Frontier Research Topics in Network Science Experimental Report

Error and Attack Tolerance of Complex Networks

Zhongyuan Chen

tolerance testing network implementation using C++

Hangzhou Dianzi University
School of Cybersecurity

2019-6-10

Contents

1 Overview

This report introduces two kinds of typical networks and compare the tolerance of them with code implementation.

1.1 Abstract

Many complex systems display a surprising degree of tolerance against errors. For example, if we randomly remove relatively small amount of some websites from the Internet, the whole Internet will still work. But if we deliberately remove some large websites, in other words, attack the Internet, will the Internet still be safe and sound?

This paper introduces the concept of tolerance and sees what if exponential networks and scale-free networks meet failures or be attacked.

1.2 Background

Students majored in information security must have some notion of Internet attacks. But what if we analyses Internet attacks from a mathematical perspective? Will this way offer a different view? Answers are in this paper.

Attacks and failures are actually different, malicious attackers may pretend that failures have happened, in order to cover the fact that they have done something bad. Knowing the difference between attacks and failures is also critical for us students.

2 Important Concepts

Several concepts appear frequently in this paper are introduced here.

An **exponential network** is a network with an exponential tail [1]. To produce an exponential network, we first define the N nodes, and then connect each pair of nodes with probability p. This kind of networks' connectivity follows a Poisson distribution.

A **scale-free network** is a network whose degree distribution follows a power law [2], at least asymptotically. 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 $2 < \gamma < 3$.

The **diameter** of networks is defined as the average length of the shortest paths between any two nodes in the network [3]. We denote it as d in this paper. d characterizes the ability of two nodes to communicate with each other. d also describes the interconnectedness of a network. The smaller d is, the shorter is the expected path between them.

3 Model Description

This model consists of two parts: the generation of two networks and the failure-attack simulation.

3.1 Generation of Networks

We first write codes to generate an exponential network and a scale-free network.

The generation of the exponential network is relatively simple. We just define an enough amount of nodes and connect every pair of them with a certain probability.

The scale-free model incorporates two ingredients common to real networks: growth and preferential attachment. Scale-free networks expand continuously by the addition of new vertices. And when a scale-free network grows, new vertices attach preferentially to sites that are already well connected. The model starts with m_0 nodes. At every time step t a new node is introduced, which is connected to m of the already-existing nodes. The probability Π_i that the new node is connected to node i depends on the connectivity k_i of node i such that $\Pi_i = k_i/\Sigma_j k_j$.

3.2 Failure-Attack Simulation

For failure simulation, we randomly remove some fraction of nodes and see how the diameter changes. For attack simulation, we choose the most well-connected nodes and remove them.

4 Code Implementation

We start with the C++ header file which will offer an overview of the whole implementation.

```
// tolerance attack testing graph module

#ifndef TGRAPH_H
#define TGRAPH_H

#include <array>
#include <vector>

class TGraph : public Graph {
public:
    // TGraph constructor
    explicit TGraph(int nodes_num);

    // exponential graph generator
    void exp_gen();

    // scale-free graph generator
    void sf_gen();

    // failure simulation
    void failure_sim();

    // attack simulation
    void attack_sim();

    // calculate graph diameter
    void diameter();

    // diameter getter
    double get_diameter() const;
private:
    int num_vertices;
    double diameter;
    std::array<std::vector<int>*, GRAPH_SIZE> adj_list;
}

#endif
```

Listing 1: TGraph.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 seven functions as the class interface: $TGraph()$ initializes the whole testing graph; exp_gen generates the exponential graph; $sf\_gen()$ generates the scale-free graph; $failure\_sim()$ is for failure simulation. $attack\_sim()$ is for attack simulation. $diameter()$ is to calculate the graph diameter; $get\_diameter()$ is to return the graph diameter.

