Frontier Research Topics in Network Science Experimental Report

Emergence of Scaling in Random Networks

Zhongyuan Chen

Scaleless-network implementation using C++

Hangzhou Dianzi University
School of Cybersecurity
2019-3-29

Contents

1 Overview

This paper introduces the concept of scale-free power-law distributed networks and offers code implementation.

1.1 Abstract

Scale-free power-law distributed networks are described in detail by Albert-Laszlo Barabasi and Reka Albert[1]. In this paper, we dive into Barabasi's theories and algorithms and further offers a simple implementation of such networks.

We mainly focus on two features of such networks: **preferential attachment** and **growth**. Either of these ingredients will be integrated into the model for simulation.

We use C++ to generate several networks consists of over 200,000 vertices. We see roughly a line when we apply logarithmic function to the results. Thus we verify the power-law distribution of scale-free networks.

1.2 Background

Many networks have been reported to be scale-free, e.g. the World Wide Web and the citation pattern in science. Thus researchers begin to think about their shared characteristics and build related models. Barabasi and Albert proposed a generative mechanism to explain the appearance of power-law distributions for younger generations, namely us, to learn a bit about scale-free networks.

2 Important Concepts

Several concepts appear frequently in this paper are introduced here.

2.1 Scale-free network

A scale-free network is a network whose degree distribution follows a power law, at least asymptotically[2]. That is, the fraction $P(k)$ of nodes in the network

having k connections to other nodes goes for large values of k as:

$$P(k) \sim k^{-\gamma} \tag{1}$$

where γ is a parameter whose value is typically in the range $2 < \gamma < 3$.

2.2 Growth

Scale-free networks expand continuously by the addition of new vertices. Our modeled networks shall behave similarly to real networks like WWW. Thus they can never be without growth.

2.3 Preferential Attachment

When a scale-free network grows, new vertices attach preferentially to sites that are already well connected. The more the degree of a vertex is, the easier it get connected by a new vertex. This feature is consistent to Matthew Effect or "the rich get richer and the poor get poorer".

3 Model Description

Our model is basically a network with vertices and edges. We are allowed to add ingredients into it and see how it behaves. The ingredients available are growth and preferential attachment, which are covered in detail in section 2.2 and section 2.3.

Networks with the growth ingredient grow dynamically. New vetices arrives continuously to make the networks larger and richer. Each new coming vertex will connect to several old vertices thus form new edges. Networks without the growth ingredient does not grow, but randomly choose an existing vertex to connect to other non-neighbor vertices.

Networks with the preferential attachment orders the connecting vertex to connect preferentially to existing vertices. Which means, we alter the probability of being connected according to degrees. Networks without this ingredient just ask its vertices to connect with equal probability.

4 Code Implementation

We are essentially budding computer scientists. Networks grow by adding vetices while we ourselves grow by designing and writing codes.

We guess programmers would like to view the code first rather than walk through tedious text. As the saying "Talk is cheap, show me the code" goes. The definition of our modeled network Graph is as follows.

```
#ifndef GRAPH_H
#define GRAPH_H

#include <string>
#include <vector>
#include <array>

const int GRAPH_SIZE = 3e5;

// ingredient: GROWTH
const int ING_GROWTH = 0b01;

// ingredient: PREFERENTIAL ATTACHMENT
const int ING_PREF = 0b10;

class Graph {

public:
    // Graph constructor
    explicit Graph(int init_nv, int init_i, int init_vne);

    // called every time step
    void grow();

    // write graph data to file for plotting
    void output(std::string file_path) const;

    // show graph information
    void desc() const;

    // getter
    int get_num_vertices() const;

private:
```

```
    // add a new vertex with ‘num_edges‘ edges
    void grow_normal();

    // randomly select a vertex and connect it
    // with another one
    void grow_stationary();

    // select a vertex
    // either with higher-degree-preferential randomness
    // or uniformly distributed randomness
    // vetices in ‘exceptions‘ will not be considered
    int roulette_select(const std::vector<int> & exceptions) const;

    // update adjacent list
    void add_edge(int v1, int v2);

    // adjacent list
    std::array<std::vector<int>*, GRAPH_SIZE> adj_list;

    // quantity of vertices populated in the graph
    int num_vertices;

    // GROWTH, PREFERENTIAL ATTACHMENT or both
    int ingredients;

    // ‘vnew_num_edges‘ edges will be added
    // if a new vertex is added
    int vnew_num_edges;

};

#endif
```

Listing 1: Graph.h

4.1 Data Structures

About data structures, we choose a traditional one and a self-defined fancy one, namely Adjacency list and roulette, respectively.

