6) {

std::string s;

for (infile >> s; !infile.eof(); infile >> s) {

Graph graph(s);

set\_statistic(data, &graph, s, graph.get\_hamilton\_paths().size());

}

}

if (data->config->input\_type == IT\_MATRIX) {

// used for testing

int n;

infile >> n;

std::vector<std::vector<int> > mat(n, std::vector<int>(n, 0));

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

for (int j = 0; j < n; ++j) {

infile >> mat[i][j];

}

}

Graph graph(mat);

std::cout << graph.is\_hamilton() << std::endl;

auto paths = graph.get\_hamilton\_paths();

if (paths.empty()) {

std::cout << "path not found" << std::endl;

return;

}

for (const auto& item: paths) {

for (int i: item) {

std::cout << i << " ";

}

std::cout << std::endl;

}

}

}

void run(data\_t \* data) {

switch (data->config->multithread\_mode) {

case TM\_MULTI:

thread\_mod(data);

break;

case TM\_SINGLE:

single\_mod(data);

break;

}

std::string context = split(std::string(data->config->filename), "/").back();

std::string p = "../result/";

std::string dir = split(context, ".")[0];

std::string path = p + dir + '/';

CreateDirectory(path.c\_str(), NULL);

std::ofstream out(path + "hamilton\_stat\_FULL.txt");

for (auto item: data->hamilton\_cycles) {

out << item.first << "\t" << item.second << std::endl;

}

out.close();

out.open(path + "hamilton\_stat\_HE.txt");

for (auto item: data->ham\_cyc\_h\_e) {

out << item.first << "\t" << item.second << std::endl;

}

out.close();

out.open(path + "hamilton\_stat\_HNE.txt");

for (auto item: data->ham\_cyc\_h\_ne) {

out << item.first << "\t" << item.second << std::endl;

}

out.close();

save\_vector\_into\_file(data->h\_e, path + "h\_e.txt"); // hamilton + euler

save\_vector\_into\_file(data->nh\_e, path + "nh\_e.txt"); // not hamilton + euler

save\_vector\_into\_file(data->h\_ne, path + "h\_ne.txt"); // hamilton + not euler

save\_vector\_into\_file(data->nh\_ne, path + "nh\_ne.txt"); // not hamilton + not euler

}

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

clock\_t start\_time = clock();

config\_t config;

data\_t data;

config\_init (&config, argc, argv);

data\_init (&data, &config);

run(&data);

std::cout << "\ntime: " << ((float)(clock() - start\_time)) / CLOCKS\_PER\_SEC << std::endl;

return 0;

}

Файл Graph.cpp

#include <iostream>

#include <algorithm>

#include "Graph.h"

#include "graph\_algorithm.h"

Graph::Graph(std::string g) {

// std::cout << "Graph constructor from string" << std::endl;

this->graph = g6\_to\_adjacency\_list(g);

this->size = this->graph.size();

}

Graph::Graph(std::vector<std::vector<int> > g) {

// Read graph from matrix to adjacency list

this->size = g.size();

this->graph = std::vector<std::vector<int> >(g.size());

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

for (int j = 0; j < g.size(); ++j) {

if (g[i][j] == 1) {

this->graph[i].push\_back(j);

// this->graph[j].push\_back(i);

}

}

}

}

Graph::Graph(Graph \*g) {

this->graph = g->graph;

this->size = g->size;

this->\_is\_hamilton = -1;

this->\_is\_euler = -1;

}

Graph::~Graph() = default;

std::vector<std::vector<int> > Graph::g6\_to\_matrix(std::string &g6) {

// std::cout << "Call function g6\_to\_matrix" << std::endl;

int n = (int(g6[0]) - 63);

std::vector<std::vector<int> > result(n, std::vector<int>(n, 0));

int row = 0;

int col = 1;

for (int i = 1; i < g6.size(); i++) {

int ch = g6[i] - 63;

int bit = 0;

for (int ibit = 5; ibit >= 0; --ibit) {

bit = (ch >> ibit) & 1;

result[row][col] = bit;

result[col][row] = bit;

row++;

if (row == col) {

row = 0;

col++;

