IMPROVING GEMFSIM: A STOCHASTIC SIMULATOR FOR THE GENERALIZED EPIDEMIC MODELING FRAMEWORK

by

FUTING FAN

B.S., Beijing University of Aeronautics and Astronautics, China, 2011

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2016

Approved by:

Major Professor
Caterina Scoglio

Abstract

The generalized epidemic modeling framework simulator (GEMFsim) is a tool designed by Dr. Faryad Sahneh, former PhD student in the NetSE group. GEMFsim simulates stochastic spreading process over complex networks. It was first introduced in Dr. Sahneh's doctoral dissertation *"Spreading processes over multilayer and interconnected networks"*[1] and implemented in Matlab. As limited by Matlab language, this implementation typically solves only small networks; the slow simulation speed is unable to generate enough results in reasonable time for large networks. As a generalized tool, this framework must be equipped to handle large networks and contain sufficient support to provide adequate performance.

The C language, a low-level language that effectively maps a program to machine instructions with efficient execution, was selected for this study. Following implementation of GEMFsim in C, I packed it into Python and R libraries, allowing users to enjoy the flexibility of these interpreted languages without sacrificing performance.

GEMFsim limitations are not limited to language, however. In the original algorithm (Gillespie's Direct Method[2,3]), the performance (simulation speed) is inversely proportional to network size, resulting in unacceptable speed for very large networks. Therefore, this study applied the Next Reaction Method[4], making the performance irrelevant of network size. As long as the network fits into memory, the speed is proportional to the average node degree of the network, which is not very large for most real-world networks.

This study also applied parallel computing in order to advantageously utilize multiple cores for repeated simulations. Although single simulation can not be paralleled as a Markov process, multiple simulations with identical network structures were run simultaneously, sharing one network description in memory.

Table of Contents

List of Figures

List of Tables

Acknowledgments

My deepest gratitude goes to Dr. Caterina Scoglio for her ample assistance throughout my educational journey. Without her constant support and inspiration, I would not have been able to complete this academic endeavor.

I would also like to express my sincere appreciation of Dr. Faryad Sahneh for his groundwork in this area and continuous help in my work.

I wish to express my gratitude to Dr. William Hsu for his time and effort reviewing my report manuscript and being in the committee.

I would like to thank Aram Vajdi and Heman Shakeri, who gave a lot of useful suggestions and helped test the code.

Last but not least, I want to take this opportunity and express my love to my girlfriend, Lu Jiang. I would not be here without your support. Thank you for always believing in me.

Dedication

This is dedicated to my parents who love me, believe in me, inspire me and let go of me even when they don't agree with me.

Preface

This report is submitted for partial fulfillment of the requirements for the degree Master of Science at Kansas State University. The report work was conducted from May 2015 to Oct 2016 under the supervision of Professor Caterina Scoglio in the Department of Electrical and Computer Engineering, Kansas State University.

To the best of my knowledge this work is original except where acknowledgments and references are made to previous work.

This dissertation should be of interest to broad communities of engineers, chemists, and physicists with interest in network science and exact stochastic simulation over complex networks.

Finally, this report is based upon the work I did while I was supported as a research assistant by funds from the National Science Foundation under Award CIF-1423411.

Chapter 1

Introduction

In this section I give a brief introduction to the background of my work as well as some essential concepts in epidemic modeling.

1.1 Background

Epidemic spreading occurs in both natural and technological contexts, such as pathogen spreading among human or animal groups, computer virus propagation over the Internet, or viral rumours in SNS. Various models have been developed to predict the movements of spreading processes. Traditional models define individuals' compartments including immune, susceptible, exposed, infectious, symptomatic, recovered, dead, vaccinated, and then the models define rules for compartment transformation, assuming that the entire network is fully connected.

Scientists have studied this area for decades and proven that interactions among populations significantly impact spreading dynamics. Classic random network models assume only a particular network structure, however, so a generic model was recently developed, in which nodes represent individuals, links denote interactions between a pair of individuals, and a node's current state (compartment) and states of its neighbors determine the system transition.

Due to the infinite possibilities of node state definitions and transition rules, the number of potential models is unmeasurable. However, one widely applied fundamental assumption is that individuals influence each other by statistically independent pairwise interactions. Independent means that interaction between nodes A and B is statistically independent of interactions between nodes A and C or nodes C and D. Pairwise indicates no higher order interaction such as $A - B - C$, which is equivalent to $A - B$, $B - C$, and $A - C$ combined.

1.2 Generalized Epidemic Modeling Framework

GEMFsim is a stochastic simulator for the Generalized Epidemic Modeling Framework (GEMF)[5] that was designed by Dr. Faryad Sahneh. GEMFsim numerically simulates epidemic spreading processes over complex networks, including the expansion of viruses and information over the Internet and the propagation of multiple pathogens within a group of hosts.

1.2.1 Motivating example: SIS model on a graph

Although many models with various complexities are available for studying epidemic spreading driven by interactions of individuals in a network, I chose the susceptible-infected-susceptible (SIS) model because of its simplicity. In the SIS model, nodes of the graph represent individuals and edges of the graph stand for possible interactions. A link between node m and node n is evident if one of the nodes can potentially infect the other node directly. Moreover, $x_n(t) \in \{1, 2\}$ denotes the state of node n at time t, where $x_n(t) = 1$ means node n is susceptible and $x_n(t) = 2$ means node n is infected. The SIS model contains two types of transition: *susceptible* to *infected* $(1 \rightarrow 2)$ and *infected* to *susceptible* $(2 \rightarrow 1)$. The first transition occurs if a susceptible node has infected neighbors. When a node n is in the susceptible state, the probability of this transition depends on a parameter β determined by the epidemic model and the number of infected neighbors. The second transition, or recovery process, occurs spontaneously with a rate δ, independent from the state of neighbors.

Fig. 1.1 shows node-level transitions for the SIS model.

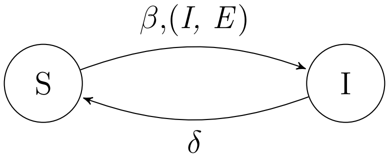


Figure 1.1: Transition diagram for SIS epidemic model: δ is the recovery rate of infected nodes, and β is the rate for infecting a susceptible node by an infected neighbor in the network denoted by the edge set E.