4.1.1 Adjacency List

According to wikipedia, an adjacency list is a collection of unordered lists used to represent a finite graph and each list describes the set of neighbors of a vertex in the graph. We use STL array and vector to implement this. We make an array of vectors a private variable of class Graph. The index of a vector serves as an identifier of a certain vertex. Neighbors of that certain vertex are stored in the vector.

4.1.2 Roulette

We name this data structure in a fancy style. Basically, it is used to preferentially select a vertex from a group of vertex. We simply insert sticks in a number axis, and the distance between two neighboring sticks represent the probability of being the lucky dog. Then we use embedded pseudo-random number generator of C to get a picker, which falls between 0 and the largest stick. The details of this process is inside the function body of $roulette\_select()$ and roulette itself is a local variable of that function.

4.2 Class Interface and Simulation Steps

We have mainly four functions as the interface: the class initializer, $grow()$, $output()$ and $desc()$. First we are ought to initialize it with initial parameters, and let it grow for thousands of time steps, finally we output the result to a csv file for plotting. Function $desc()$ is for outputting the graph information to screen.

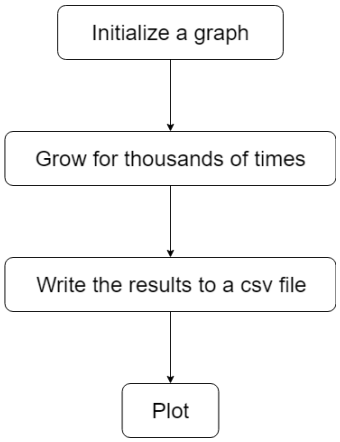

Figure 1: Simulation steps

5 Results and Analysis

Three test cases are provided here. Specifically, we choose either or both of the two ingredients discussed in section 2.2 and section 2.3.

5.1 Growth and Preferential Attachment

We get five initial vertices with no edge. At every time step (i.e. each time we call function *grow*()), we add a new vertex with 5 edges that link the new vertex to 5 different vertices already present in the network. We collect the power-law connectivity distribution at $t = 100,000$, $t = 150,000$ and $t = 200,000$.

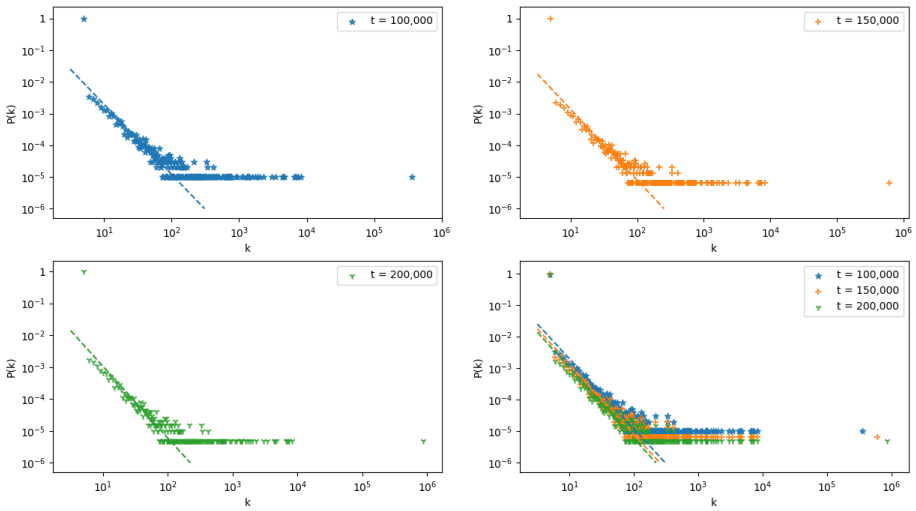

Figure 2: Model results with both ingredients

This test case is ought to generate a scale-free network as we discussed in section 2.1. This kind of network should present a power-law degree distribution as equation 1. For convenience of mathematical analysis, we restate it here.

$$P(k) \sim k^{-\gamma} \tag{2}$$

We simply apply logarithmic function to both side of equation 2. Thus we get:

$$ln\ P(k) \sim -\gamma ln\ k \tag{3}$$

Which implies a beautiful linear relationship between $ln\ P(k)$ and $ln\ k$. And figure 2 also presents linearity.

If we put the connectivity distribution at $t = 100,000$, $t = 150,000$ and $t = 200,000$ together like the lower-right subplot of figure 2, we can see that the three regression lines share a common slop and their intercepts do not vary too much. This feature is what we call scale-free or scale-invariance.

5.2 Growth Only

If we remove the preferential attachment ingredient and only keep the growth ingredient, things would be much more different. This model leads to another degree distribution as:

$$P(k) \sim e^{-\beta k} \tag{4}$$

We apply logarithmic function as we did in section 5.1.

$$ln\ P(k) \sim -\beta k \tag{5}$$

A different linearity is clear enough in equation 5. If we plot it, we will still get a compatible result.