}

if (col == n)

return result;

}

}

return std::vector<std::vector<int> >();

}

std::vector<std::vector<int> > Graph::g6\_to\_adjacency\_list(std::string &g6) {

// std::cout << "Call function g6\_to\_adjacency\_list" << std::endl;

int n = (int(g6[0]) - 63);

std::vector<std::vector<int> > result(n);

int row = 0;

int col = 1;

for (int i = 1; i < g6.size(); i++) {

int ch = g6[i] - 63;

for (int ibit = 5; ibit >= 0; --ibit) {

if ((ch >> ibit) & 1) {

// printf("here1\n");

result[row].push\_back(col);

result[col].push\_back(row);

// printf("here2\n");

}

row++;

if (row == col) {

row = 0;

col++;

}

if (col == n) {

return result;

}

}

}

return std::vector<std::vector<int> >();

}

std::vector<std::vector<int> > Graph::adjacency\_list\_to\_matrix(std::vector<std::vector<int> > &graph) {

// std::cout << "Call adjacency\_list\_to\_matrix" << std::endl;

std::vector<std::vector<int> > result(graph.size(), std::vector<int>(graph.size()));

for (int row = 0; row < graph.size(); ++row) {

for (int col: graph[row]) {

result[row][col] = 1;

result[col][row] = 1;

}

}

return result;

}

std::vector<std::vector<int> > Graph::matrix\_to\_adjacency\_list(std::vector<std::vector<int> > &graph) {

// std::cout << "Call matrix\_to\_adjacency\_list" << std::endl;

std::vector<std::vector<int> > result(graph.size(), std::vector<int>());

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

for (int j = 0; j < graph.size(); ++j) {

if (graph[i][j]) {

result[i].push\_back(j);

result[j].push\_back(i);

}

}

}

return result;

}

/\*

\* Check for Euler

\*/

bool Graph::is\_euler() {

if (this->\_is\_euler != -1)

return bool(this->\_is\_euler);

// 1. Количество вершин с нечетной степенью равно нулю.

for (const auto &vec: this->graph) {

if (vec.size() % 2 == 1) {

this->\_is\_euler = 0;

return false;

}

}

// 2. Все компоненты связности кроме, может быть одной, не содержат ребер.

int flag = false;

for (auto comp\_size: this->components()) {

if (comp\_size != 0) {

if (flag) {

this->\_is\_euler = 0;

return false;

}

flag = true;

}

}

this->\_is\_euler = 1;

return true;

}

int get\_edge\_count(Graph \*graph) {

int result = 0;

for (const auto& item: graph->graph) {

result += item.size();

}

return result / 2;

}

void euler\_paths(int v, std::vector<std::vector<int> > \*graph, std::vector<int> \*current\_path, int &total\_edge\_count, int current\_edge\_count, std::vector<std::vector<int> > \*paths) {

if (total\_edge\_count == current\_edge\_count) {

// Если 1 == back и путь состоит из всех вершин и замыкается на первой

paths->push\_back(\*current\_path);

return;

}

for (int u = 0; u < graph->size(); u++) {

if ((\*graph)[v][u] == 1) {

(\*graph)[v][u] = 0;

(\*graph)[u][v] = 0;

current\_path->push\_back(u);

euler\_paths(u, graph, current\_path, total\_edge\_count, current\_edge\_count + 1, paths);

current\_path->pop\_back();

(\*graph)[v][u] = 1;

(\*graph)[u][v] = 1;

}

}

}

std::vector<std::vector<int> > Graph::get\_euler\_paths() {

std::vector<std::vector<int> > result;

if (this->is\_euler()) {

std::vector<int> current\_path(1, 0);

std::vector<std::vector<int> > g = this->adjacency\_list\_to\_matrix(this->graph);

int total\_edge\_count = get\_edge\_count(this);

euler\_paths(0, &g, &current\_path, total\_edge\_count, 0, &result);