1.2.2 GEMF description

Although spreading models have unique specific assumptions, most models share the characteristic of independent pairwise interaction. Therefore, Dr. Sahneh developed the generalized epidemic modeling framework (GEMF), which facilitates systematic development of a broad spectrum of stochastic spreading processes over complex networks. As a simple epidemic model, SIS can be formulated within GEMF, and similar to the SIS model, GEMF uses nodes and edges of networks to represent individuals and interactions, respectively. However, because contact network can contain several layers, I represented the network by $G(V, E_1, \ldots, E_L)$, where L is the number of contact layers, V is a set of N nodes, and E_l is a set of links between the nodes in layer l. As with the SIS model, the state of node n at time t is a random variable denoted by $x_n(t)$. However, each node can assume a state among M possible states, which are labeled with an integer from 1 to M (i.e., $x_n(t) \in \{1, \ldots, M\}$). In GEMF, transitions of a node n over all possible states 1 to M are divided into two categories.

The multilayer network topology makes GEMF a novel framework for stochastic simulation.

1.2.3 Nodal transition

Nodal transition occurs independently from the states of neighbors in the network (i.e., recovery process in the SIS model). Since any state i can potentially transition to another state j, I define a $M \times M$ nodal transition matrix, A_δ, where element $A_\delta(i, j)$ is the transition rate of a node n from state i to state j. In other words, nodal transition $i \rightarrow j$ can be considered a jump of node n from state i to j with a jump time exponentially distributed with rate $A_\delta(i, j)$. Actually, I used this data structure to store and access such parameters in the program.

1.2.4 Edge-based transitions

Edge-based transitions are caused by interactions with neighbors in the network depending on states of the neighbors (i.e., infecting process in the SIS model). For a single contact layer, the edge-based transition matrix is similar to that of a nodal transition. Since GEMF allows each layer to have their own transition rates, I defined a transition rate array A_β to represent all edge-based transition rates. Element $A_\beta\ (i, j; l)$ was the rate of transition of a node n from state i to j as a result of interaction with neighbors in state $q(l)$ for layer l. State $q(l)$ was called the influencer state for layer l; only one influencer state was identified in each layer. If k $(k > 1)$ influencers are present in one layer for some epidemic models, GEMF must consider k network layers with identical network structure and one influencer state each layer. The layer provides contacts for a node in the influencer state to spread over neighboring nodes. In the SIS model, the influcencer of the only layer is the infected state represented by integer 2. Similar to nodal transitions, I considered the edge-based transition $i \rightarrow j$ of layer l as a jump of node n from state i to j with a jump time that was exponentially distributed with a rate related to $A_\beta\ (i, j; l)$ and the number of neighbor nodes in state $q(l)$. If no neighbor of node n was in state $q(l)$, then the edge-based transition of node n for layer l did not exist.

These two types of transitions, nodal transition and edge-based transition, are not exclusive, however. Depending on the epidemic model, the jumping of node n from state $i \rightarrow j$ could be a result of neighbor interaction or nodal transition. In this case, the processes are assumed to be mutually independent and the transition rate is the sum of all rates for possible processes, that

$$Pr(x_n(t+\Delta t)=j|x_n(t)=i)=\lambda_n(i\to j)\Delta t,$$

where $\lambda_n(i\to j)=r_1+\cdots+r_k$ and $r_1,\ldots,r_k$ are rates for the possible process.

1.2.5 Algorithm

For a system of stochastically interactional components, Gillespie[3] proposed two exact stochastic simulation algorithms. I used the Direct Method, which primarily focuses on determining the sequence and timing of each reaction by specifying the probability density $P(n,t)$ that the next reaction is n and occurs at time t. Therefore,

$$P(n,t)dt=r_n exp(-t\sum_j r_j)dt.$$

Integration of $P(n,t)$ over all t from 0 to ∞ results in

$$Pr(Reaction=n)=r_n/\sum_j r_j,$$

which is the probability distribution for reactions. Sum $P(n,t)$ over all n results in

$$P(t)dt=(\sum_j r_j)exp(-t\sum_j r_j)dt,$$

which is the probability distribution for time.

The Direct Method presented above is similar to the algorithm of GEMF, except that the transition rate in GEMF is more complicated due to nodal transition and multilayer edge-based transition. GEMF simulation is a Markov process with dynamics that arise from node-level transitions. Using contact network $G(V,E_1,\ldots,E_L)$, nodal transition matrix A_δ, and edge-based transition array A_β, all node-level transition rates set S can be calculated. I used $\lambda_n(x_n\to j)$ to represent the transition rate of node n from its current state x_n to state

j.

The probability of $T_n(x_n \to j)$ being the minimum element in S is

$$Pr(T_n(x_n \to j) = min(S)) = \frac{\lambda_n(x_n \to j)}{\lambda_{tot}},$$

where $\lambda_{tot} \triangleq \sum_{n=1}^{N} \sum_{j=1}^{M} (x_n \to j)$ is the sum of all transition rates corresponding to elements of S.

Although probability distribution allows sampling of a node-level transition that is a transition of the network state, the occurring time of the transition also must be sampled. Because elements of S have exponential distributions and are independent, $T = min(S)$ can easily be proven to be exponentially distributed with a rate of λ_{tot}. Therefore, I sampled a time δ_t for the network state transition based on the distribution of T, and then I selected a node n that achieved the next transition according to probability distribution $Pr(n) = \lambda_n/\lambda_{tot}$.

After selecting the node , I chose a new state j according to probability distribution $Pr(j|n) = \lambda_n(x_n \to j)/\lambda_n$, resulting in the acquisition of complete information of current transition, including node n, state jump from i to j, and time cost δ_t. As a Markov process, this procedure can be repeated after updating the state change followed by the current transition. Modification includes updating the transition rates of node n and neighbors that are affected by node n in any contact layer. The other rates remain unchanged.

