Frontier Research Topics in Network Science Experimental Report

Collective dynamics of ‘small-world’ networks

Zhongyuan Chen

‘small world’ network implementation using C++

Hangzhou Dianzi University
School of Cybersecurity

2019-4-6

Contents

1 Overview **1**

1.1 Abstract 1

1.2 Background 1

2 Important Concepts **2**

2.1 Characteristic Path Length 2

2.2 Clustering Coefficient 2

3 Model Description **2**

3.1 ‘Small-World’ Network Construction 3

3.2 Virus-spreading simulation 3

4 Code Implementation **3**

4.1 Data Structures 6

4.1.1 Vertex Struct 6

4.1.2 Adjacency List 6

4.1.3 Roulette 7

4.2 Class Interface and Simulation Steps 7

5 Results and Analysis **8**

5.1 The Practice of ‘Small-World’ Network Construction 8

5.2 The Practice of Virus-Spreading Simulation 10

6 Conclusion **10**

Appendices **11**

1 Overview

This report introduces the concept of 'small-world' networks and offers the corresponding code implementation.

1.1 Abstract

The concept of 'small-world' networks is introduced by Duncan J. Watts and Steven H. Strogatz [1]. In this report, we dive into Duncan J. Watts's theories and algorithms and further offer a simple implementation of such networks.

We quantify the structural properties of these graphs by their characteristic path length $L(p)$ and clustering coefficient $C(p)$. $L(p)$ and $C(p)$ play an important role in our model, both in describing the network structure and offering some inspiration for virus-spreading simulation.

We use C++ to generate several networks consisting of 1000 vertices. We see the plot of $L(p)$ and $C(p)$ waving elegantly, through which we can figure out our desired 'small-world' network. Further, we simulate virus-spreading processes in those networks and have a look at the relationship between $L(p)$ and the spreading speed.

1.2 Background

The well known theory, Six Degrees of Separation, suggests that any two people, at any distance, can be connected by six chained friends. In other words, one reaches his or her friend, then the friend reaches the 2-order friend... That process goes on and on and finally, the source-person reaches the destination-person through about a 6-order friendship. This phenomenon suggests that our friendship is roughly a 'small-world' network.

Basically, a 'small-world' network is just a middle ground between a regular network and a compeletely random network. A lot of real networks, like biological oscillators and genetic control networks, are better assumed to be 'small-world' networks during modelling.

2 Important Concepts

Several concepts appear frequently in this paper are introduced here.

2.1 Characteristic Path Length

Characteristic path length $L(p)$ is defined as the number of edges in the shortest path between two vertices, averaged over all pairs of vertices. $L(p)$ measures the typical separation between two vertices in the graph, aka the distance between clusters.

$$L(p) = \frac{\sum_i^n \sum_j^n distance(v_i, v_j)}{n(n-1)} \tag{1}$$

where n is the number of vertices.

2.2 Clustering Coefficient

The clustering coefficient $C(p)$ is defined as follows: Suppose that a vertex v has k_v neighbors; then at most $k_v(k_v-1)/2$ edges can exist between them (this occurs when every neighbor of v is connected to every other neighbor of v). Let C_v denote the fraction of these allowable edges that actually exist. Define C as the average of C_v over all v.

$$C_v = \frac{\sum_i^{k_v} \sum_j^{k_v} is\_linked(v_i, v_j)}{k_v(k_v-1)} \tag{2}$$

$$C(p) = \frac{\sum_v C_v}{n} \tag{3}$$

where n is the number of vertices.

3 Model Description

Our model consists of two parts. We first build candidates of 'small-world' networks. Then, we simulate virus-spreading processes within these candidates.

3.1 'Small-World' Network Construction

We start from a regular network with n vertices. These vertices sit as a ring. Each vertex is connected to its k nearest neighbors (k is an positive even integer).

For each edge, with probability p, we reconnect this edge to a vertex chosen uniformly at random over the entire ring, with duplicate edges forbidden; otherwise we leave the edge in place. We repeat this process by moving clockwise around the ring, considering each vertex in turn until one lap is completed. Next, we consider the edges that connect vertices to their second-nearest neighbors clockwise. As before, we randomly rewire each of these edges with probability p, and continue this process, circulating around the ring and proceeding outward to more distant neighbors after each lap, until each edge in the original lattice has been considered once. (As there are $nk/2$ edges in the entire graph, the rewiring process stops after $k/2$ laps.)