}

return result;

}

std::vector<int> Graph::components() {

std::vector<bool> used(this->size, false);

std::vector<int> result;

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

if (!used[i]) {

int deep = galgo::dfs(this, i, &used);

result.push\_back(deep);

}

}

return result;

}

/\*

\*\*\* Check for Hamilton \*\*\*

\*/

std::vector<int> sequence\_degree\_vertex(Graph \*graph) {

/\*

\* return: sorted vector with number of vertex | deg(result[i]) > deg(result[j]) : i < j

\* \*/

std::vector<int> result;

std::vector<std::pair<int, int> > pair\_result;

for (int idx = 0; idx < graph->size; ++idx) {

auto &vlist = graph->graph[idx];

std::pair<int, int> pair = std::make\_pair(vlist.size(), idx);

pair\_result.push\_back(pair);

}

sort(pair\_result.begin(), pair\_result.end());

result.reserve(pair\_result.size());

for (auto p: pair\_result) {

result.push\_back(p.second);

}

reverse(result.begin(), result.end());

return result;

}

std::vector<int> get\_not\_adjacent\_vertexes(Graph \*graph, int v) {

std::vector<int> result;

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

bool candidate = true;

for (auto ne: graph->graph[v]) {

if (i == v or ne == i) {

candidate = false;

break;

}

}

if (candidate) {

result.push\_back(i);

}

}

return result;

}

Graph graph\_closure(Graph \*graph) {

/\*

\* Source: http://freeusermanuals.com/backend/web/manuals/1521810604HamiltonBondyAndChvatal.pdf

\* \*/

Graph closure(graph);

for (int j = 0; j < graph->size; ++j) {

bool stop = true;

for (auto v\_source: sequence\_degree\_vertex(&closure)) {

std::vector<int> not\_adjacent = get\_not\_adjacent\_vertexes(&closure, v\_source);

for (auto v\_destination: not\_adjacent) {

// condition number 3: deg(v\_destination) + deg(v\_source) >= |graph.vertexes|

if (closure.graph[v\_destination].size() + closure.graph[v\_source].size() >= closure.size) {

closure.graph[v\_source].push\_back(v\_destination);

closure.graph[v\_destination].push\_back(v\_source);

stop = false;

}

}

if (!stop)

break;

}

if (stop)

break;

}

return closure;

}

Graph Graph::get\_closure() {

return graph\_closure(this);

}

bool dirac(Graph \*graph) {

if (graph->size >= 3) {

for (const auto &vertex: graph->graph) {

if (vertex.size() < graph->size / 2) {

return false;

}

}

}

return true;

}

bool th\_ore(Graph \*graph) {

if (graph->size >= 3) {

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

auto &current\_vertex = graph->graph[i];

for (int j = 0; j < graph->size; ++j) {

if(std::find(current\_vertex.begin(), current\_vertex.end(), j) == current\_vertex.end()) {

if (current\_vertex.size() + graph->graph[j].size() < graph->size) {

return false;

}

}

}

}

}

return true;

}

bool th\_posh(Graph \*graph) {

if (graph->size >= 3) {

for (int k = 0; k < graph->size / 2; ++k) {

for (const auto &vertex: graph->graph) {

if (vertex.size() < k + 1) {

return false;

}

}

}

}

return true;

}

bool th\_bondi\_chvatal(Graph \*graph) {

if (graph->size >= 3) {

Graph closure = graph\_closure(graph);

if (dirac(&closure) or th\_ore(&closure)) {

return true;

}

}

return false;

}

bool Graph::is\_hamilton() {

/\*

\* This function give't a true hamilton stat!