Utilization of GEMF also allowed the assignation of weight to each link in order to quantify the effect of neighbors on edge-based transitions. Compared to the unweighted model, all parameters, including $q(l)$, A_δ, and A_β, remained the same. When calculating the rates of edge-based transitions, the weight of the link must be multiplied to each edge. The revised transition rate formula of node n jumping from current state x_n to state j is

$$\lambda_n(x_n \to j) = A_\delta(x_n, j) + \sum_{l=1}^{L} A_\beta(x_n, j; l) \sum_{m=1}^{N} W(m, n; l)\delta_{x_m, q(l)},$$

where $\delta_{s,t}$ is Kronecker delta and $\lambda_{tot} \triangleq \sum_{n=1}^{N} \sum_{j=1}^{M} (x_n \to j)$.

1.3 Contributions

This reported work is primarily theoretical. Although Dr. Faryad Sahneh implemented GEMFsim in Matlab language, this implementation was incapable of handling large networks due to inefficient memory management of the Matlab language.

I first implemented GEMFsim in C language, thereby increasing simulation speed approximately 10^3 times depending on network size and the epidemic model. Overall, the speed ratio of the C version compared to the Matlab version was bigger for larger networks. The C implementation had $O(n)$ time complexity, where n is the number of nodes in the contact network, a barrier inherent in the algorithm and unbreakable by any optimization. Therefore, I bypassed the barrier via the Next Reaction Method developed by Gibson[4]. With proper data structure adopted, the revised program has $O(m)$ time complexity, where m is the sum of average node degrees over all layers of the network. This value does not grow with network size for a fixed network model and is small for most situations as common sense.

As a low-level language, C is, unfortunately, not at all user friendly. Therefore, I packed the simulation function into Python and R libraries so that users can generate or import contact networks and control parameters in these script languages and run simulations with the C function, consequently preserving convenience and efficiency. Moreover, I applied parallel computation techniques in order to advantageously utilize multi-core CPUs, thereby saving significant simulation time without additional memory requirements since all threads share one copy of the contact network.

Chapter 2

Implementation of the Direct Method

In this chapter I explain basic implementation of the GEMFsim tool in C language.

2.1 Program structure

The simple program structure as shown in Figure 2.1 was comprised of read, run, and save, and the program consisted of four files as shown in Figure 2.2. The running process was controlled by a main function in gemf.c, and functions used to check and read the config file were stored in para.c. Data structure and common functions such as time and mathematical calculations were contained in common.c. The simulation function was in sim.c.

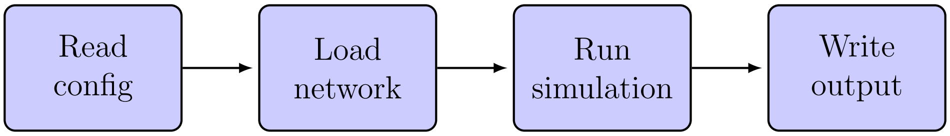


Figure 2.1: Program flowchart

In order to support the generalized model and multilayer simulation, however, the usage is more complicated than the structure: users must specify a network file in adjacency list format, nodal transition matrix, and edge-based transition matrices, as well as basic simulation control parameters. A sample configuration file is included in Appendix A. Additional usage details can be acquired from program manual[6].

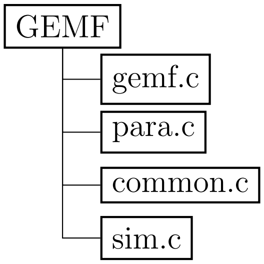


Figure 2.2: Program file list

2.2 Data structure and interface

The simulation function is defined below.

```
int sim(Graph* graph, Transition* tran, Status* sts, Run* run);
```

All four arguments are C structs: the first three are inputs, and the last one is input/output dual-direction. All input values are subject to change except for transition matrices.

Details of these self-defined structures are presented in Appendix B. This independent simulation function is the foundation for the next steps, including packing into Python and R library and applying new algorithms, which I will talk about later.

2.3 Performance-improving methods

In order to save simulation time for each step, I optimized the program from two main aspects: saving memory space and reducing computation. I used an adjacency list to store contact networks, more efficiently saving memory space compared to a 2-D adjacency matrix when the matrix is sparse. I also defined different structs for weighted and unweighted models to avoid memory waste caused by dealing with an unweighted network model using the weighted link struct.

In order to reduce computation, I generated an index array to track the start and end position of links oriented from every node, thereby requiring the network to be sorted and directed. If the network is undirected, I transformed it into directed and then sort the network with qsort. Because the search space was monotone nondecreasing, a search was performed among all nodes to find the expected node n according to λ_n distribution. Then $\sum_i^M \lambda(i \rightarrow j)$ was stored for all j since it is frequently used. For λ_{tot}, an increment according to the rate change of every affected node in current event was used to update instead of full re-summation.

2.4 Simulation and performance

Experimental simulations demonstrated that C significantly outperformed Matlab due to direct memory control of C and adopted optimization tricks.

net	nodes	edges	event number	simulation time	sec/event
G_1	10^3	9589	$2 * 10^4$	0.10s	5.0*10^{-6}
G_2	10^4	102230	$2 * 10^5$	3.97s	1.98*10^{-5}
G_3	10^5	1029821	$2 * 10^6$	347.61s	1.73*10^{-4}
G_4	10^6	10350638	$2 * 10^6$	13179.84s	6.58*10^{-3}

Table 2.1: Update time

The simulation shown in Table 2.1 was based on $\rho(G1) = 28$, $\delta = 1$, $\beta = \frac{3\delta}{\rho(G_1)} = 0.11$ and was performed in a Windows 7 64-bit system, on a dell workstation with 2 Intel Xeon X5650 2.67 GHZ CPUs and 60.0 GB RAM. The table gave basic metrics of the four networks involved in the simulation and the corresponding update time per event.

Experimental results in Figure 2.3 showed that, as expected, the simulation speed was linear to the network size. In each step of of the Gillespie's Direct Method a node was drawn probabilistically, making time complexity $O(n + log(n))$ for one event.