Actually, if $p = 0$, the original regular network remain unchanged. And if $p = 1$, we will obtain a completely random network. For candidates of 'small-world' networks, we select p from $0 < p < 1$.

3.2 Virus-spreading simulation

We use candidates from section 3.1 as the population structure. At time $t = 0$, a single infective individual is introduced into an otherwise healthy population. Infective individuals are removed permanently (by immunity or death) after a period of sickness that lasts one unit of dimensionless time. During this time, each infective individual can infect each of its healthy neighbors with probability r. On subsequent time steps, the disease spreads along the edges of the graph until it either infects the entire population, or it dies out, having infected some fraction of the population in the process.

4 Code Implementation

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

```
#ifndef WSGRAPH_H
#define WSGRAPH_H
```

```
#include <string>
#include <set>
#include <array>
#include <vector>

// start with a ring of ‘WSN‘ vertices
// each connected to its ‘WSK‘ nearest neighbours by undirected edges
const int ws_N = 1000;
const int ws_K = 10;
const int ws_R_SAMPLE_SIZE = 100;


struct Vertex{
    std::set<int>* pneighbors;

    // for virus infection
    // 1: normal; 0: infected; -1: dead or immunized
    int status;

    Vertex() :
        pneighbors(new std::set<int>),
        status(1) {}

};

struct GraphProp{
    double p;
    double Cp;
    double Lp;
    double Tp;
    double r_half;
};


class WsGraph {

public:
    // Graph constructor
    explicit WsGraph(double ws_p);

    // show graph information
    void desc() const;

    // get graph properties
    GraphProp* dump() const;
```

```
private:
    // initialize as a regular network
    void init_regular();

    // break the link between ‘va‘ and ‘vb‘
    // randomly choose ‘vc‘ and link it to ‘va‘
    void reconnect(int va, int vb);

    // calculate the structural properties
    // namely C(p) and L(p)
    void cal_props();

    // breadth first search
    // generates shortest path lengths
    void bfs_lp(int center,
        std::array<int, ws_N>& path_lengths);

    // breadth first search
    // for virus spreading
    void bfs_virus(int center, double infect_rate);

    // prepare for virus infection
    void init_virus();

    void spread();

    void is_infecting();

    // randomly select a vertex
    // vetices in ‘exceptions‘ will not be considered
    int roulette_select(std::set<int>* pexceptions) const;

    void add_edge(int v_, int _v);

    void remove_edge(int v_, int _v);

    // with probability ‘p‘
    // we reconnect this edge to another vertex
    double p;

    // clustering coefficient C(p)
    double Cp;

    // characteristic path length L(p)
    double Lp;

    // the time required for global infection T(p) with r = 1
    double Tp;

    // probability of infection
    std::array<double, ws_R_SAMPLE_SIZE> r;

    // critical infection rate
    double r_half;

    // maximumly infected corresponding to ‘r‘
    std::array<int, ws_R_SAMPLE_SIZE> max_infect;

    // adjacency list
    std::array<Vertex*, ws_N> adj;

};


#endif
```

Listing 1: WsGraph.h

4.1 Data Structures

We are not using any fancy data structures.

4.1.1 Vertex Struct

As defined in the front of WsGraph.h, a vertex struct simply consists of its infection status and neighbors. The infection status indicates whether the person it represents is healthy, infected or dead (immunized). The neighbor list is represented as a STL set, which guarantees that the elements inside are always ordered. Thus we can search through the neighbor list in $\Omega(logN)$ time complexity.

4.1.2 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 implement this. We make an array of vertex structs (see section 4.1.1) a private variable of class WsGraph. The index of a vertex struct serves as an identifier of a certain vertex.

4.1.3 Roulette

We name this data structure in a fancy style. Basically, it is used to randomly select a vertex from a group of vertex with exceptions. 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*(), *dump*() and *desc*(). First we are ought to initialize it with initial parameters, and it will perform simulation automatically, then we output the result from *dump*() to a csv file for plotting. Function *desc*() is for outputting the graph information to screen.