\* call this function after get\_hamilton\_paths() for true result

\*/

if (this->\_is\_hamilton != -1)

return bool(this->\_is\_hamilton);

this->\_is\_hamilton = th\_bondi\_chvatal(this);

return bool(this->\_is\_hamilton);

}

void hamilton\_paths(int v, Graph \*graph, std::vector<int> \*current\_path, std::vector<bool> \*used,

std::vector<std::vector<int> > \*paths) {

if (current\_path->size() == graph->size + 1 && (\*current\_path)[0] == current\_path->back()) {

// Если 1 == back и путь состоит из всех вершин и замыкается на первой

paths->push\_back(\*current\_path);

return;

}

(\*used)[v] = true;

for (int i = 0; i < graph->graph[v].size(); ++i) {

auto &item = graph->graph[v][i];

if (!(\*used)[item] || (current\_path->size() == graph->size && item == (\*current\_path)[0])) {

current\_path->push\_back(item);

hamilton\_paths(item, graph, current\_path, used, paths);

current\_path->pop\_back();

}

}

(\*used)[v] = false;

}

std::vector<std::vector<int> > Graph::get\_hamilton\_paths() {

std::vector<std::vector<int> > result;

std::vector<int> current\_path(1, 0);

std::vector<bool> used(this->size);

hamilton\_paths(0, this, &current\_path, &used, &result);

this->\_is\_hamilton = result.empty() ? 0: 1;

return result;

}

void Graph::print() {

std::vector<std::vector<int> > g = this->adjacency\_list\_to\_matrix(graph);

for (int i = 0; i < g.size(); ++i, std::cout << std::endl) {

for (int j = 0; j < g.size(); ++j) {

std::cout << g[i][j] << " ";

}

}

}

Файл data.cpp

#include "data.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <getopt.h>

int config\_init (config\_t \* config, int argc, char \* argv[])

{

config->multithread\_mode = TM\_SINGLE;

int correct = 0;

const char \*opt\_string = "msf:t:";

for (;;) {

int opt = getopt(argc, argv, opt\_string);

if (-1 == opt)

break;

switch (opt) {

case 'm':

config->multithread\_mode = TM\_MULTI;

break;

case 's':

config->multithread\_mode = TM\_SINGLE;

break;

case 'f':

config->filename = optarg;

correct = 1;

break;

case 't':

if (strcmp(optarg, "g6") == 0)

config->input\_type = IT\_G6;

if (strcmp(optarg, "mat") == 0)

config->input\_type = IT\_MATRIX;

if (strcmp(optarg, "list") == 0)

config->input\_type = IT\_ADJACENCY;

break;

default:

printf("Incorrect arguments.\n! -f = filename with graphs\n -m = multithread mod\n -s = single mod\n -t = [g6, mat, list] graph format in input file\n");

return (EXIT\_FAILURE);

}

}

if (correct == 0) {

printf("Incorrect arguments.\n! -f = filename with graphs\n -m = multithread mod\n -s = single mod\n -t = [g6, mat, list] graph format in input file\n");

exit(EXIT\_FAILURE);

}

return (EXIT\_SUCCESS);

}

int config\_create (config\_t \* config) {

config->multithread\_mode = TM\_SINGLE;

return (EXIT\_SUCCESS);

}

int data\_init (data\_t \* data, config\_t \* config) {

data->config = config;

queue\_init (&data->queue);

data->queue.stop\_pop = data->queue.stop\_push = 0;

printf("%s\n", "config and data init successful");

return (EXIT\_SUCCESS);}

# Приложение Б. Каталог графов

В данном разделе на рисунках 1-23 приведены иллюстрации графов, представляющих из себя единичные случаи, которые выявлены после анализа таблиц из раздела 4.

