Dynamics of Negotiation in a Network of Intelligent Software Agents: Technical Report[*]

Minh Van Nguyen
nguyenminh2@gmail.com

27 February 2009

1 Early studies of social networks

Milgram [8] is one of the early quantitative studies of the structure of social networks. The study describes an experiment in which Milgram wished to send a number of letters to his friend in another city. The letters were first distributed to a random selection of people. These people were instructed to deliver the letters to the addressee, under the conditions that the letters must be passed from person to person, and the passers were permitted to only deliver the letters to people whom they knew on a first-name basis. For those letters that eventually reached the intended addressee, it was found that on average six steps were required for a letter to reach its destination. The path length of six within social networks is colloquially known as the "six degrees of separation". Within mathematical circles, a similar type of social network is found in the scientific collaboration network of Erdös numbers [2].

2 Watts-Strogatz small-world networks

Watts and Strogatz [11] study a class of networks that has become known as small-world networks. The Watts-Strogatz model considers a generic graph G having N vertices and K edges, and satisfying the following properties:

1. G is an unweighted or topological graph.
2. G is simple in that it has no loops and no multiple edges.
3. G is sparse in the sense that $K \ll \dfrac{N(N-1)}{2}$.
4. G is connected such that there is a path between any distinct pair of vertices.

[*]The author acknowledges financial support from the Commonwealth Scientific and Industrial Research Organisation (CSIRO) during this project. The project was carried out while the author was a 2008/2009 Summer Research Scholar within CSIRO's Mathematical and Information Sciences division, supervised by CSIRO research scientist Dr Rodolfo Garcia-Flores.

For a random graph, the quantities N and K must satisfy

$$N \gg K \gg \ln(N) \gg 1$$

where $K \gg \ln(N)$ guarantees that the graph is connected [1].

2.1 Characteristic path length and clustering coefficient

Watts and Strogatz [11] analyze the structure of such a network by means of two quantities: the characteristic path length L; and the clustering coefficient C. Let $\{d_{ij}\}$ be the geodesic matrix of G, i.e. the matrix of shortest edge counts between pairs of vertices in G. Then the characteristic path length L is defined as the average shortest path between distinct pairs of vertices in G:

$$L(G) = \frac{1}{N(N-1)} \sum_{i \neq j \in V(G)} d_{ij} \tag{1}$$

which is a global property of G. Furthermore, Watts and Strogatz also consider a local property of G, called the clustering coefficient. To define the clustering coefficient of G, they first introduce the local clustering coefficient C_i of vertex i:

$$C_i = \frac{K_i}{N_i(N_i - 1)/2}$$

where K_i is the number of edges in the graph of immediate neighbours of i and N_i is the number of immediate neighbours of vertex i. The graph of immediate neighbours of i is a subgraph of G. It consists of all vertices ($\neq i$) that are adjacent to i, preserving the adjacency relation among those vertices as found in the supergraph G. Then the clustering coefficient C of G is defined by

$$C(G) = \frac{1}{N} \sum_{i \in V(G)} C_i$$

where the sum is taken over all vertices i of G. The quantity C can be interpreted to mean the average cliquishness of vertices in G, hence C is known as a local property of G.

2.2 The Watts-Strogatz model

In [11], Watts and Strogatz propose an edge rewiring method for constructing a class of graphs that interpolate between a regular lattice and a random graph. Known as the Watts-Strogatz model, the method starts with a one-dimensional lattice G having N vertices, periodic boundary conditions, and each vertex connecting to its k neighbours for some even k. Identify the vertex set $V(G)$ with the elements of the ring $\mathbb{Z}/N\mathbb{Z}$ for some fixed integer $N > 2$. The lattice can be conceptualized as a circulant graph, where each vertex $i \in \mathbb{Z}/N\mathbb{Z}$ is linked by an edge with each of the vertices $i + j$ and $i - j$ for each $j \in \{1, 2, \ldots, k/2\}$, where vertex arithmetic is performed modulo N. We refer to such a graph as a k-circulant graph on N vertices, or a ring lattice of N nodes and per-vertex degree k. Small-world networks are graphs that are intermediate between regular ring lattices and Erdös-Rényi [3] random graphs. Figure 1 illustrates the interpolation from a ring lattice with rewiring probability $p = 0$ to an Erdös-Rényi random graph where the rewiring probability is $p = 1$. The graphs are produced using Sage's [10] interface to the NetworkX [4] Python package.

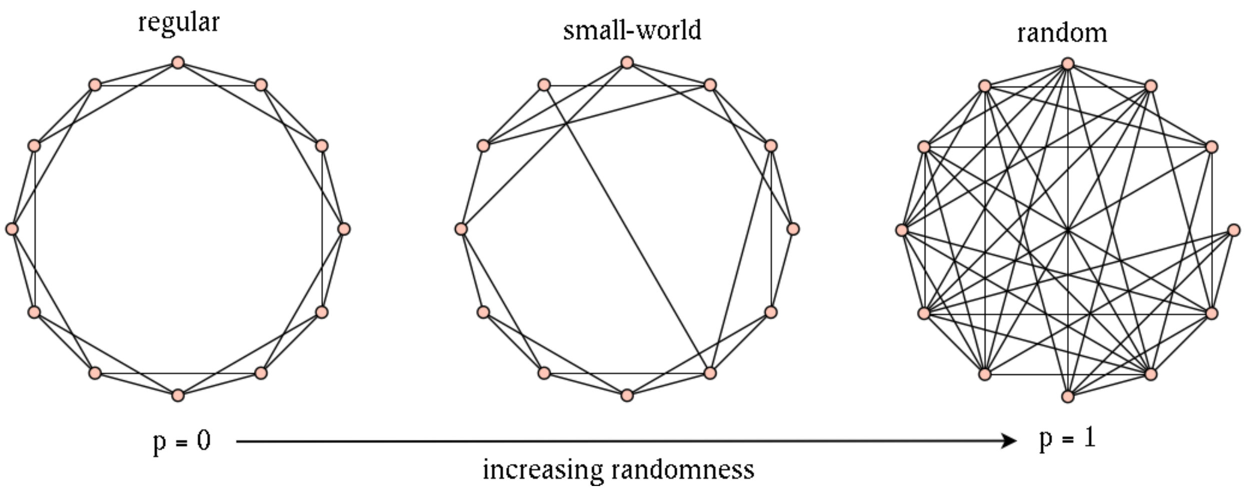

Figure 1: From a regular ring lattice (left) to a random graph (right).