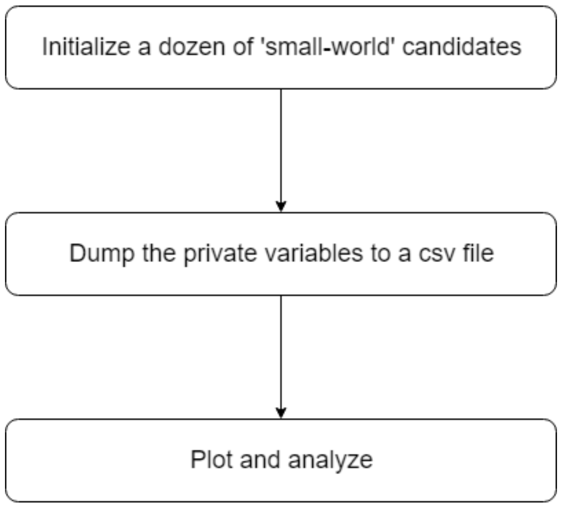
Initialize a dozen of 'small-world' candidates
Dump the private variables to a csv file
Plot and analyze

Figure 1: Simulation steps

5 Results and Analysis

The results of ‘small-world’ network construction and virus-spreading simulation are provided here.

5.1 The Practice of ‘Small-World’ Network Construction

Characteristic path length $L(p)$ and clustering coefficient $C(p)$, as discussed in section 2, are indices of choosing ‘small-world’ networks. Our desired networks should have a relatively small $L(p)$ and a relatively high $C(p)$. For convenience of comparing the two indices, we divide them by their maximum values for normalization. Now we plot the two functions as below:

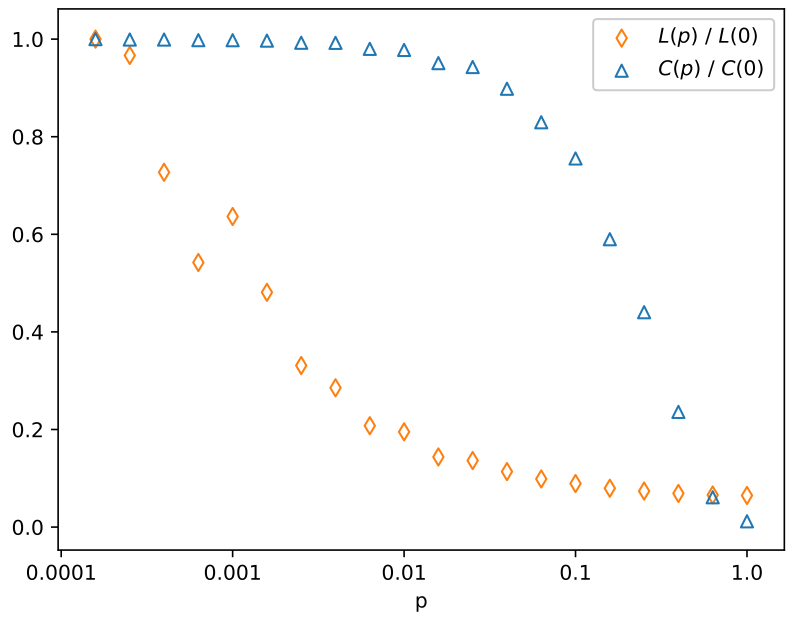

Figure 2: $L(p)$ and $C(p)$

From figure 2, we can see that in the vicinity of $p = 0.01$, $L(p)$ has decayed greatly while $C(p)$ still remains large.

