**Лабораторная C0909**

Прудников Евгений

[**Запуск теста graph500 на ноутбуке** 2](#_Toc463718959)

[**Модель веб графа Боллобаша–Риордана** 3](#_Toc463718960)

[**Пример получившегося веб графа (n = 10)** 3](#_Toc463718961)

[**Характеристики получившегося графа для n = 4000, m =12** 3](#_Toc463718962)

[**Выводы и наблюдения** 4](#_Toc463718963)

[**Литература** 4](#_Toc463718964)

[**Приложение** 5](#_Toc463718965)

# **Запуск теста graph500 на ноутбуке**

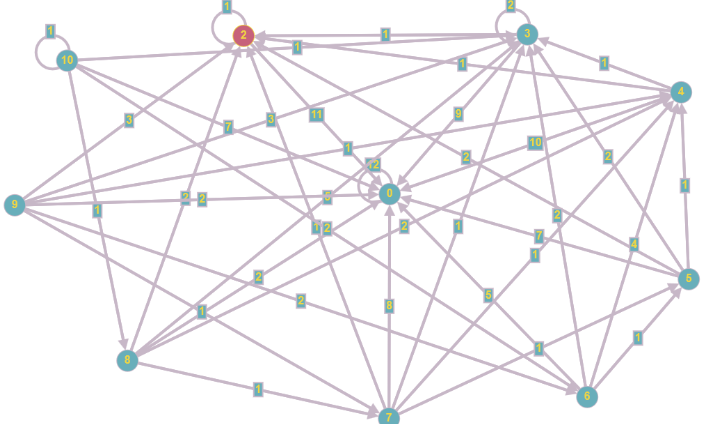


После запуска реализации для MPI построился граф с 212 вершин и 212 \* 16 (65536) ребрами, далее были выбраны 64 случайные вершины и производился поиск в ширину. На моем ноутбуке поиск в ширину проходил со скоростью 65536 ребер/0.004924 секунд ~ 13309504 ребер/секунду

# **Модель веб графа Боллобаша–Риордана**

Был написан генератор веб графа по модели **Боллобаша–Риордана** (код в приложении)**.**

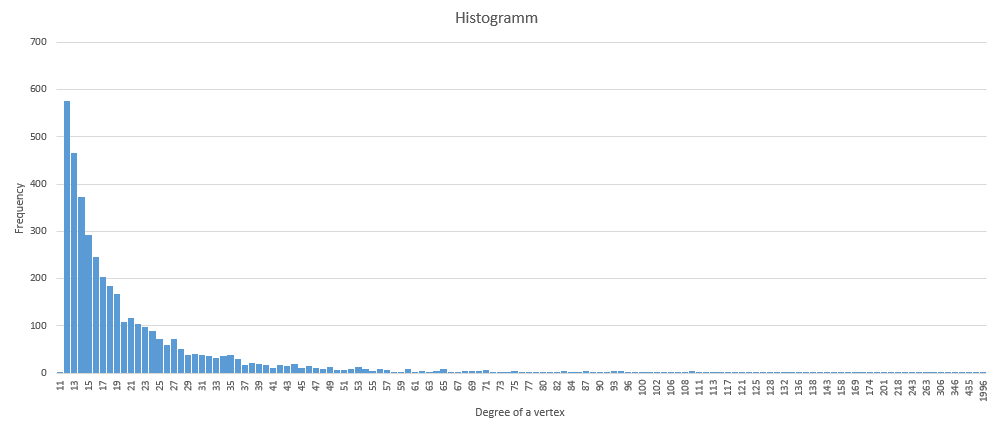
## **Пример получившегося веб графа (n = 10)**



## **Характеристики получившегося графа для n = 4000, m =12**

*При запуске 1000+ раз характеристики не будут сильно отличатся*

* Диаметр: 4
* Оценка диаметра в модели: 3.9205
* Количество компонент связности: 1 (Для всех вершин получилась одна компонента связности)
* Среднее расстояние в каждой компоненте связности: 2.72421
* Гистограмма # (n, d):



* Вероятность быть связным:

Вероятность быть связным можно попробовать оценить следующим образом:

Для того чтобы граф был несвязным (Pнс) и появилась новая компонента связности, необходимо чтобы m подряд идущих новых вершин в случайном графе на первом этапе построения имели петлю, тогда при «схлопывании» графа в веб граф группа вершин v {mn + 1, …, nm + m} будет иметь только петлю и не будет связана с другими вершинами, а также каждая последующая вершина (j) была не связана с этой группой вершин. Так как появление каждой новой вершины и её связь независимое (зависит только от количества вершин в графе, и не зависит от исхода прошлой вершины) событие вероятность этого равна Pнс = Pm (что будет m петель подряд) \* Pg (вероятность каждой последующей точки не соединиться с группой из m петель)

Тогда вероятность быть связным Pc = 1 – Pнс. Видно, что при увеличении n вероятности быть несвязным падают, следовательно, вероятность быть связным стремится к 1.

# **Выводы и наблюдения**

Построенный в задании граф достаточно хорошо моделирует модель интернета, однако модель не идеальная и требует уточнений.

# **Литература**

* Модели случайных графов и их применения. А.М. Райгородский
* Математические модели интернета. А. М. Райгородский «Квант» №4, 2012

# **Приложение**

В представленном коде сначала генерируется случайный граф по правилам модели, а затем схлопывается в m раз сохраняя при этом все связи и получается модель веб графа. Случайный граф хранится в виде вектора, где индекс - номер вершины, значение – номер вершины соединения. Итоговый веб граф храниться в виде матрицы смежности. Далее рассчитываются параметры получившегося графа (диаметр, среднее расстояние между вершинами в одной компоненте связности и т.д.)