Given a k-circulant graph on N vertices, the Watts-Strogatz rewiring procedure is as follows. Let the probability of choosing a vertex be uniformly distributed. Rewire each vertex with probability p to another vertex chosen at random. The rewiring must result in a graph that:

1. has no multiple edges;
2. has no loops; and
3. the number of edges does not change.

The Watts-Strogatz model does not specifically require that a rewired graph be connected, hence the result of one round of random edge rewiring may be a disconnected graph. However, by definition of the characteristic path length in (1), the underlying graph must be connected, otherwise the geodesic matrix $\{d_{ij}\}$ has ∞ as one of its entries.

Figure 2 shows a plot of the characteristic path lengths and clustering coefficients normalized. The horizontal axis follows a log scale. The plotted metrics were obtained in an effort to verify by computer simulation results reported in [11]. The ring lattice in question is a 10-circulant graph on 1000 vertices with 37 rewiring probability points. The rewiring probabilities are chosen as follows. Let G be a k-circulant graph on N vertices. For $i = 1, 2, \dots, r$ the i-th rewiring probability p_i is given by

$$p_i = p_{\min} \times F^{i-1} \qquad \text{with} \qquad F = \left(\frac{p_{\max}}{p_{\min}}\right)^{1/(r-1)} \tag{2}$$

where $p_{\min}$ and $p_{\max}$ are the minimum and maximum rewiring probabilities, respectively.

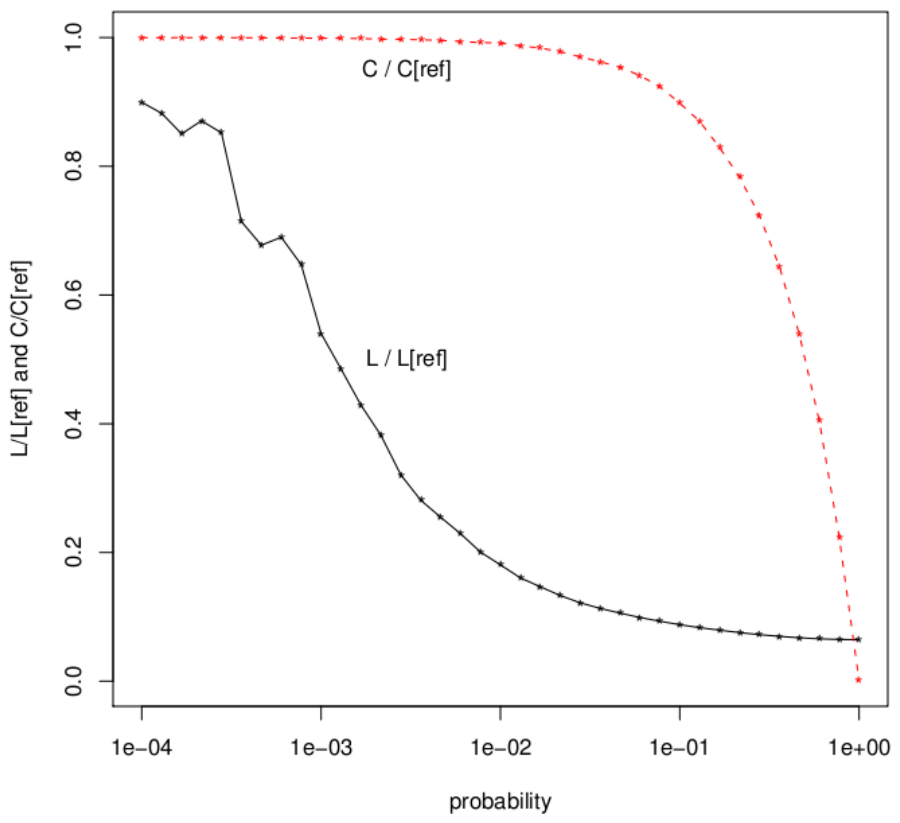

Figure 2: Normalized characteristic path lengths and clustering coefficients.

Next, we describe the procedure for normalizing L and C. Let B be the number of batches of r ring lattices to be rewired with probabilities chosen according to (2). That is, each batch contains r k-circulant graphs on N vertices and the i-th graph from each batch is to be rewired with probability p_i. In particular, our computer simulation rewired a total of $Br = 20 \times 37 = 740$ ring lattices. Define G_{p_i} as the connected graph resulting from rewiring G with probability p_i. For each rewiring probability p_i, define the normalized characteristic path length (respectively clustering coefficient) by

$$\mathrm{norm}_{p_i}(L) = \frac{1}{B}\sum_{G_{p_i}} \frac{L(G_{p_i})}{L(G)} \qquad \text{and} \qquad \mathrm{norm}_{p_i}(C) = \frac{1}{B}\sum_{G_{p_i}} \frac{C(G_{p_i})}{C(G)} \tag{3}$$

where each sum is taken over all graphs G_{p_i}. From Figure 2, we note that there is a range of rewiring probabilities that result in connected graphs with high C and a rapid decrease in L. This is qualitatively consistent with results reported in [11]. The decrease in L is attributed to a number of vertices with links to distant vertices, while the value of C remains high because only a relatively small proportion of vertices have long-range connections. This phenomenon of graphs having the twin characteristics of high cliquishness and low average path length is referred to as the small-world effect.

3 Generalizing the Watts-Strogatz model

Whereas [11] uses the characteristic path length L and clustering coefficient C to study small-world networks, Latora and Marchiori [7] generalize the method by using the notions of local and global efficiencies as defined in section 3.1. The generalization is applicable to both directed and undirected graphs, as well as weighted and unweighted graphs. For weighted graphs, the weight can be a cost associated with the edge connecting two vertices. A graph G with low cost is said to be economic, while G is said to exhibit small-world behaviour provided that it has high efficiency at both the local and global levels. If G has these two properties—both economic and efficiency—then it is referred to as an economic small-world.