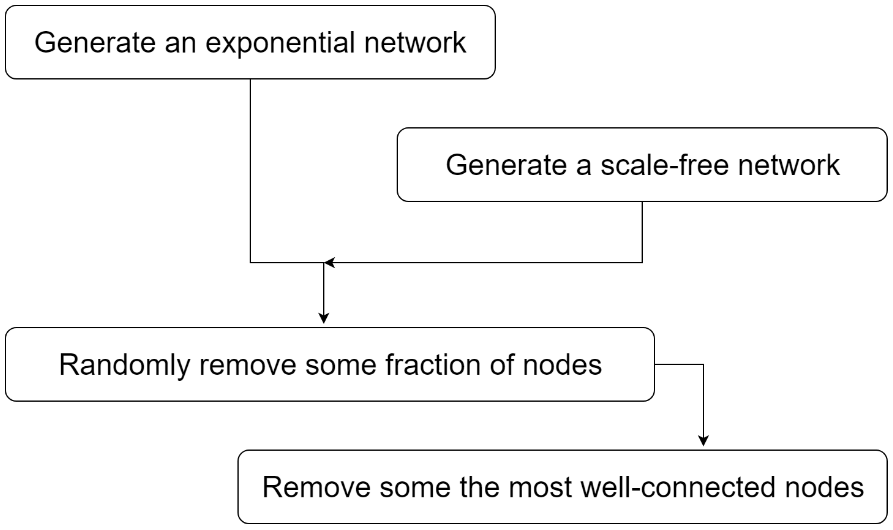
Generate an exponential network
Generate a scale-free network
Randomly remove some fraction of nodes
Remove some the most well-connected nodes

Figure 1: Simulation steps

5 Results and Analysis

We see changes in the diameter d of the network as a function of the fraction f of the removed nodes.

For the exponential network the diameter increases monotonically with f, it is increasingly difficult for the remaining nodes to communicate with each other. This behavior is rooted in the homogeneity of the network: since all nodes have approximately the same number of links, they all contribute equally to the network's diameter, thus the removal of each node causes the same amount of damage. In contrast, we observe a drastically different and surprising behavior for the scale-free the diameter remains unchanged under an increasing level of errors.