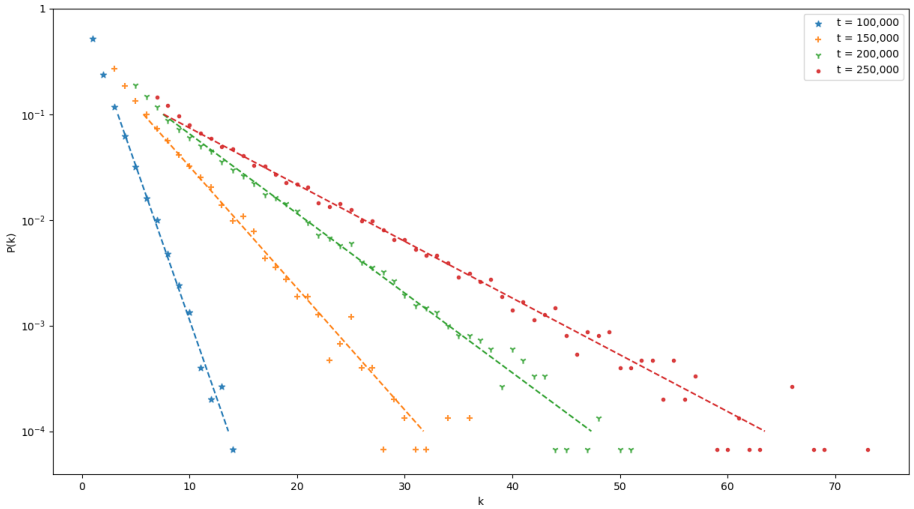

Figure 3: Model results with only the growth ingredient

5.3 Preferential Attachment only

We can not get detailed parameters from the referential paper. We are afraid this section must remain vacant due to our laziness : −).

6 Conclusion

In this practice, we learned the basic concepts of scale-free networks and implemented a model ourselves. Throughout the whole process, we gained the ability of reading English papers and reproducing algorithms accordingly. Cheers!

References

[1] A. L. Barabasi and R. Albert, "Emergence of scaling in random networks," 1999.

[2] wikipedia, "Scale-free network," 2019.

Appendices

```
#include "Graph.h"
#include <stdio.h>
#include <iostream>
using namespace std;

int main()
{

    // GROWTH and PREFERENTIAL ATTACHMENT
    Graph my_graph(5, ING_GROWTH | ING_PREF, 5);
    for (int i = 0; i < 2e5+1; ++i) {
        my_graph.grow();
        if (i % 1000 == 0)
            cout << "v" << i << " is added" << endl;
        if (i + 1 == 1e5 || i + 1 == 1.5e5 || i + 1 == 2e5) {
            char path[256];
            sprintf(path, "data/dc_t%dw.csv", i / 10000 + 1);
            my_graph.output(path);
        }
    }

    return 0;
}
```

Listing 2: Code of the test case with both ingredients

```
#include "Graph.h"
#include <stdio.h>
#include <iostream>
using namespace std;

int main()
{
   // GROWTH
   for (int m = 1; m < 9; m += 2) {
        Graph my_graph(5, ING_GROWTH | ING_PREF, 0);
        for (int i = 0; i < 2e5; ++i)
            my_graph.grow();
        char path[256];
```

```
        sprintf(path, "data/dc_growth_m%d.csv", m);
        my_graph.output(path);
    }

    return 0;
}
```