3.1 Global and local efficiencies and network cost

Let G be a graph (either weighted or unweighted) having N vertices and K edges. To define local and global efficiencies, Latora and Marchiori [7] introduce the concept of average efficiency. If i and j are distinct vertices of G, let d_{ij} be the shortest path length between i and j. Then the average efficiency of G is

$$E(G) = \frac{1}{N(N-1)} \sum_{i \neq j \in V(G)} \frac{1}{d_{ij}} \tag{4}$$

Let κ_N be the complete graph on N vertices so that $E(\kappa_N)$ is the average efficiency of κ_N. Define the global efficiency of G as

$$E_{\text{glob}} = \frac{E(G)}{E(\kappa_N)} \tag{5}$$

For each vertex i of G, let $G_{\mathbf{i}}$ be the subgraph of neighbours of i. Then vertex i is excluded from the vertex set V_i of $G_{\mathbf{i}}$. Define the local efficiency of G by

$$E_{\text{loc}} = \frac{1}{N} \sum_{i \in V(G)} \frac{E(G_{\mathbf{i}})}{E(\kappa_{|V_i|})} \tag{6}$$

where $|V_i|$ is the cardinality of V_i. Note that the metrics (4), (5) and (6) are also applicable to directed graphs as well as weighted graphs. The cost of G can be defined as

$$C_G = \frac{\displaystyle\sum_{i \neq j \in V(G)} a_{ij}\gamma(\ell_{ij})}{\displaystyle\sum_{i \neq j \in V(G)} \gamma(\ell_{ij})}$$

where $\{a_{ij}\}$ and $\{\ell_{ij}\}$ are the adjacency and weight matrices of G, respectively. In the Watts-Strogatz model, the weight ℓ_{ij} assigned to the edge connecting vertices i and j is $\ell_{ij} = 1$. The cost evaluator function γ measures the cost needed to set up a connection with a given length. The Watts-Strogatz model assumes γ to be the identity function

$\gamma(\ell_{ij}) = \ell_{ij} = 1$ for all $i \neq j$. Thus $\{a_{ij}\} = \{\ell_{ij}\}$ holds for the specific case of the Watts-Strogatz model and therefore

$$C_G = \frac{\sum\limits_{i \neq j \in V(G)} a_{ij}\gamma(\ell_{ij})}{\sum\limits_{i \neq j \in V(G)} \gamma(\ell_{ij})} = \frac{\sum\limits_{i \neq j \in V(G)} a_{ij}}{\sum\limits_{i \neq j \in V(G)} 1} = \frac{2K}{N(N-1)} \tag{7}$$

However, for weighted graphs Latora and Marchiori [7] define the network cost as

$$C_G = \frac{\sum\limits_{i \neq j \in V(G)} a_{ij}\ell_{ij}}{\sum\limits_{i \neq j \in V(G)} \ell_{ij}} \tag{8}$$

where ℓ_{ij} is defined in (9).

Appendix A contains an R [9] script implementing the Latora-Marchiori metrics for graphs that are unweighted, undirected and connected.

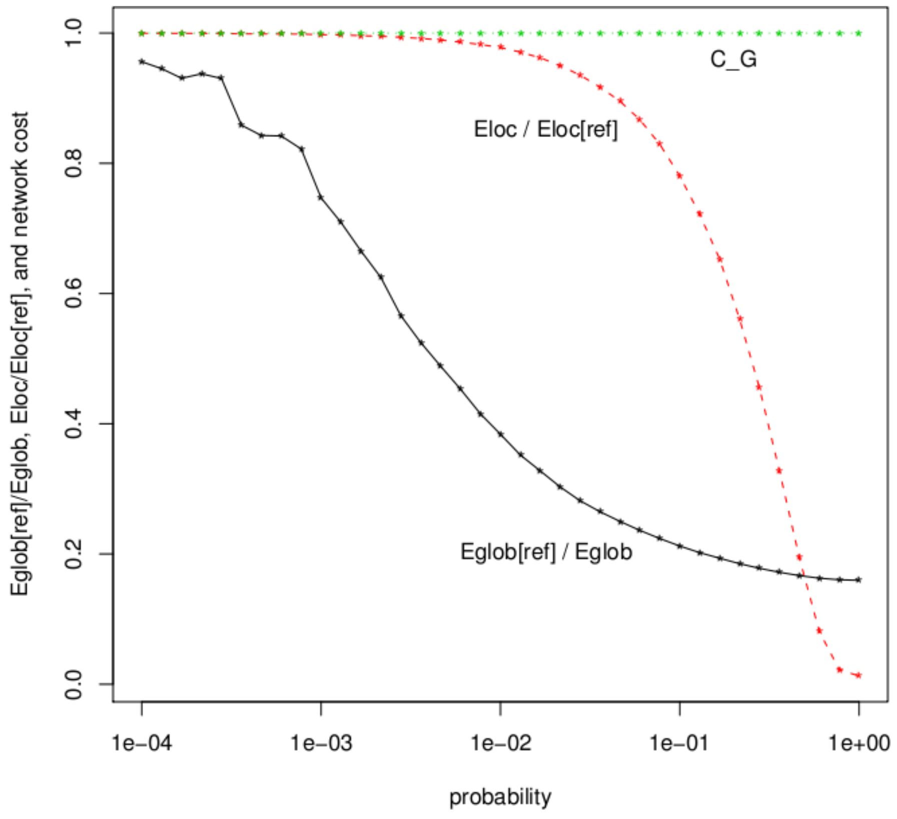

Figure 3: Normalized global and local efficiencies.

Figure 3 shows a plot of the global and local efficiencies normalized, together with normalized network costs. The results are similar to those reported by Latora and Marchiori.