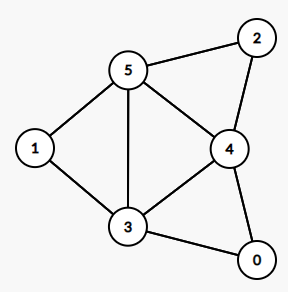


Рисунок 1 – Эйлеров и гамильтонов граф (EElw)

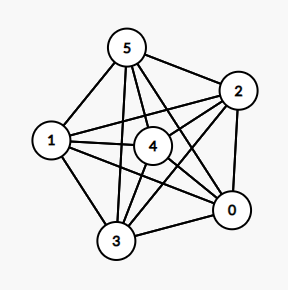


Рисунок 2 – Гамильтонов и не эйлеров граф (E~~w)

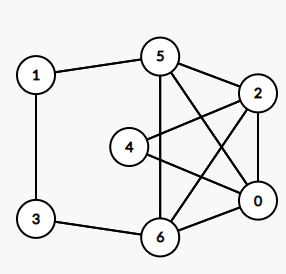


Рисунок 3 – Гамильтонов и эйлеров граф (FQjdg)

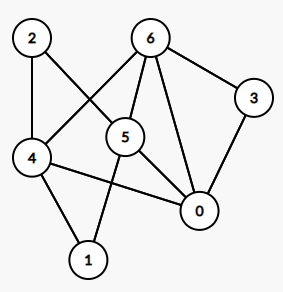


Рисунок 4 – Не гамильтонов и эйлеров граф (FCzcw)

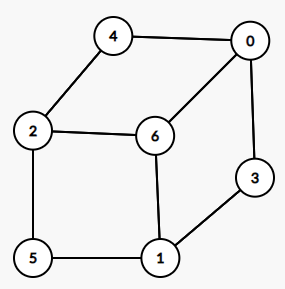


Рисунок 5 – Не гамильтонов и не эйлеров граф (FEhf?)

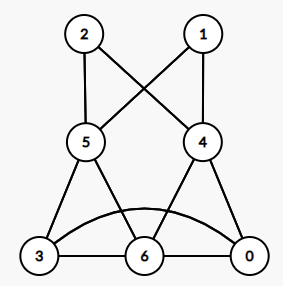


Рисунок 6 – Не гамильтонов и не эйлеров (FCxsw)

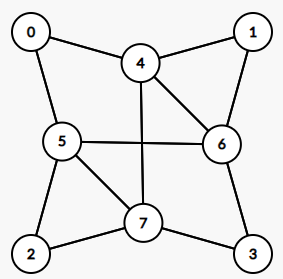


Рисунок 7 – Гамильтонов и эйлеров граф (G?qaxw)

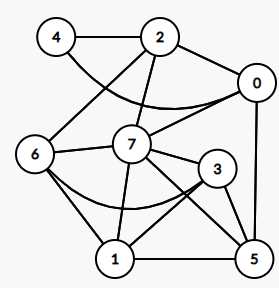


Рисунок 8 – Гамильтонов и эйлеров граф (GQjRfk)

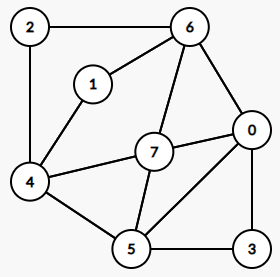


Рисунок 9 – Не гамильтонов и эйлеров (GCY^C[)

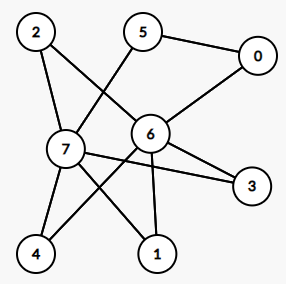


Рисунок 10 – Не гамильтонов и не эйлеров (G?AFrw)

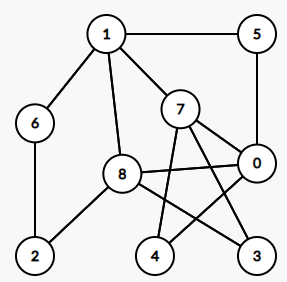


Рисунок 11 – Гамильтонов и эйлеров (H?bBEro)