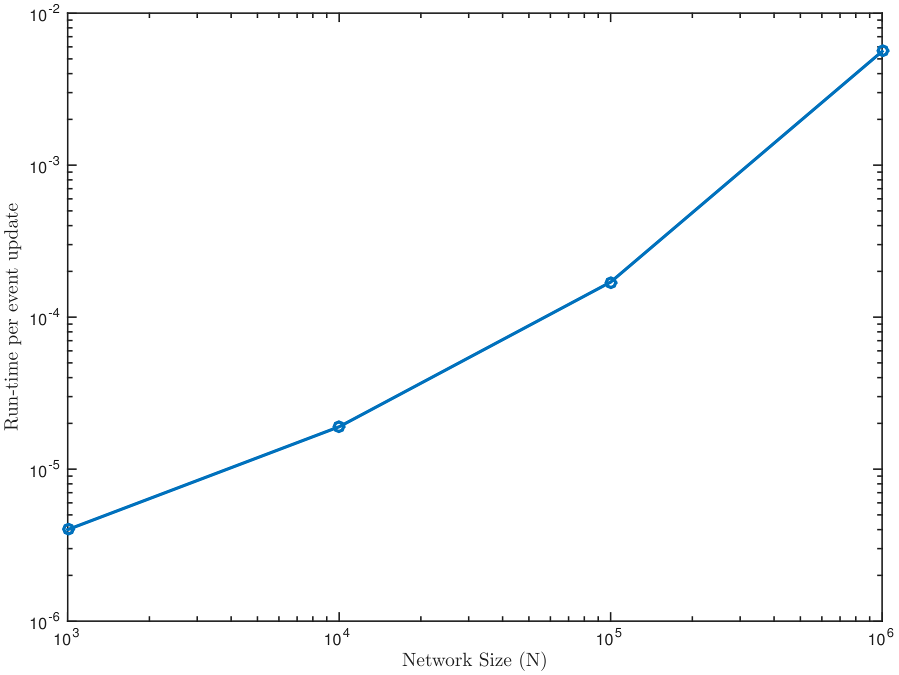


Figure 2.3: Avarage time to simulate an event in the SIS model, where the contact network is assumed to be a geometric network. The average node degree was kept constant for various network sizes.[7]

Chapter 3

Improvement via the new algorithm and parallel computing

Gillespie's Direct Method algorithm is direct because it generates next event m and time τ directly. However, as shown in Figure 2.3, simulation speed is proportional to network size, which is not good enough.

Therefore, this chapter introduces Gibson's Next Reaction Method[4], which is optimal.

3.1 Next Reaction Method

As mentioned in Section 1.2.5, Gillespie developed two algorithms. The Direct Method was explained previously, but the second method, the First Reaction Method (FRM),[2] although inefficient, offers a new solution to the epidemic modeling problem and thus can be optimized significantly. FRM generates a putative time τ_i for each reaction, in which m is the reaction with the least putative time, and τ is the putative time τ_m. Formally, the algorithm for the First Reaction Method is as follows:

Exact Stochastic Simulation First Reaction Method:

1. Initialize (set initial values, set $T \leftarrow 0$).
2. Calculate the propensity function λ_i for all i.

3. For each i, generate a putative time τ_i, according to exponential distribution with parameter λ_i.
4. Let m be the reaction with the lest putative time τ_m.
5. Let τ be τ_m.
6. Change affected λ_is to reflect execution of reaction m. Set $T \leftarrow T + \tau$.
7. Go to Step 2.

For n mutually independent random variables $X_1, X_2, \ldots, X_n$ with $X_i \sim \mathit{exponential}(\lambda_i)$, the distribution of $\mathit{min}\{X_1, X_2, \ldots, X_n\}$ can be recognized as $\mathit{exponential}(\sum_{i=1}^{n} \lambda_i)$. Based on this theorem, the First Reaction Method is equivalent to the previous Direct Method.[2]

Although inefficient in practice, the First Reaction Method can be adapted into an optimal method. In order to save computation, the Next Reaction Method focuses on three activities that occur during every iteration which take time proportional to the number of potential events, n:

1. Updating all n $\lambda_i s$
2. Generating a putative time τ_i, for each $i \in \{1, 2, \ldots n\}$
3. Identifying the smallest putative time τ_m among $\tau_i, i \in \{1, 2, \ldots, n\}$

In order to prevent unnecessary computation, the Next Reaction Method incorporates the following main ideas:

1. Store τ_i, not just λ_i.
2. Be extremely sensitive in recalculating λ_i (and τ_i); recalculate only if it changes.
3. Reuse τ_is where appropriate.

The Next Reaction Method is formally expressed as follows:

1. Initialize (set initial values, set $t \leftarrow 0$).
2. Calculate the propensity function λ_i for all i.

3. For each i, generate a putative time τ_i according to an exponential distribution with parameter λ_i.
4. Let m be the reaction whose putative time τ_m is least.
5. Let T be τ_m.
6. Change affected λ_is to reflect execution of reaction m.
7. For all node $j \in \{\text{neighbours of } m\}$, set $\tau_j \leftarrow (\lambda_{j,old}/\lambda_{j,new})(\tau_j - T) + T$.
8. Set $\tau_m \leftarrow \rho + T$, where ρ is a random number generated according to an exponential distribution with parameter λ_m.
9. Go to Step 4.

Although random numbers often cannot legitimately be reused since Monte Carlo simulations assume them to be statistically independent, it is legitimate in this particular special case. Gibson proved[4] that for all $i \neq m$ where m is the next event, τ_i is distributed according to

$$Pr(T_i > u) = \begin{cases} exp(-\lambda_{i,n}(u - t_{n+1})) & u > t_{n+1} \\ 1 & otherwise, \end{cases}$$

thereby making the Next Reaction Method equivalent to the First Reaction Method, switching from relative time to absolute time. Meanwhile, for those $i \neq m$ whose λ_i remains constant from the n^{th} to $n+1^{th}$ iteration, $\lambda_{i,n+1} = \lambda_{i,n}$, preventing the need to change these $\tau_i s$.