The metrics were obtained from computer simulation of random edge rewiring of 20 batches of 37 ring lattices, each lattice being a 10-circulant graph on 1000 vertices. That is, each batch contained 37 ring lattices and therefore a total of $20 \times 37 = 740$ ring lattices to be rewired. Edges of the i-th ring lattice of each batch were rewired with the probability in (2). The Latora-Marchiori metrics (5) and (6) were then calculated on the rewired 740 graphs. The normalized E_{glob} and E_{loc} corresponding to rewiring probability p_i were obtained using a normalization procedure similar to (3). In particular, let B be the number of batches with r rewiring probabilities chosen according to (2), let G be a k-circulant graph on N vertices, and let G_{p_i} be the connected graph resulting from rewiring G with probability p_i. For each rewiring probability p_i, the normalized local and global efficiencies are defined by

$$\text{norm}_{p_i}(E_{\text{glob}}) = \frac{1}{B} \sum_{G_{p_i}} \frac{E_{\text{glob}}(G)}{E_{\text{glob}}(G_{p_i})} \qquad \text{and} \qquad \text{norm}_{p_i}(E_{\text{loc}}) = \frac{1}{B} \sum_{G_{p_i}} \frac{E_{\text{loc}}(G_{p_i})}{E_{\text{loc}}(G)}$$

where each sum is taken over all graphs G_{p_i} that have been rewired with probability p_i. The normalized network cost is similarly defined by

$$\text{norm}_{p_i}(C_G) = \frac{1}{B} \sum_{G_{p_i}} \frac{C_G}{C_{G_{p_i}}}$$

However, by definition of C_G for unweighted, undirected graphs as specified by (7), it is clear that $\text{norm}_{p_i}(C_G) = 1$ for all rewiring probabilities. Further details can be found in Appendix A.

3.2 Extending the Watts-Strogatz model to weighted networks

This section considers Latora and Marchiori's [7] generalization of the Watts-Strogatz model to the case of weighted, undirected networks. The network is a k-circulant graph on N vertices where $N = 1000$ and $k = 6$. After generating a ring lattice satisfying these parameters, one would get a graph G with $K = 3000$ edges. The Latora-Marchiori approach, as detailed in "Model 4" of [7], is to randomly eliminate $K/2 = 1500$ of the edges of G and then proceed with the rewiring process of the Watts-Strogatz model. The weight of each edge is defined in terms of the Euclidean distance. In particular, if i and j are vertices of G for $i, j = 1, 2, \ldots, N$ then the Euclidean distance between i and j is

$$\ell_{ij} = \frac{2\sin(|i-j|\pi/N)}{2\sin(\pi/N)} = \frac{\sin(|i-j|\pi/N)}{\sin(\pi/N)} \tag{9}$$

Note that the metric (9) is specific to ring lattices. The distance between each pair of neighbouring vertices is $\ell_{ij} = 1$ and the distance from i to itself is trivially $\ell_{ii} = 0$. The weight matrix of G is denoted $\{\ell_{ij}\}$, which has zero along the main diagonal and is symmetric about this diagonal. For unweighted graphs, the geodesic matrix $\{d_{ij}\}$ is a matrix of minimum edge counts separating each pair of vertices i and j. If there are no paths from i to j, where $i \neq j$, then Latora and Marchiori [7] define $d_{ij} = +\infty$. In case $i = j$, then $d_{ij} = 0$. On the other hand, for weighted graphs the weight matrix $\{\ell_{ij}\}$ can be interpreted as the matrix of physical distances. Then d_{ij} is the minimum sum of physical distances from i to j. Furthermore, $d_{ij} = 0$ if $i = j$, and $d_{ij} = +\infty$ whenever there are no paths from i to j.

The following scripts support computer simulation of weighted, undirected networks as described in “Model 4”:

- `k_circulant_n.sage` — This Sage script generates ring lattices, each with half the total number of edges removed.
- `rewire-lattices.r` — This R script can be used to rewire (n, k) ring lattices that have had 50 percent of their total number of edges removed.
- `mat2r.py` — This Python script converts text representation of a Latora-Marchiori network to its R code representation.
- `network-metrics-lm.r` — An R script to compute network metrics of weighted, undirected Latora-Marchiori networks.

Further details on these scripts can be found in Appendix B. Using the above scripts, the computed network metrics are plotted using R and shown in Figure 4. The results are qualitatively similar to those reported in [7].

4 Conclusion & further research

In this paper, we have provided verification of results reported in [7]. The reported results are qualitatively similar to those contained in [7].

In [6], Kaihara formulates the problem of virtual market based supply chain management (SCM) in terms of a discrete resource allocation problem, and proposes an algorithm for SCM under a dynamic environment. The simulation reported in [6] concerns a single input/output circulatory resource flow within a network of two economic agents and two virtual markets.

As a direction for future research that incorporates a network approach to economics, we propose to use the Latora-Marchiori network metrics in computer simulations of a multi-agent network of buyers and sellers. Instead of the edge weight (9), we propose to use a multi-dimensional version of the Cobb-Douglas or constant elasticity of substitution functions [5]. Our research approach has the advantage of generalizing [6] to the case of multiple input and output.

Revision

- 2010-01-09 — Some clarification suggested by David Joyner (US Naval Academy), including: explaining what is the graph of immediate neighbours of a vertex; and some improvements to the exposition of the paper.