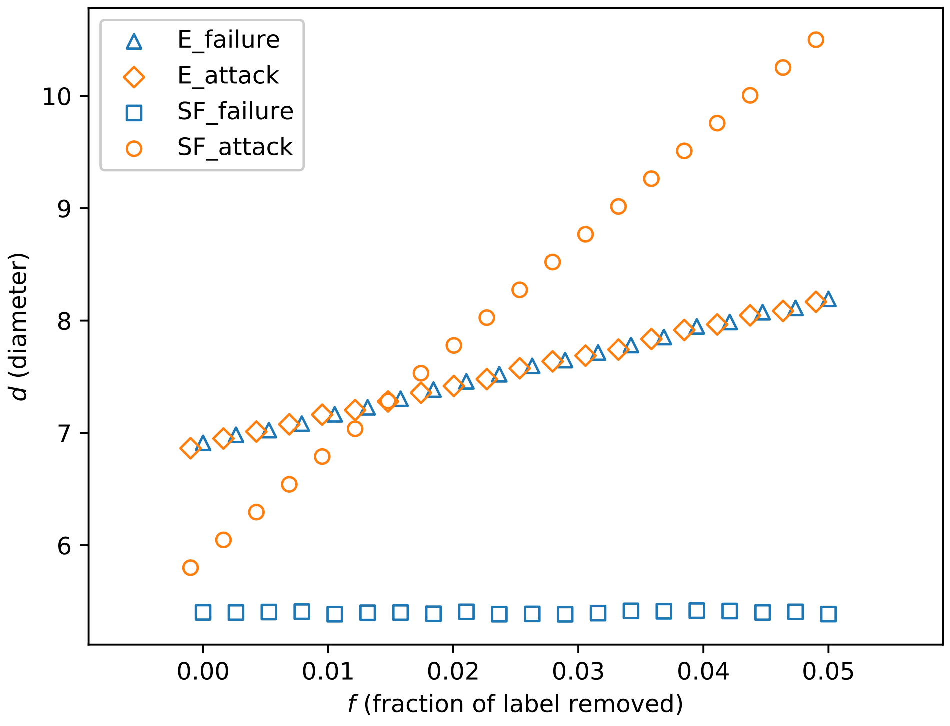

Figure 2: Diameter changes

6 Conclusion

In conclusion, we find that scale-free networks display a surprisingly high degree of tolerance against random failures, a property not shared by their exponential counterparts.

References

[1] P. ERDdS and A. R&wi, “On random graphs i,” *Publ. Math. Debrecen*, vol. 6, pp. 290–297, 1959.

[2] A. L. Barabasi and R. Albert, “Emergence of scaling in random networks,” 1999.

[3] R. Albert, H. Jeong, and A.-L. Barabási, “Error and attack tolerance of complex networks,” *nature*, vol. 406, no. 6794, p. 378, 2000.

Appendices

```
#include "TGraph.h"
using namespace std;

TGraph::TGraph(int nodes_num) {
    this->num_vertices = nodes_num;
}

void TGraph::exp_gen() {
    for (int i = 0; i < this->adj_list.size(); ++i) {
        for (int j = 0; j <= i; j++) {
            if (rand() % 1000 <= 1) {
                Graph::add_edge(i, j);
            }
        }
    }
}

void TGraph::sf_gen() {
    for (int i = 0; i <  2e5+1; ++i) {
        Graph::grow();
    }
}

void TGraph::diameter() {
    for (int i = 0; i < this->adj_list.size(); ++i){
        for (int j = 0; j < this->adj_list.size(); ++j){
            this->adj_list[i][j] = -1;
        }
        this->adj_list[i][i] = 0;
    }

    //edges count
    int m = 0;
    for (int i = 0; i < this->adj_list.size(); ++i) {
        m += this->adj_list[i].size();
    }

    while(m--){
        //nodes - let the indexation begin from 1
        int a, b;

        //edge weight
        int c;

        scanf("%d %d %d", &a, &b, &c);
        this->adj_list[a][b] = c;
    }

    //Floyd-Warshall
    for (int k = 1; k <= this->num_vertices; ++k){
        for (int i = 1; i <= this->num_vertices; ++i){
            if (this->adj_list[i][k] != -1){
                for (int j = 1; j <= this->num_vertices; ++j){
                    if (this->adj_list[k][j] != -1 &&
                        (this->adj_list[i][j] = -1 ||
                        this->adj_list[i][k]+this->adj_list[k][j] <
                        this->adj_list[i][j])){
                        this->adj_list[i][j] =
                            this->adj_list[i][k]+this->adj_list[k][j];
                    }
                }
            }
        }
    }

    this->diameter = -1;

    //look for the most distant pair
    for (int i = 1; i <= this->num_vertices; ++i){
        for (int j = 1; j <= this->num_vertices; ++j){
            if (this->diameter < this->adj_list[i][j]){
                this->diameter = this->adj_list[i][j];
                printf("%d %d\n", i, j);
            }
        }
    }

    return 0;
}

void TGraph::failure_sim() {
    for (int i = 0; i < 100; ++i) {
        int t1 = rand() % this->adj_list.size();
        remove_node(t1);
    }
}
```

```cpp
void TGraph::attack_sim() {
    for (int i = 0; i < 100; ++i) {
        int t1 = 0;
        for (int j = 0; j < this->adj_list.size(); j++) {
            if (this->adj_list[j].size() > this->adj_list[t1].size()) {
                t1 = j;
            }
        }
        remove_node(t1);
    }
}

double TGraph::get_diameter() const {
    return this->diameter;
}
```

Listing 2: TGraph.cpp

```cpp
#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 3: Graph.cpp