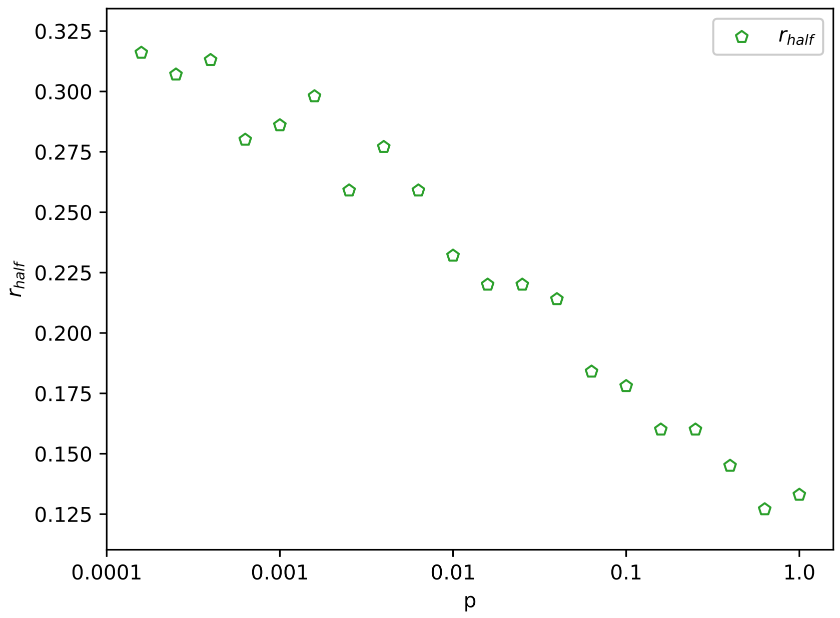

Figure 3: The relationship between r_{half} and p

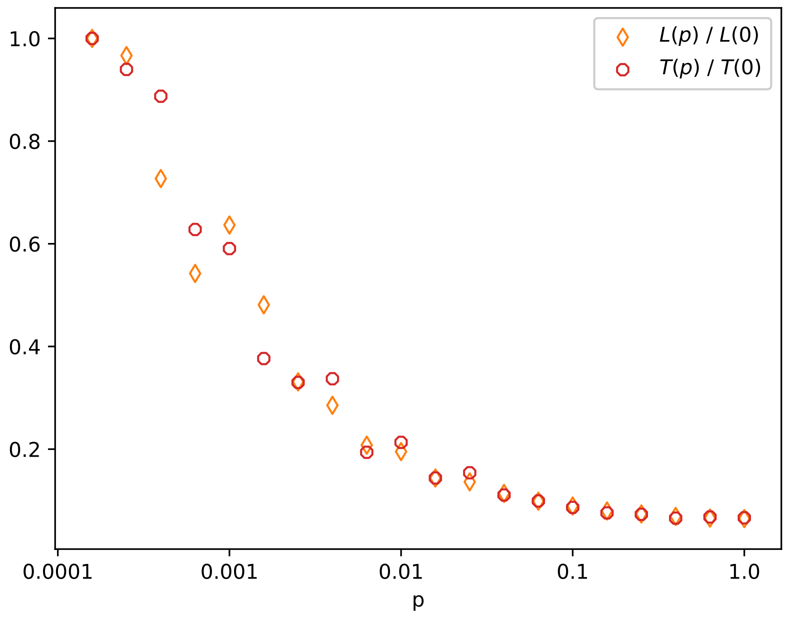

Figure 4: $L(p)$ and $T(p)$

5.2 The Practice of Virus-Spreading Simulation

Two interesting results emerge here.

We define the critical infectiousness r_{half} as the infection rate at which the disease infects half the population. The relationship between r_{half} and p is plotted in figure 3. We can see that r_{half} decreases rapidly for small p.

We define $T(p)$ as the time required for global infection. We can see in figure 4, $T(p)$ resembles $L(p)$. In fact, if we iterate through all the vertices and choose them as the original infected person in turn, then we plot the average $T(p)$, we will have a $T(p)$ curve exactly the same as $L(p)$.

6 Conclusion

'Small-world' networks are nothing but regular networks randomly reconnect part of their edges. However, those simple but attractive networks do play an significant role in figuring out the properties of real networks. For example, the virus-spreading simulation performed in section 5.2 shows that infectious diseases are predicted to spread much more easily and quickly in a small world; the alarming and less obvious point is how few short cuts are needed to make the world small.

References

[1] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks.," *Nature*, 1998.