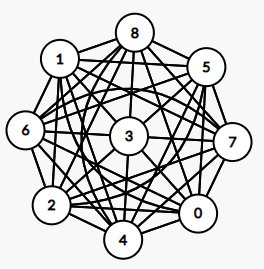


Рисунок 12 – Гамильтонов и эйлеров (H~~~~~~)

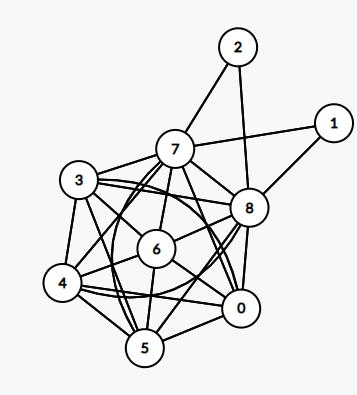


Рисунок 13 – Не гамильтонов и эйлеров (HCe[~~~)

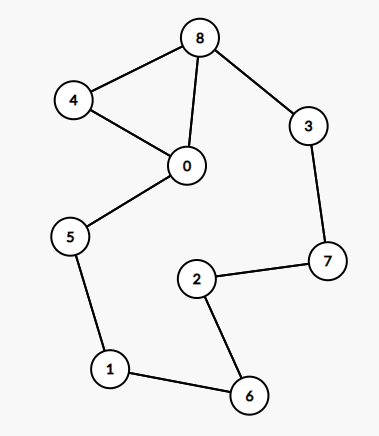


Рисунок 14 – Гамильтонов и не эйлеров (H?bB@aW)

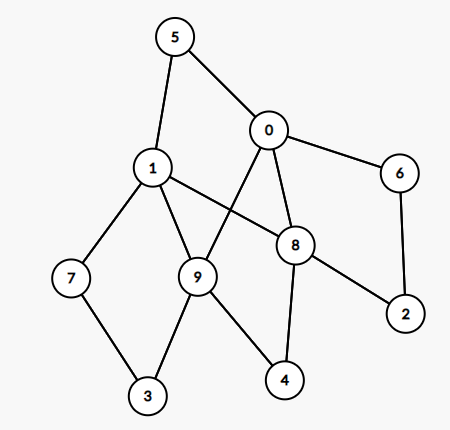


Рисунок 15 – Гамильтонов и эйлеров (I?BDAbgu?)

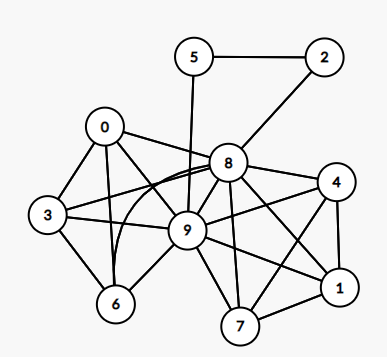


Рисунок 16 – Не гамильтонов и эйлеров (ICOcaRzvw)

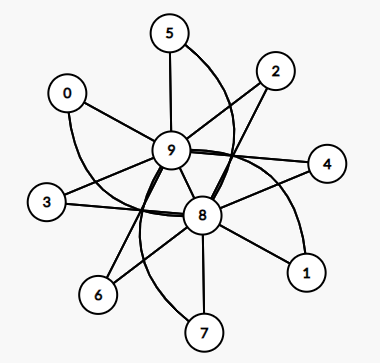


Рисунок 17 – Не гамильтонов и не эйлеров (I????B~~w)

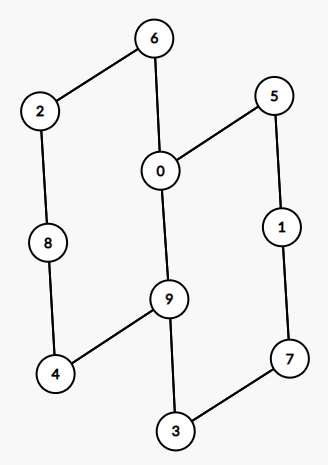


Рисунок 18 – Гамильтонов и не эйлеров (I?BDA\_ge?)