Experiment results showed that the Next Reaction Method obtained the same result as the Direct Method, as shown in Figure 3.1. However, appropriate data structures must be used to store λ_is (and τ_is) so that those that change can be updated efficiently. Since the τ_is are frequently updated but only the least is read, binary heap is the optimal choice with $O(log(n))$ for updating and $O(1)$ for least value reading.

Figure 3.2 compares simulation results with Gillespie's Direct Method. The simulation was performed on a server with Intel Xeon E5-2630 CPUs and 4GB memory limitation for each process. The speed of Gibson's Next Reaction Method was nearly network size free.

It is easy to deduce from the NRM algorithm that the number of update operations

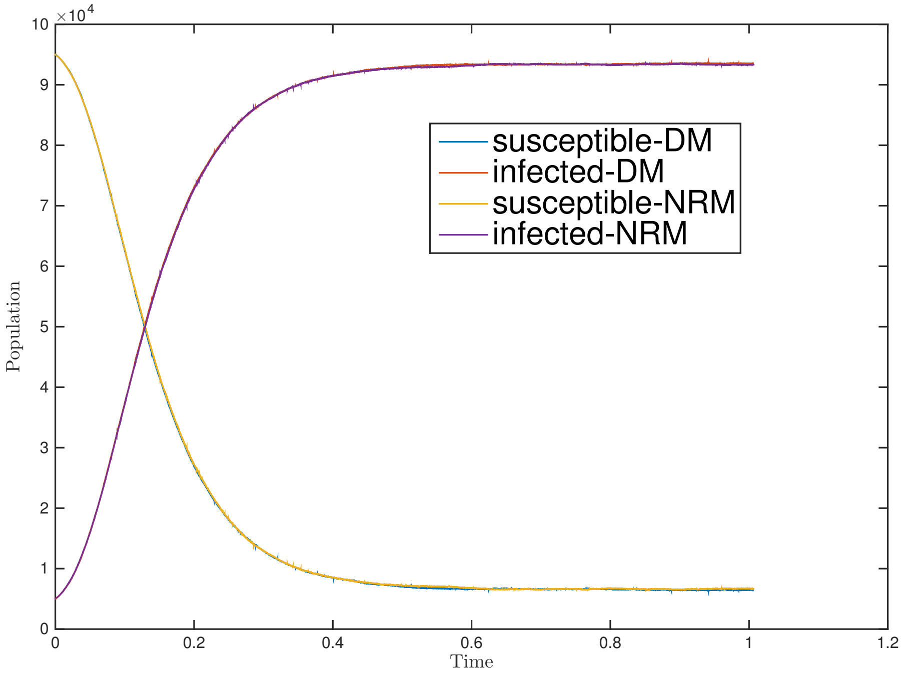


Figure 3.1: Simulation result of the Direct Method and the Next Reaction Method

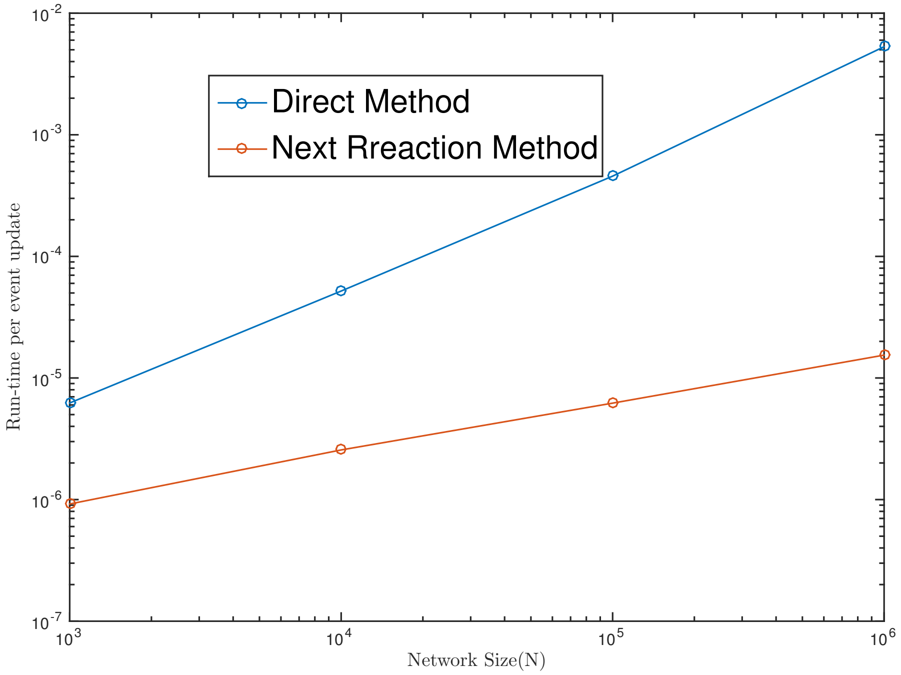
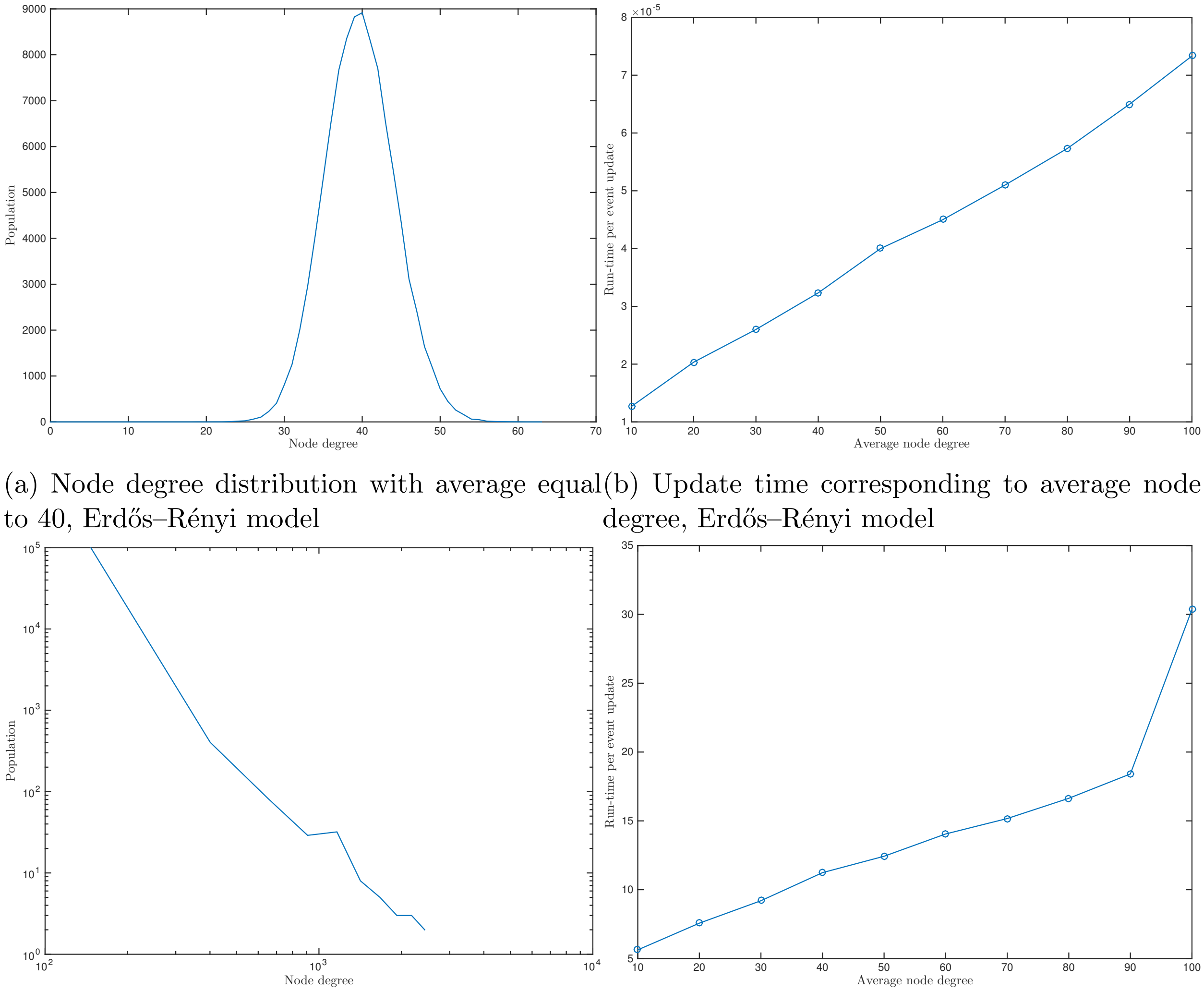