Степени вершин хранятся в векторе verticesOfWebGraph по ним строится гистограмма # (n, d)

**Вывод программы:**

n = 4000

m = 12

Connected Components Count: 1

Component 0 avg dist: 2.72421

webGraph Diam: 4

web Graph estimated diam: 3.92054

done.

**Исходный код:**

// graphgen.cpp : Defines the entry point for the console application.

//

#include <vector>

#include <random>

#include <iostream>

#include <queue>

#include <ctime>

#include <math.h>

using namespace std;

const int M = 12;

const int N = 4000;

const int NM = N\*M;

void GenerateRandomGraph(vector<int> &g, vector<int> &v);

void GenerateWebGraph(vector<int> &g, vector<int> &v, vector<vector<int>> &wg, vector<int> &wv);

int FindConnCompnts(vector<vector<int>> & wg, vector<int> & connectedComponents);

void FloydWarshallAlgorithm(vector<vector<int>> & wg, vector<vector<int>> & dists);

int FindDiamAvgDistsComponents(vector<int> & connectedComponents, int componentsCount

,vector<vector<int>> & dists, vector<double> & avgDists);

int main()

{

vector<int> verticesOfRandomGraph(NM);

vector<int> randomGraph(NM);

vector<int> verticesOfWebGraph(N); // for #(n,d) histogramm

vector<vector<int>> webGraph(N, vector<int>(N, 0));

vector<int> connectedComponents(N, -1);

int componentsCount;

vector<vector<int>> dists(N, vector<int>(N, 0));

int webGraphDiam;

randomGraph[0] = 0; // first vertice with loop

verticesOfRandomGraph[0] = 2; // # of edges in first vertice

{

GenerateRandomGraph(randomGraph, verticesOfRandomGraph);

GenerateWebGraph(randomGraph, verticesOfRandomGraph, webGraph, verticesOfWebGraph);

}

componentsCount = FindConnCompnts(webGraph, connectedComponents);

vector<double> avgDists(componentsCount);

FloydWarshallAlgorithm(webGraph, dists);

webGraphDiam = FindDiamAvgDistsComponents(connectedComponents, componentsCount, dists, avgDists);

{

cout << "n = " << N << endl;

cout << "m = " << M << endl;

cout << "Connected Components Count: " << componentsCount << endl;

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

cout << "Component " << i << " avg dist: " << avgDists[i];

cout << endl << "webGraph Diam: " << webGraphDiam << endl;

cout << "web Graph estimated diam: " << log(N) / log(log(N))<< endl;

cout << endl << "done." << endl;

}

getchar();

return 0;

}

void FloydWarshallAlgorithm(vector<vector<int>> & wg , vector<vector<int>> & dists)

{

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

{

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

{

dists[i][j] = wg[i][j] > 0 ? 1 : N\*N;

}

}

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

{

dists[i][i] = 0;

}

cout << endl;

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

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

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

dists[i][j] = min(dists[i][j], dists[i][k] + dists[k][j]);

wg.clear();

}

int FindDiamAvgDistsComponents(vector<int> & connectedComponents, int componentsCount, vector<vector<int>> & dists,vector<double> & avgDists)

{

int d = 0;

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

{

int comp\_dist = 0;

int comp\_dist\_n = 0;

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

{

if (connectedComponents[j] != i)

{

continue;

}

for (int k = j + 1; k < N; k++)

{

if (connectedComponents[k] != i) {

continue;

}

comp\_dist += dists[j][k];

++comp\_dist\_n;

if (d < dists[j][k])

{

d = dists[j][k];

}

}

}

avgDists[i] = comp\_dist / static\_cast<double>(comp\_dist\_n);

}

return d;

}

int FindConnCompnts(vector<vector<int>> & wg, vector<int> & connectedComponents)

{

int componentsCount = 0;

queue <int> q;

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

{

if (connectedComponents[i] != -1)

continue;

q.push(i);

while (!q.empty())

{

int v = q.front();

q.pop();

if (connectedComponents[v] != -1)

continue;

connectedComponents[v] = componentsCount;

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

{

if (wg[j][v] > 0 && connectedComponents[j] == -1)

{

q.push(j);

}

}

}

componentsCount++;

}

return componentsCount;

}

void GenerateRandomGraph(vector<int> &g, vector<int> &v)

{

srand((unsigned int)time(NULL));

for (int i = 1; i < NM; i++)

{

double r = (double)rand() / RAND\_MAX;

double ni = i + 1;

double probabilityInN = 1 / (2 \* ni - 1);

if (r < probabilityInN)

{

g[i] = i;

v[i] = 2;

}

else

{

double probabilityInV = probabilityInN;

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

{

probabilityInV += v[k] / (2 \* ni - 1.0);

if (r <= probabilityInV)

{

g[i] = k;

v[i] += 1;

v[k] += 1;

break;

}

}

}

}

}

void GenerateWebGraph(vector<int> &g, vector<int> &v, vector<vector<int>> &wg, vector<int> &wv)

{

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

{

wg[i / M][g[i] / M] += 1;

wg[g[i] / M][i / M] += 1;

wv[i / M] += v[i];

}

g.clear(); v.clear();

}