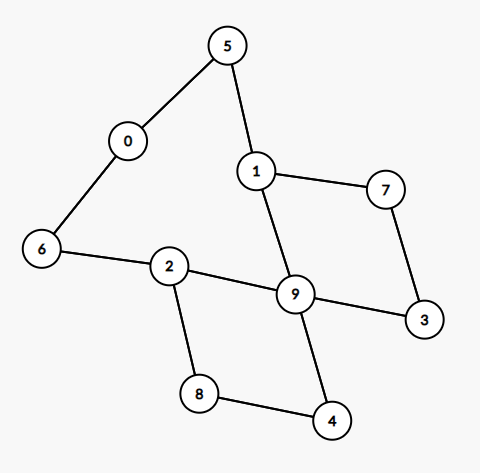


Рисунок 19 – Гамильтонов и не эйлеров (I?BDA\_g]?)

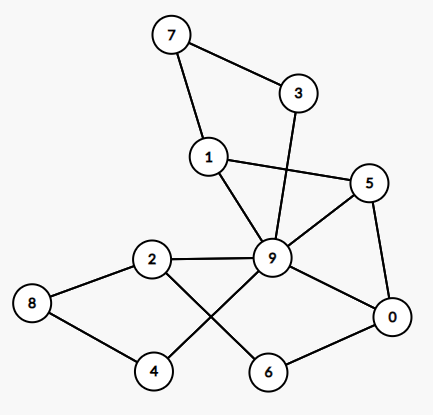


Рисунок 20 – Гамильтонов и не эйлеров (I?BDA\_g~?)

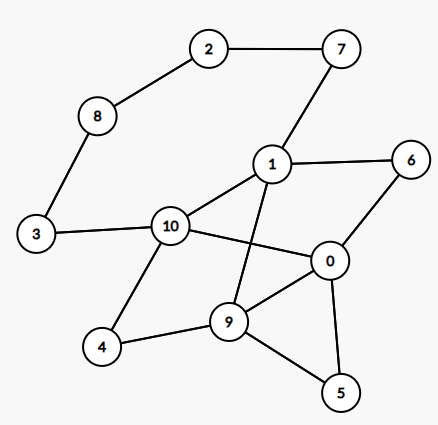


Рисунок 21 – Гамильтонов и эйлеров (J?AEB?orEo?)

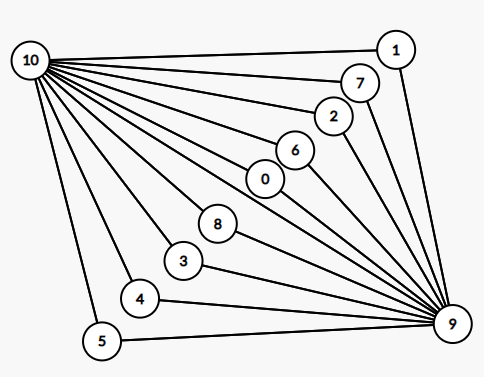


Рисунок 22 – Не гамильтонов и эйлеров (J??????~~~\_)

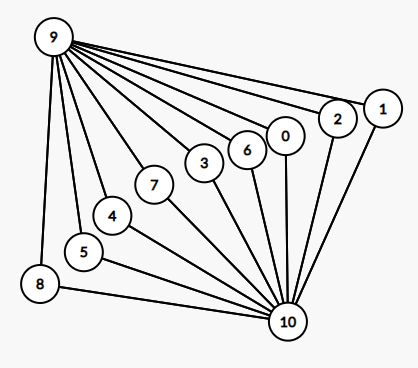


Рисунок 23 – Не гамильтонов и не эйлеров (J??????~~~?)