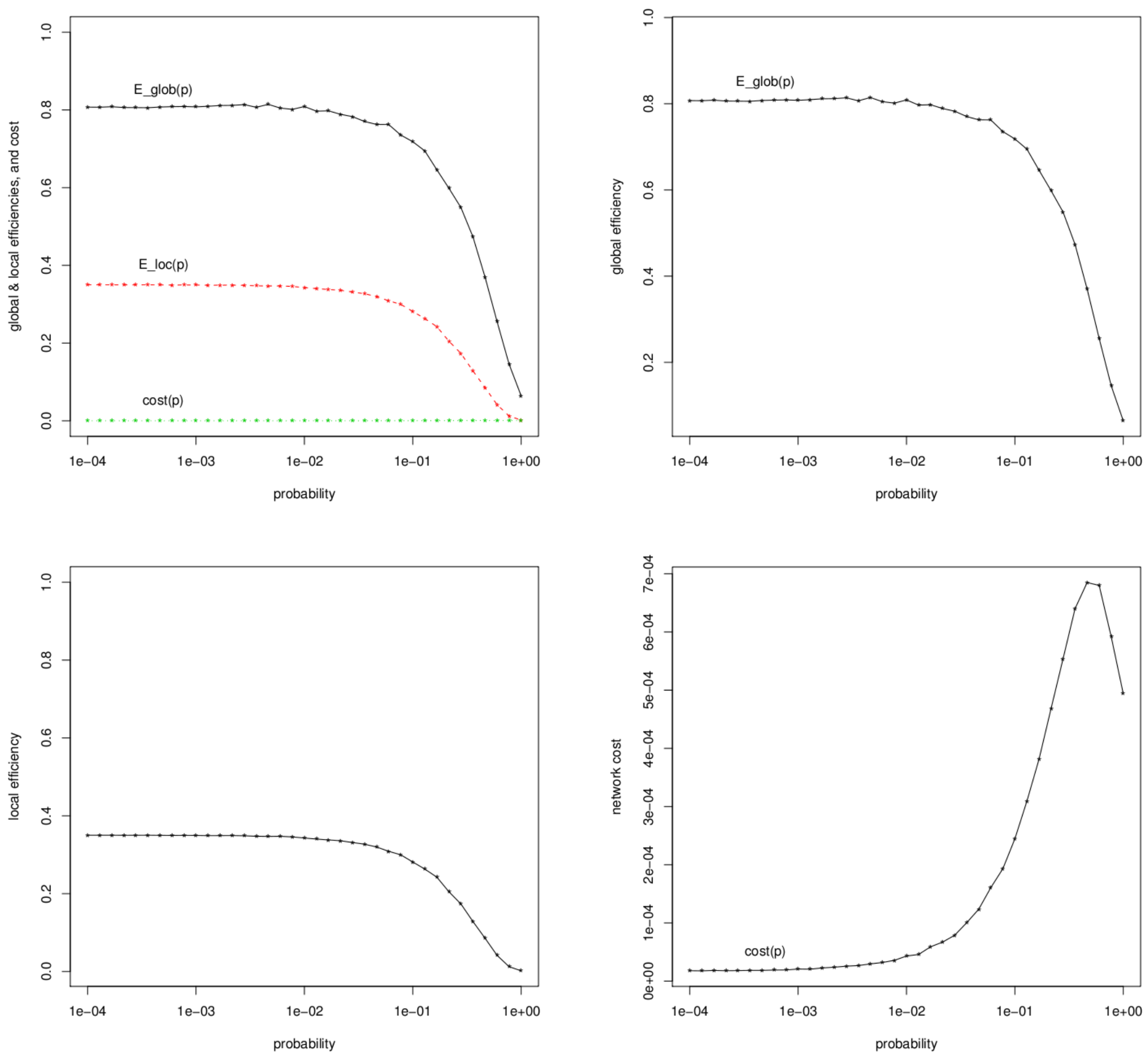

Figure 4: Network metrics for weighted, undirected graphs.