Figure 3.2: Speed comparison of the Direct Method and the Next Reaction Method

is proportional to the average node degree since only neighbours are affected in each step. Therefore I ran simulations based on two network models: the Erdős–Rényi model and the Barabasi-Albert model. For each model, simulations were run over 10 networks with identical sizes of 10^5 with average node degree varying from 10 to 100 at step 10. Node degree distribution was similar within networks of the same model. One sample with average node degree equal to 40 is shown in Figure 3.3a and Figure 3.3c. Simulation results are shown in Figure 3.3b and Figure 3.3d. This experiment perfectly supported the inference.



(a) Node degree distribution with average equal to 40, Erdős–Rényi model

(b) Update time corresponding to average node degree, Erdős–Rényi model

(c) Node degree distribution with average equal to 40, Barabasi-Albert model

(d) Update time corresponding to average node degree, Barabasi-Albert model

Figure 3.3: Simulation on networks with node number= 10^5 and average node degree varying from 10 to 100 for the Erdős–Rényi model and the Barabasi-Albert model

3.2 Packing into Python library

Python is a script language with implementation written in C. The Python application programmers interface (API) defines a set of functions, macros, and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header ”Python.h”.

In order to make the program useful in Python, I extracted the simulation function and transplanted to Python 3 by writing an interface function. The main task of this interface function is to translate Python variables to C types and pass it to the C simulation function. I also imposed some validity checks. In order to increase process efficiency, I tried to avoid memory manipulation. I adopted the Numpy library to construct an adjacency network identical to C, allowing the memory address to be passed directly, quickly, and efficiently to the C function.

3.3 Packing into R library

Although an R library is constructed similarly to a Python library, several practical differences exist in implementation. The interface between C and R is simpler than the interface between C and Python. Technically, a C standard shared library can be called directly in R, but an interface programme is still necessary due to the complex data type of the simulation function. The main obstacle is that R does not support C format struct. Therefore, I packed parameters with similar properties into one vector and maintained the large vectors.

By applying **rapply(Para,c)** to the Para variable from the original R version GEMF-sim, all parameters were transformed into one real-type vector. Because this vector contains all transition information, it can be recovered after passed to C, without additional configuration. Since no struct type was present, network input and simulation output must be passed separately. Three vectors represented i, j, and $weight$ and one vector defined **stop condition** as input.

Output varied according to transition matrices and stop condition. For a single simulation, four vectors stored **time, node, old status, and new status** of each event in addition to an integer n that indicated the number of events comprising the output. Aside from time vector and event number, a vector indicating the population at each time point for every compartment was expected if the simulation was run multiple times. In order to avoid variable parameter numbers in the interface, I concatenated all integer vectors into one single vector, allowing simple extraction of expected information when the vector was split with event number n. In this way, a majority of the preparation procedure from the original R version can be used to call this C function.

3.4 Parallel computation

Multi-thread parallel computation allows advantageous utilization of multiple cores. This technique is useful for GEMFsim because the simulations always repeat many times since GEMFsim is based on stochastic method. Because the networks remain unchanged during simulation, all threads can share one copy of adjacency lists, thereby saving a lot of space. To ensure stability and robustness of the program, all threads run independently without data synchronization, and read-only data is shared while live simulation status is preserved by each thread. In addition to networks, all threads use initial input status to reset respective values at the initialization of each iteration. After simulations of all threads are complete simulation are gathered results from all threads and the final output is calculated with histogram.

I used POSIX Threads to implement parallel computation. POSIX Threads is a parallel execution model that exists independently from a language. It allows a program to control multiple flows of work that overlap in time. Each flow is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API. The API is widely supported by Unix-like POSIX-conformant operating systems such as FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, and Solaris, making this solution flexible and compatible with different platforms without further requirements.