Appendices

```cpp
#include <iostream>
#include <iomanip>
#include <stdlib.h>
#include <algorithm>
#include <set>
#include <cmath>
#include <numeric>
#include <cassert>
#include <queue>
#include <fstream>
#include "WsGraph.h"
using namespace std;

WsGraph::WsGraph(double ws_p)
    : p(ws_p), Cp(0), Lp(0), Tp(0)
{
    init_regular();
    for (int i = 0; i < ws_N * ws_K / 2; ++i) {
        reconnect(i % ws_N, (i + i / ws_N + 1) % ws_N);
    }
    cal_props();
    init_virus();
    spread();
}

void WsGraph::desc() const {
    int width = ceil(log10(ws_N)) + 1;
    for (int i = 0; i < ws_N; ++i) {
        Vertex* pv = this->adj[i];
        cout << "v" << setw(width) << left << i;
        cout << "| " << setw(2) << left << pv->status;

        if (pv->pneighbors->size()) cout << " <- ";
        for (size_t j = 0; j < pv->pneighbors->size(); ++j)
            cout << "v" << setw(width) << *next(pv->pneighbors->begin(),
                j) << " ";

        cout << "\n";
    }

    cout << endl;
}
```

```
void WsGraph::init_regular() {
    for (int v = 0; v < ws_N; v++)
        this->adj[v] = new Vertex;
    for (int v = 0; v < ws_N; v++)
        for (int jump = 1; jump < ws_K / 2 + 1; jump++)
            add_edge(v, (v + jump + ws_N) % ws_N);
}

void WsGraph::add_edge(int v_, int _v) {
    if (!this->adj[v_]->pneighbors->insert(_v).second)
        cerr << "[warning] " << _v
            << " already in adjacency of " << v_ << '\n';
    if (!this->adj[_v]->pneighbors->insert(v_).second)
        cerr << "[warning] " << v_
            << " already in adjacency of " << _v << '\n';
}

void WsGraph::remove_edge(int v_, int _v) {
    if (!this->adj[v_]->pneighbors->erase(_v))
        cerr << "[warning] " << _v
            << " not in adjacency of " << v_ << '\n';
    if (!this->adj[_v]->pneighbors->erase(v_))
        cerr << "[warning] " << v_
            << " not in adjacency of " << _v << '\n';
}

void WsGraph::reconnect(int va, int vb) {
    if (1.0 * rand() / RAND_MAX > this->p)
        return;
    remove_edge(va, vb);

    // exceptions
    set<int>* pe = new set<int>(
        this->adj[va]->pneighbors->begin(),
        this->adj[va]->pneighbors->end()
    );
    pe->insert(va);

    int vc = roulette_select(pe);
    add_edge(va, vc);
}

int WsGraph::roulette_select(set<int>* pexceptions) const {
    set<int>* pe = pexceptions;
    if (pe != nullptr && pe->size() == (unsigned)ws_N) {
        cerr << "[error] all vertices in exception list" << endl;
        return -1;
    }

    vector<int> roulette(ws_N, 0);
    roulette[0] = pe->find(0) == pe->end() ? 1 : 0;
    for (int i = 1; i < ws_N; ++i) {
        if (pe == nullptr || pe->find(i) == pe->end())
            roulette[i] = roulette[i - 1] + 1;
        else
            roulette[i] = roulette[i - 1];
    }

    int roulette_picker = rand() % roulette[ws_N - 1];

    int lucky_dog = -1;

    // binary search
    int begin = 0;
    int end = ws_N;
    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;

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

    return lucky_dog;
}

void WsGraph::cal_props() {
    /* --- characteristic path length L(p) --- */

    // shortest path length
    array<int, ws_N> spl;
    for (int v = 0; v < ws_N; ++v) {
        for (int i = 0; i < ws_N; ++i)
            spl[i] = -1;
        spl[v] = 0;

        bfs_lp(v, spl);

        if (spl.end() != find(
                spl.begin(),
                spl.end(),
                -1
            )) {
            throw runtime_error("[error] bfs_lp failed");
        }

        double avg = accumulate(
            spl.begin(),
            spl.end(),
            0.0
        ) / (ws_N - 1); // exclude 0

        this->Lp = (v * this->Lp + avg) / (v + 1);
        if (v == 0) this->Tp = avg;
    }

    /* --- clustering coefficient C(p) --- */
    for (int v = 0; v < ws_N; ++v) {