A Latora-Marchiori metrics: unweighted

This appendix presents a script written in the R [9] language for computing local and global efficiencies and network cost of unweighted, undirected graphs. The graphs in question have $N = 1000$ vertices and per-vertex degree $k = 10$. Starting with a 10-circulant graph G on 1000 vertices, the script rewires G with probability p according to the Watts-Strogatz random edge rewiring method to produce a rewired connected graph G'. The global and local efficiencies of G' are then calculated according to (5) and (6), respectively. The network cost is calculated using (7). The 37 rewiring probabilities are chosen according to (2).

```
# sw-latora-marchiori.r -- modelling small-world networks
# Copyright (C) 2008 Minh Van Nguyen <nguyenminh2@gmail.com>
#
# An R script to model small-world networks. This implements techniques
# in the paper:
#
# V. Latora & M. Marchiori. Economic small-world behavior in weighted
# networks. The European Physical Journal B, 32(2):249--263, 2003.
#
# which generalizes the approach described in:
#
# D. Watts & S.H. Strogatz. Collective dynamics of "small-world"
# networks. Nature, 393(4):440--442, 1998.
#
# Rodolfo Garcia-Flores has written an R script that implements the
# Watts-Strogatz model described in (Watts & Strogatz 1998).
#
# Minh Van Nguyen extended Rodolfo's code based on a generalization
# of the Watts-Strogatz model as detailed in the paper
# (Latora & Marchiori 2003).
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.


# Clear memory, removing (almost) everything in the working environment
# without any warning. Be careful with what you wish for.
rm(list = ls())
# For colours and fonts
library(grDevices)
# For fit, simulate and diagnose exponential-family models for networks
library("ergm")
# For social network analysis
library("sna")
# library("igraph")


# ===================================================================
# 1. SETUP (DATA, MODELS AND FUNCTIONS)
# ===================================================================
```

```r


### Problem data


# Actual values should be n = 1000, k = 10. When run with these values,
# the script should take a few hours to complete. Test values can be
# n = 20, k = 4
n <- 1000             # vertices
k <- 10               # edges per vertex, MUST BE EVEN.
timesToRepeat <- 20  # should be 20

# Data for a logarithmically-scaled probability vector.
numberOfPoints <- 37
minProb <- 1e-4
maxProb <- 1

# Directories and file names
# Subdirectory names to organise I/O.
input <- "input"
output <- "output"

# Files in input directory to search for.
# Output file names
summaryFileName <- "small-world-summary.txt"

# Platform-specific directory separator.
slash <- .Platform$file.sep

# (Relative) subdirectory paths.
outputDir <- paste(".", slash, output, slash, sep = "")
inputDir  <- paste(".", slash, input, slash, sep = "")


### Functions


# Simulates a circular list. We are only interested in the index i of a
# member of this list, which has n members. One way to conceptualize this
# list is to visualize all n members as arranged in a cycle graph, in
# which each member i has an edge connecting it to i + 1, and an edge
# connecting it to i - 1. Another way to think about this function is to
# interpret it as a simple implementation of the group Z/Zn, where only
# the index of each i in Z/Zn is returned.
#
# INPUT:
#    index -- integer; index of an element in this circular list.
#    length -- integer > 0; the number of elements in this circular list.
#       FIXME: Maybe it's a good idea to implement the case where
#       length < 0, or provide some sanity checking to take care of that
#       possibility.
#
# OUTPUT:
#    If 0 < index <= length, then return index. If index > length, then
#    return index mod length. Else index < 0, so return index mod length.
#    If length is 0, then return NaN (not a number).
#
# AUTHOR:
#     Rodolfo Garcia-Flores
#     Documentation by Minh Van Nguyen <nguyenminh2@gmail.com>
#
returnIndex <- function(index, length) {
  if ((index > 0) && (index <= length)) {
    index
  }
  else if (index > length) {
    index - length * floor(index / length)
  }
  else {
    (length + index) + length * floor(-1 * index / length)
  }
}

# The re-wiring routine.
#
# INPUT:
#    aMatrix -- an adjacency matrix.
#    aProbability -- double; a probability value p such that 0 < p < 1. This
#       value determines the probability that an edge incident on a vertex
#       is re-wired.
#
# OUTPUT:
#    An adjacency matrix with a number of the vertices re-wired.
#
# AUTHOR:
#     Rodolfo Garcia-Flores
#     Documentation by Minh Van Nguyen <nguyenminh2@gmail.com>
#
reWire <- function(aMatrix, aProbability) {
  currentMat <- aMatrix
  for (i in 1:n) {
    for (j in 1:i) {
      if ((currentMat[i,j] != 0) && (runif(1) < aProbability)) {
        # To vertices different to i and
        # different to those already connected,
        # preferably to nodes that are isolated.
        # This should prevent having isolated regions.
        isolatedNodes <- c(1:n)[colSums(currentMat[,]) == 0]
        nodesAlreadyConnected <- c(1:n)[currentMat[i,] > 0]
        excludedNodes <- c(i, nodesAlreadyConnected)
        notExcludedNodes <- (1:n)[-excludedNodes]

        # A list whose first elements are isolated nodes, the rest are
        # shuffled, not-excluded values.
        validPrioritisedNodes <- c(isolatedNodes,
                                   sample(setdiff(notExcludedNodes,
                                                  isolatedNodes)))

        # Take first node index in list.
        newVertex <- validPrioritisedNodes[1]
        currentMat[i,j] <- 0
        currentMat[j,i] <- 0
        currentMat[i, newVertex] <- 1
        currentMat[newVertex, i] <- 1
      }
    }
  }
  currentMat
}

# The average efficiency E of a graph G, where the graph has N vertices and
# K edges. This function can also be used to calculate the average
# efficiency even if G is a complete graph on N vertices. Such a complete
# graph is also denoted G^{ideal}, i.e. the ideal case where G has all the
# possible
#
# N(N - 1) / 2
#
# edges. Then the global and local efficiencies are defined in terms of E(G)
# and E(G^{ideal}). These notions of efficiency of a graph are defined in
# the paper (Latora & Marchiori 2003).
#
```

```
# INPUT:
#    geodesicMat -- the matrix of the shortest path lengths between pairs of
#       vertices in G. If i and j are vertices of G, then d_ij denotes the
#       shortest path length between i and j. If geodesicMat describes the
#       geodesics of pairs of vertices in an undirected graph, then
#       geodesicMat is symmetric about the main diagonal. Note that
#       geodesicMat must be a square matrix, so that its row and column
#       dimensions both equal the number of vertices in the underlying
#       graph.
#
# OUTPUT:
#    the average efficiency E of the graph G.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
averageEfficiency <- function(geodesicMat) {
  # the number of vertices in geodesicMat
  N <- dim(geodesicMat)[1]
  E <- NULL

  # check for the case that geodesicMat is a 1 x 1 matrix
  if (N == 1) {
    # harmonicSum <- 0
    E <- 0
  }
  else {
    # Compute the harmonic sum of lower triangular matrix of geodesicMat,
    # excluding the main diagonal.
    colLimit <- 0
    colStart <- 1
    rowStart <- 2
    harmonicSum <- 0
    for (row in rowStart:N) {
      colLimit <- colLimit + 1
      for (col in colStart:colLimit) {
        # Avoid the case where there's no path between vertices i and j. If
        # no path exists between i and j, then d_ij = +oo, which is positive
        # infinity. As d_ij -> +oo, then (1 / d_ij) -> 0.
        if (geodesicMat[row, col] != "Inf") {
          harmonicSum <- harmonicSum + (1 / geodesicMat[row, col])
        }
      }
    }

    # compute average efficiency
    E <- harmonicSum / (N * (N - 1))
  }

  E
}

# The global efficiency E_glob of a graph G, where the graph has N vertices
# and K edges. The notion of global efficiency of a graph is defined in the
# paper (Latora & Marchiori 2003). See also the function averageEfficiency,
# which defines the average efficiency of G. The measure E_glob is defined
# as
#
# E_glob = E(G) / E(G^{ideal})
#
# where G^{ideal} is the complete graph on N vertices. Thus E_glob is a
# ratio of the average efficiencies of two types of graphs: (1) the average
# efficiency of G itself; (2) the average efficiency of the complete graph
# on N vertices, which is the number of vertices in G.
#
# INPUT:
```

```
#    geodesicMat -- the matrix of the shortest path lengths between pairs of
#        vertices in G. If i and j are vertices of G, then d_ij denotes the
#        shortest path length between i and j. If geodesicMat describes the
#        geodesics of pairs of vertices in an undirected graph, then
#        geodesicMat is symmetric about the main diagonal.
#
# OUTPUT:
#    the global efficiency E_glob of the graph G.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
globalEfficiency <- function(geodesicMat) {
  # the number of vertices in geodesicMat
  N <- dim(geodesicMat)[1]

  # compute the average efficiency
  aveEfficiency <- averageEfficiency(geodesicMat)

  # Construct the adjacency matrix of a complete graph on N vertices. By
  # definition of complete graphs, a complete graph K_n and its geodesic
  # matrix G_dist are equivalent. That is, K_n and G_dist are copies of
  # each other. Also, G_dist has 1 everywhere, and 0 on the main diagonal.
  gIdealMat <- matrix(nrow = N, ncol = N)
  gIdealMat[,] <- 1
  for (i in 1:N) {
    gIdealMat[i, i] <- 0
  }

  # compute the average efficiency of the complete graph
  aveEfficiencyGIdeal <- averageEfficiency(gIdealMat)

  # compute the global efficiency
  Eglob <- aveEfficiency / aveEfficiencyGIdeal
}

# The adjacency matrix of G_i. If G is an undirected graph and i is a
# vertex of G, then G_i is the subgraph of neighbours of i, excluding i
# itself.
#
# INPUT:
#    aMat -- the adjacency matrix of the graph G.
#    i -- the index of the vertex whose neighbours we want to consider.
#        Let r and c be the row and column dimensions of aMat, respectively.
#        Then 1 < i < r or 1 < i < c.
#
# OUTPUT:
#    the adjacency matrix of G_i.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
neighboursAdjMat <- function(aMat, i) {
  # the row dimension of aMat
  ## rowNum <- dim(aMat)[1]
  # the column dimension of aMat
  colNum <- dim(aMat)[2]

  # find indices of the immediate neighbours of vertex i
  neighIndex <- c()
  for (col in 1:colNum) {
    if (aMat[i, col] == 1) {
      neighIndex <- c(neighIndex, col)
    }
  }

  # Adjacency matrix of neighbours of i, i.e. the adjacency matrix of G_i
  # in the notation of the paper (Latora & Marchiori 2003).
  neighAMat <- matrix(nrow = length(neighIndex),
                      ncol = length(neighIndex))
  neighAMat[,] <- 0
  for (row in 1:length(neighIndex)) {
    for (col in row:length(neighIndex)) {
      if (aMat[neighIndex[row], neighIndex[col]] == 1) {
        neighAMat[row, col] <- 1
        neighAMat[col, row] <- 1
      }
    }
  }

  neighAMat
}

# The local efficiency E_loc of a graph G, where the graph has N vertices
# and K edges. The notion of local efficiency of a graph is defined in the
# paper (Latora & Marchiori 2003). See also the function averageEfficiency,
# which defines the average efficiency of G. The measure E_loc is defined
# as
#
# E_loc = (1/N) \sum_{i \in G} E(G_i) / E(G^{ideal}_i)
#
# where G_i is the subgraph of neighbours of vertex i, and G^{ideal}_i is
# the complete graph on N_i, which is the number of vertices in G_i. Note
# that G_i excludes the vertex i, and only considers the graph formed by
# its immediate neighbours.
#
# INPUT:
#    aMat -- the adjacency matrix of the graph G. If G is an undirected
#       graph, then aMat is symmetric about the main diagonal.
#
# OUTPUT:
#    the local efficiency E_loc of the graph G.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
localEfficiency <- function(aMat) {
  # The number of vertices in the underlying graph G. Thus the column and
  # row dimensions must be equal.
  N <- dim(aMat)[1]

  # summing the ratios (EGi / EIdealGi) for all vertices i
  cumSum <- 0            # the cumulative sum
  EGi <- 0               # average efficiency of G_i
  EIdealGi <- 0          # average efficiency of G^{ideal}_i
  geodesicGi <- 0        # geodesic matrix of G_i
  geodesicIdealGi <- 0   # geodesic matrix of G^{ideal}_i
  idealGi <- 0           # adjacency matrix of G^{ideal}_i
  for (i in 1:N) {
    geodesicGi <- geodist(neighboursAdjMat(aMat, i))$gdist
    EGi <- averageEfficiency(geodesicGi)

    # Construct the adjacency matrix of a complete graph on K_i vertices. By
    # definition of complete graphs, a complete graph K_n and its geodesic
    # matrix G_dist are equivalent. That is, K_n and G_dist are copies of
    # each other. Also, G_dist has 1 everywhere, and 0 on the main diagonal.
    idealGi <- matrix(nrow = dim(geodesicGi)[1], ncol = dim(geodesicGi)[2])
    idealGi[,] <- 1
    for (j in 1:dim(idealGi)[1]) {
      idealGi[j, j] <- 0
    }