Listing 3: Code of the test case with only the growth ingredient

```
#include <cassert>
#include <cstdlib>
#include <algorithm>
#include <iostream>
#include <stdexcept>
#include <iomanip>
#include <math.h>
#include <set>
#include <fstream>
#include "Graph.h"
using namespace std;

Graph::Graph(int init_nv, int init_i, int init_vne)
        : num_vertices(init_nv),
          ingredients(init_i),
          vnew_num_edges(init_vne)
{
    assert(init_nv <= GRAPH_SIZE && init_vne <= init_nv);

    for (int i = 0; i < init_nv; ++i)
        this->adj_list[i] = new vector<int>;

    // ------ log begin ------

    cout << "Graph object initialized with "
        << init_nv << " vertices\n"
        << "every new vertex with " << init_vne
        << " edges\ningredients: ";
    if ((init_i & ING_GROWTH) == ING_GROWTH)
        cout << "GROWTH; ";
    if ((init_i & ING_PREF) == ING_PREF)
        cout << "PREFERENTIAL ATTACHMENT; ";
    cout << endl << endl;

    // ------ log end ------
```

```
    srand(23333);
}

void Graph::grow() {
    if ((this->ingredients & ING_GROWTH) == ING_GROWTH) {
        grow_normal();
    } else {
        grow_stationary();
    }
}

void Graph::output(string file_path) const {
    multiset<size_t, less<int>> degrees;
    for (int i = 0; i < this->num_vertices; ++i)
        degrees.insert(this->adj_list[i]->size());

    // distinc degrees
    set<size_t, less<int>> dd(
        degrees.cbegin(),
        degrees.cend()
    );

    ofstream writer(file_path, ios::out);
    if (!writer) {
        cerr << "[error] failed to open file" << endl;
        return;
    }

    writer << "degree,count\n";
    for (auto i = dd.cbegin(); i != dd.cend(); ++i)
        writer << *i << "," << degrees.count(*i) << "\n";
}

int Graph::get_num_vertices() const {
    return this->num_vertices;
}

void Graph::desc() const {
    cout << get_num_vertices() << " vertices here\n";

    int width = ceil(log10(this->num_vertices)) + 1;
    for (int i = 0; i < this->num_vertices; ++i) {
        cout << "v" << setw(width) << left << i;

        vector<int>* pneighbors = this->adj_list[i];
```

```
        if (pneighbors->size()) cout << "<- ";
        for (size_t i = 0; i < pneighbors->size(); ++i)
            cout << "v" << setw(width) << (*pneighbors)[i] << " ";

        cout << "\n";
    }

    cout << endl;
}

void Graph::grow_normal() {
    assert(this->num_vertices + 1 <= GRAPH_SIZE);

    int vnew = num_vertices;
    this->adj_list[vnew] = new vector<int>;

    // exclude vertices already connected with `vnew`
    vector<int> exceptions;
    for (int i = 0; i < this->vnew_num_edges; ++i) {
        int vtarget = roulette_select(exceptions);
        exceptions.push_back(vtarget);
        add_edge(vtarget, vnew);
    }

    num_vertices++;

    // ------ log begin ------

    desc();

    // ------ log end ------
}

void Graph::grow_stationary() {
    // randomly choose `vv` to add edge
    int vv = rand() % this->num_vertices;

    // exclude `vv` itself and its neighbors
    vector<int> exceptions{vv};
    copy(
        this->adj_list[vv]->cbegin(),
        this->adj_list[vv]->cend(),
        exceptions.begin() + 1
    );
    int vnew = roulette_select(exceptions);
```

```
    if (vnew != -1)
        add_edge(vv, vnew);

    // ------ log begin ------

    cout << "growing stationarily" << endl;
    desc();

    // ------ log end ------
}

int Graph::roulette_select(const vector<int> & exceptions) const {
    if ((int)exceptions.size() == this->num_vertices) {
        cerr << "[error] all vertices in exception list" << endl;
        return -1;
    }
    int with_pref = (this->ingredients & ING_PREF) == ING_PREF;

    vector<int> roulette(this->num_vertices, 0);
    roulette[0] = (with_pref ? adj_list[0]->size() : 0) + 1;
    for (int i = 1; i < this->num_vertices; ++i) {
        if (find(exceptions.begin(), exceptions.end(), i) ==
            exceptions.end())
            // if i not in `exceptions`
            roulette[i] = roulette[i - 1]
                + (with_pref ? adj_list[i]->size() : 0) + 1;
        else
            roulette[i] = roulette[i - 1];
    }

    int roulette_picker = rand() % roulette[this->num_vertices - 1];

    int lucky_dog = -1;

    // binary search
    int begin = 0;
    int end = this->num_vertices;
    int middle = (begin + end) / 2;

    while (begin < end) {
        if (roulette_picker < roulette[middle])
            end = middle;
        else
            begin = middle + 1;
        middle = (begin + end) / 2;
```

```
    }

    lucky_dog = begin;

    // ------ log begin ------

    cout << "roulette begins";
    if (exceptions.size()) cout << "\nexceptions: ";
    for (int e : exceptions)
        cout << e << " ";
    cout << "\n#\tdegree\tstick\n";

    for (int i = 0; i < this->num_vertices; ++i)
        cout << "v" << i << "\t"
            << this->adj_list[i]->size() << "\t"
            << roulette[i] << "\n";

    cout << "roulette picker falls on " << roulette_picker
        << "\nv" << lucky_dog << " is selected preferentially"
        << endl << endl;

    // ------ log end ------

    if (lucky_dog == -1)
        throw runtime_error("roulette failed");

    return lucky_dog;
}

void Graph::add_edge(int v_, int _v) {
    this->adj_list[v_]->push_back(_v);
    this->adj_list[_v]->push_back(v_);
}
```

Listing 4: Graph.cpp