Simulation results are shown in Figure 3.4. The network reading and time analysis were not accounted for since this period is constant for a fixed network; the simulation time, however, varies according to the simulation condition. If a network with N nodes needs approximately N events to achieve significant results, then the preprocess time should require approximately 1% of the total simulation time. Therefore, I considered only the simulation part in this study.

I used the Beocat cluster from the Computer Science Department to do the experiment. The host I used had 8 Intel Xeon E5-2630 CPUs. I ran simulation with different thread numbers. All four networks are generated from geometric model with an average node degree of 20. Each simulation ran for 120 rounds, with each round generating 10,000 events. All 120 rounds were divided evenly to all threads. Results showed that, compared to one thread only, two threads save time. Increasing threads did not speed up when the network was small, even occasionally slowing down the entire process. When the network was as large as 10^6, additional threads caused faster average speed.

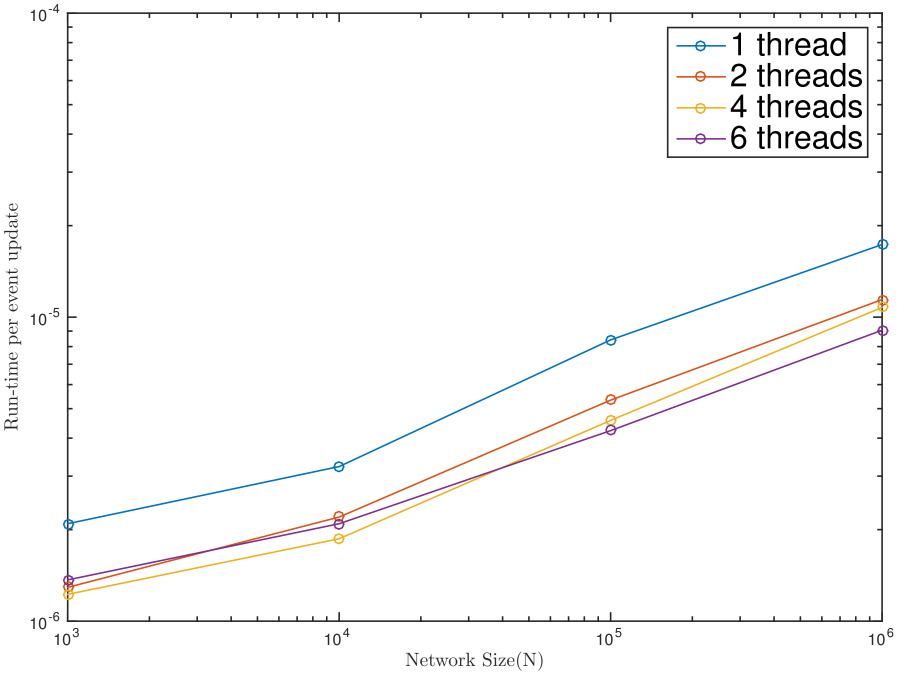
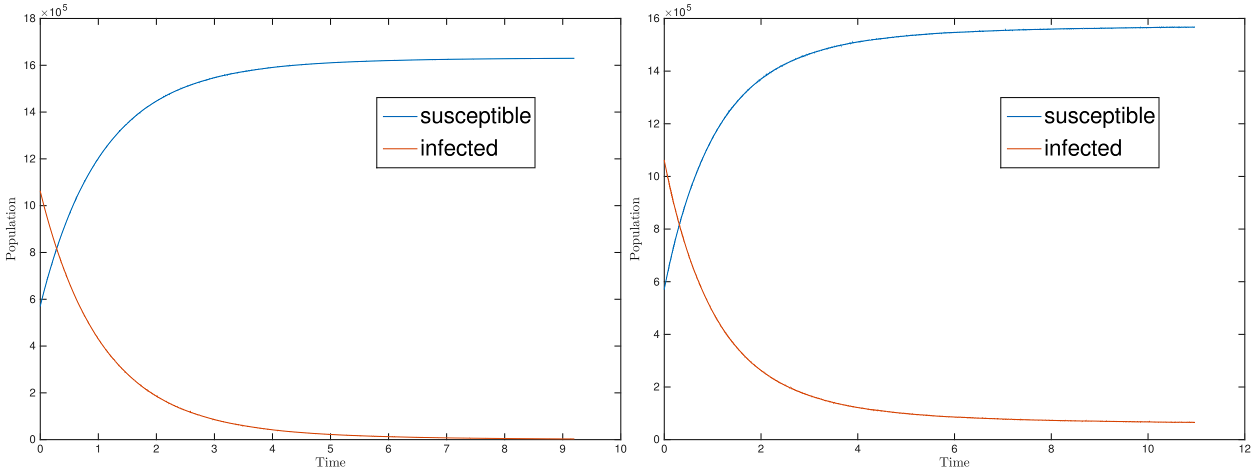


Figure 3.4: Speed comparison: parallel computation with 1, 2, 4, and 6 threads

3.5 Demonstration of phase transition

Van[8] proved the existence of a threshold for an SIS model. If $\tau = \frac{\beta}{\delta}$ is over the threshold τ_c, a fraction of nodes always remains infected. If τ is below the threshold, the disease will die out given enough time. Threshold τ_c is equal to $\frac{1}{\lambda}$, where λ is the spectral radius of the matrix, which represents the contact network.

In order to demonstrate this phase transition phenomenon, I used a real-world network, Pokec social network from Slovakia.[9] Although the network contained 1,632,803 nodes and 30,622,564 edges, the network is weakly all connected but not strongly all connected. Therefore, I made the network undirected for simulation. After transformation the network contained 22,301,964 edges. The largest eigenvalue λ was 148, so I tried $\tau = \frac{0.9}{\lambda}$ and $\tau = \frac{2}{\lambda}$. Approximately 1.5 million events occurred in each simulation. In fact, $\tau = \frac{1.1}{\lambda}$ also obtained phase transition. However, the population of infected state was only several hundreds, which is too small to make a difference in the figure. Therefore, I chose $\tau = \frac{2}{\lambda}$. Simulation results are shown in Figure 3.5



(a) $\frac{\beta}{\delta} = \frac{0.9}{\lambda} = 0.6$

(b) $\frac{\beta}{\delta} = \frac{2}{\lambda} = 1.4$

Figure 3.5: Demonstration of phase transition

Chapter 4

Conclusion

This report introduced the GEMFsim tool and described its implementation into C language. An improved algorithm was applied to the tool, implementation was packed into Python and R libraries, and parallel computing was utilized. As a generalized model, GEMFsim can be applied to various areas, thereby affirming the usefulness of this study's attempt to improve the tool.