```

```r
    geodesicIdealGi <- geodist(idealGi)$gdist
    EIdealGi <- averageEfficiency(geodesicIdealGi)

    # Prevent division by zero, which is possible when EIdealGi = 0. If
    # both EGi and EIdealGi are zero, then we get (0 / 0), which returns
    # a NaN for "not a number". Caution: we need to consider four cases:
    #
    # EGi  EIdealGi
    # ------------
    # 0    0        <- (EGi / EIdealGi) = 0
    # 0    y1       <- (EGi / EIdealGi) = 0
    # x1   0        <- Is it possible to get this case?
    # x2   y2       <- (EGi / EIdealGi) \in RR\{0}
    #
    # where x1, x2, y1, y2 \in RR
    if ((EGi == 0)) {
      cumSum <- cumSum + 0
    }
    else {
      cumSum <- cumSum + (EGi / EIdealGi)
    }
  }

  Eloc <- cumSum / N
}

# The cost of a network G with N vertices and K edges. For now, we assume
# that G is an undirected graph so that its adjacency matrix is symmetric
# about the main diagonal. The generalization of the Watts-Strogatz model
# contained in (Latora & Marchiori 2003) considers directed as well as
# undirected graphs.
#
# INPUT:
#    adjMat -- the adjacency matrix of G. This adjacency matrix must have
#       the same dimensions as the matrix of distances of G.
#    distMat -- the matrix of distances between pairs of vertices. This
#       distance matrix has the same dimensions as the adjacency matrix of
#       G.
#    gamma -- the cost evaluator function, default is "gamma = WS" for the
#       Watts-Strogatz model. TODO: define further models here apart from
#       Watts-Strogatz.
#
# OUTPUT:
#    the cost of the network G.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
networkCost <- function(adjMat, distMat, gamma = "WS") {
  netCost <- 0
  if (gamma == "WS") {
    N <- dim(adjMat)[1]  # the number of vertices
    K <- 0               # the number of edges

    # For an undirected graph G with adjacency matrix adjMat, both G and
    # adjMat are symmetric about the main diagonal. Hence we need only
    # consider either the upper triangular or lower triangular matrices,
    # excluding entries along the main diagonal, in counting the number of
    # edges in G. On the other hand, we can also sum the entries in adjMat
    # and divide the result by 2 to get the number of edges in g.
    K <- sum(adjMat) / 2

    netCost <- (2 * K) / (N * (N - 1))
  }

  netCost
```

```
}

# The main routine. This is where the network metrics are calculated. For
# the Watts-Strogatz model, the network metrics is comprised of the
# characteristic path length L and the clustering coefficient C. As regards
# the generalization of Watts-Strogatz contained in the paper
# (Latora & Marchiori 2003), the network metrics are the local efficiency
# E_loc, the global efficiency E_glob, and the network cost C.
#
# INPUT:
#   regmat -- a regular matrix
#   probabilities -- a set of re-wiring probabilities
#
# OUTPUT:
#
# AUTHOR:
#     Minh Van Nguyen <nguyenminh2@gmail.com>
#
calculateNetworks <- function(regmat, probabilities) {
  # A set of adjacency matrices.
  nets <- array(NA, dim = c(length(probabilities) + 1, n, n))

  # First matrix is the regular matrix.
  nets[1,,] <- regmat[,]

  # Re-wire with probability p.
  # Put this in a function.
  counter <- 1
  for (p in probabilities) {
    reWiredMat <- array(0, dim = dim(regmat))
    while(connectedness(reWiredMat) < 1) {
      reWiredMat <- reWire(regmat, p)
      print(c("Re-wiring with probability ", p))
    }
    nets[counter + 1,,] <- reWiredMat[,]
    # plot(network(reWiredMat, directed=FALSE),
    #      displaylabels=TRUE, mode="circle")
    # par(ask=TRUE)
    counter <- counter + 1
  }

  # This section is for the Latora-Marchiori generalization.
  # Global and local efficiencies
  Eglob <- NULL
  Eloc <- NULL
  for (counter in 1:(length(probabilities) + 1)) {
    Eglob[counter] <- globalEfficiency(geodist(nets[counter,,])$gdist)
    Eloc[counter] <- localEfficiency(nets[counter,,])
  }

  # structure to return
  result <- cbind(Eglob[1] / Eglob, Eloc / Eloc[1])
}


# ================================================================
# 2. MAIN SCRIPT
# ================================================================

# connectivity matrix of a regular network with no loops
regularMatrix <- matrix(0, nrow = n, ncol = n)
for (i in 1:n) {
  for (index in (k / 2):1) {
    # Get right the indexes.
    jplus  <- returnIndex(i + index, n)
    jminus <- returnIndex(i - index, n)
    regularMatrix[i, jplus] <- regularMatrix[i, jminus] <- 1
  }
}
dimnames(regularMatrix)[[2]] <- paste("node", (1:n), sep = "-")
dimnames(regularMatrix)[[1]] <- paste("node", (1:n), sep = "-")

# logarithmically-scaled probability vector
factor <- (maxProb / minProb)^(1 / (numberOfPoints - 1))
probs <- NULL
for (pt in 1:numberOfPoints) {
  probs[pt] <- minProb * factor^(pt - 1)
}

# Call main routine here timesToRepeat times.
# variable aliases
GE <- 1    # global efficiency
LE <- 2    # local efficiency

# Remember, the first probability is zero, i.e. the regular matrix.
results <- array(NA, dim = c(timesToRepeat, length(probs) + 1, 2))
dimnames(results)[[3]] <- c("L / LRef", "C / CRef")
dimnames(results)[[2]] <- paste("prob", c(0, probs), sep = "-")
dimnames(results)[[1]] <- paste("experiment", (1:timesToRepeat), sep = "-")
for (experiment in 1:timesToRepeat) {
  print(c("Executing experiment ", experiment))
  results[experiment,,] <- calculateNetworks(regularMatrix, probs)
}

# averages
GEmeans <- colSums(results[,,GE]) / timesToRepeat
LEmeans <- colSums(results[,,LE]) / timesToRepeat

# Write table of results
summary <- cbind(c(0, probs), GEmeans, LEmeans)
write.table(format(summary, digits = 6, nsmall = 4, justify = "left"),
            file = paste(outputDir, summaryFileName, sep = ""), sep = "\t")

# plot variables
xdata <- probs
# All except the first value, where prob = 0.
ydata <- cbind(GEmeans[-1], LEmeans[-1])
matplot(xdata, ydata, log = "x",
        main = "Global efficiency and local efficiency",
        xlab = "prob",
        ylab = "Eglob[ref] / Eglob and Eloc / Eloc[ref]",
        type = "b")
```