        int edge_num_neighbors = 0;
        set<int>* pn = this->adj[v]->pneighbors;

        for (auto it1 = pn->cbegin(); it1 != pn->cend(); ++it1) {
            for (auto it2 = next(it1, 1); it2 != pn->cend(); ++it2) {

                set<int>* pn1 = this->adj[*it1]->pneighbors;
                if (pn1->find(*it2) != pn1->end())
                    ++edge_num_neighbors;

            }
        }

        int edge_num_max = pn->size() * (pn->size() - 1) / 2;
        double fraction = 1.0 * edge_num_neighbors / edge_num_max;
        this->Cp = (v * this->Cp + fraction) / (v + 1);
    }
}
```

```
void WsGraph::bfs_lp(int center,
        array<int, ws_N>& path_lengths) {
    array<int, ws_N>& spl = path_lengths;
    queue<int> queue_vertices;
    do {
        for (
            auto it = this->adj[center]->pneighbors->cbegin();
            it != this->adj[center]->pneighbors->cend();
            ++it
        ) {
            if (spl[*it] == -1) {
                spl[*it] = spl[center] + 1;
                queue_vertices.push(*it);
            }
        }
        center = queue_vertices.front();
        queue_vertices.pop();
    } while (!queue_vertices.empty());
}

void WsGraph::bfs_virus(int center, double infect_rate) {
    queue<int> queue_vertices;
    while (true) {
        for (
            auto it = this->adj[center]->pneighbors->cbegin();
            it != this->adj[center]->pneighbors->cend();
            ++it
        ) {
            if (this->adj[*it]->status == 1
                    && 1.0 * rand() / RAND_MAX < infect_rate) {
                this->adj[*it]->status = 0;
                queue_vertices.push(*it);
            }
        }
        this->adj[center]->status = -1;
        if (queue_vertices.empty()) break;
        center = queue_vertices.front();
        queue_vertices.pop();
    }
}

GraphProp* WsGraph::dump() const {
    GraphProp* pgp = new GraphProp;
    pgp->p = this->p;
    pgp->Cp = this->Cp;
    pgp->Lp = this->Lp;
    pgp->Tp = this->Tp;
    pgp->r_half = this->r_half;
    return pgp;
}

void WsGraph::init_virus() {
    double r_begin = 0.1;
    double r_end = 0.4;
    double r_step = (r_end - r_begin) / ws_R_SAMPLE_SIZE;
    for (int i = 0; i < ws_R_SAMPLE_SIZE; ++i) {
        this->r[i] = r_begin + i * r_step;
        this->max_infect[i] = 0;
    }
}

void WsGraph::spread() {
    for (int i = 0; i < ws_R_SAMPLE_SIZE; ++i) {
        this->adj[0]->status = 0;
        bfs_virus(0, this->r[i]);
        for (int v = 0; v < ws_N; ++v) {
            if (this->adj[v]->status == -1)
                this->max_infect[i] += 1;
            this->adj[v]->status = 1;
        }
    }
    for (int i = 0; i < ws_R_SAMPLE_SIZE; ++i) {
        if (this->max_infect[i] > ws_N / 2) {
            this->r_half = this->r[i];
            break;
        }
    }
}
```

Listing 2: WsGraph.cpp

```
#include "WsGraph.h"
#include <cmath>
#include <fstream>
#include <iostream>
using namespace std;

void model();

int main() {
    model();
    return 0;
}

void model() {
    srand(233);
    double p_begin = 0;
    double p_distance = -4;
    double p_sticks = 20;
    double p_step = p_distance / p_sticks;
    ofstream writer("data/graph-properties.csv", ios::out);
    writer << "p,Cp,Lp,Tp,rhalf\n";
    for (int i = 0; i < p_sticks; ++i) {
        WsGraph my_graph(pow(10, (p_begin + p_step) * i));
        GraphProp* pgp = my_graph.dump();
        cout << i << " / " << p_sticks << endl;
        writer << pgp->p << ',' << pgp->Cp << ','
            << pgp->Lp << ',' << pgp->Tp << ','
            << pgp->r_half << '\n';
    }
    WsGraph my_graph(0);
    GraphProp* pgp = my_graph.dump();
    writer << pgp->p << ',' << pgp->Cp << ','
        << pgp->Lp << ',' << pgp->Tp << ','
        << pgp->r_half << '\n';
}
```

Listing 3: Network constructiona and virus-spreading simulation