Chapter 1 introduced nodal transitions and edge-based transitions, as well as Gillespie's Direct Method algorithm[2], which is the basis of GEMFsim. Chapter 2 described implementation of GEMFsim in C language, including explanation of program structure and basic API. Details are included in Appendix A and Appendix B. Using Gillespie's Direct Method, C language implementation allowed the tool to work for a large network and significantly improve efficiency of the framework. However, the algorithm restricted the simulation speed to linearity with network size, which is unacceptable for super large networks. Section 3.1 introduced a new algorithm, Gibson's Next Reaction Method[4], which generates exact results but does not correlate speed to network size. This new algorithm was optimized from Gillespie's First Reaction Method[2]. Switching from relative time to absolute time and reusing unchanged rates retained a majority of the computation, and only neighbors of the node involved in current event required updating regardless of the network's size. A binary heap was used to store all rates since the least value was the only pertinent value.

As a low-level language, the C language offers efficient execution but inconvenient usage; consequently, Section 3.2 and 3.3 explained how the C program could be useful in Python and R environment. This report also explained the ideology of the framework design, which allows users to obtain high-speed simulation with handy languages.

Although the simulation was sequential as a Markov chain process, meaningful results required many simulations under identical initial conditions. Section 3.4 applied parallel computing to the program. Running multiple simulations simultaneously allowed advantageous utilization of multi-core CPU, resulting in increased speed. In addition, all threads shared one copy of network structure since it remained unchanged throughout the entire process, which is memory efficient.

Bibliography

[1] Faryad Darabi Sahneh. *Spreading processes over multilayer and interconnected networks.* PhD thesis, Kansas State University, 2014.

[2] Daniel T Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *Journal of computational physics*, 22(4):403–434, 1976.

[3] Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.

[4] Michael A Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The journal of physical chemistry A*, 104 (9):1876–1889, 2000.

[5] Faryad Darabi Sahneh, Caterina Scoglio, and Piet Van Mieghem. Generalized epidemic mean-field model for spreading processes over multilayer complex networks. *IEEE/ACM Transactions on Networking*, 21(5):1609–1620, 2013.

[6] GEMF C Manual. http://www.ece.k-state.edu/netse/files/GEMF_C_Manual.pdf, 2016. Accessed: 2016-11-07.

[7] Faryad Darabi Sahneh, Aram Vajdi, Heman Shakeri, Futing Fan, and Caterina Scoglio. Gemfsim: A stochastic simulator for the generalized epidemic modeling framework. *arXiv preprint arXiv:1604.02175*, 2016.

[8] P. Van Mieghem. Epidemic phase transition of the sis type in networks. *EPL (Europhysics Letters)*, 97(4):48004, 2012. URL http://stacks.iop.org/0295-5075/97/i=4/a=48004.

[9] Pokec. https://snap.stanford.edu/data/soc-pokec.html, 2016. Accessed: 2016-11-07.

Appendix A

Sample configuration file

```
[DATA_FILE]
ContactNetwork1.txt
ContactNetwork2.txt

[STATUS_FILE]
status_SAIS.txt

[OUT_FILE]
output_file.txt

[DIRECTED]
1

[STATUS_BEGIN]
1

[MAX_TIME]
```

```
200.000


[MAX_EVENTS]

30000


[SIM_ROUNDS]

100


[INTERVAL_NUM]

1000


[NODAL_TRAN_MATRIX]

0.0     0     0

0.3     0     0

0.0     0     0


[EDGED_TRAN_MATRIX]

0 0.94     0

0 0.00     0

0 0.47     0


0     0 0.02

0     0 0.00

0     0 0.00


[INDUCER_LIST]

2 2
```

Appendix B

Data structure

```
typedef struct {
    //adjacency list i->j
    NINT i;
    NINT j;
} Edge;

typedef struct {
    //weighted adjacency list i->j and weight w
    NINT i;
    NINT j;
    double w;
} Edge_w;
```

```
typedef struct {
    Edge **edge;
    Edge_w **edge_w;
    //number of nodes
    NINT V;
    //nodes start from _s, end at _e - 1
    NINT _s;
    NINT _e;
    //edge number list for each layer
    size_t *E;
    //weighted flag, 0 for unweighted, weighted otherwise
    int weighted;
    //directed flag, 0 for undirected, directed otherwise
    int directed;
    //number of layers
    size_t L;
} Graph;

typedef struct{
    //number of compartments
    size_t M;
    //number of layers
    size_t L;
    //compartments start from _s, end at _s+M -1
    size_t _s;
    //2D array, (_s+M+1) by (_s+M)
    double **nodal_trn;
    //3D array, L by (_s+M+1) by (_s+M)
    double ***edge_trn;
    //1 by L array, inducer for each layer
    size_t *inducer_lst;
} Transition;

typedef struct{
    //number of compartments
    size_t M;
    //compartments start from _s, end at _s+M -1
    size_t _s;
    //number of nodes
    NINT _node_V;
    //nodes start from _node_s, end at _node_e - 1
    NINT _node_s;
    NINT _node_e;
    //1 by _node_e list, initial status for each node
    size_t *init_lst;
    //1 by (_s+M) array
    //the population of each compartment
    NINT *init_cnt;
} Status;

typedef struct{
    //arbitrary stop time
    double max_time;
    //maximum events number
    size_t  max_events;
    //number of rounds
    size_t sim_rounds;
    //sampling interval number for
    //multiple simulation( histogram like)
    size_t interval_num;
    char *out_file;
}Run;
```