B Latora-Marchiori metrics: weighted

This appendix presents a number of scripts written in Python, R [9] and Sage [10] for computing local and global efficiencies and network cost of weighted, undirected graphs. The graphs in question have $N = 1000$ vertices and per-vertex degree $k = 10$. The Sage script `k_circulant_n.sage` generates 10-circulant graphs on 1000 vertices, each such graph with half the total number of edges removed. Using the R script `rewire-lattices.r`, the generated graphs can be randomly rewired according to the Watts-Strogatz random edge rewiring method. The rewired graphs are output to disk in matrix (plain textual) notation. These graphs can be converted to its R code representation using the Python script `mat2r.py`. Finally, using the script `network-metrics-lm.r`, the global and local efficiencies of the rewired graphs can be calculated according to (5) and (6), respectively, taking into account the case that we are dealing with weighted graphs. The network cost is calculated using (8). The following sections list the contents of the above scripts.

B.1 The script `k_circulant_n.sage`

```
# ----------------------------------------------------------------------------
# k_circulant_n.sage
# Copyright (C) 2009 Minh Van Nguyen <nguyenminh2@gmail.com>
#
# This Sage script generates ring lattices, each with half the total number
# of edges removed. Such graphs can then be rewired as per Watts & Strogatz.
# Note that this script does not consider the problem of random edge
# rewiring. This script was written and tested using Sage 3.2.x. For more
# information about Sage, please visit www.sagemath.org. Before running this
# script, make sure that a directory named "networks-half-edges-r" exists
# in the current directory.
#
# REFERENCES:
# [1] V. Latora & M. Marchiori. Economic small-world behavior in weighted
#     networks. The European Physical Journal B, 32(2):249--263, 2003.
#
# [2] D. Watts & S.H. Strogatz. Collective dynamics of "small-world"
#     networks. Nature, 393(4):440--442, 1998.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
# ----------------------------------------------------------------------------


def to_r(e, g):
    """
    Generates R code representation of the (n, k) ring lattice g.

    The (n, k) ring lattice g is assumed to have half the total number of
    its edges already removed. The R code matrix representation of g is an
    adjacency matrix and is written to a text file.

    INPUT:
        e -- a positive integer index used for name the R script
            containing R code matrix representation of g.
        g -- an (n, k) ring lattice with half of its total number of edges
            removed.

    OUTPUT:
        Write the R code matrix representation of g to an R script.
    """
    nVertices = g.order()
    outFile = open("networks-half-edges-r/graph-" + str(e) + ".r", "w")
    nrow = g.order()
    ncol = g.order()
    outFile.write("mat <- matrix(nrow = " + str(nrow)
                  + ", ncol = " + str(ncol) + ")\n")
    for i in xrange(nrow):
        row = str(g.adjacency_matrix()[i])
        row = "c" + row
        outFile.write("mat[" + str(i+1) + ",] <- " + row + "\n")
    outFile.close()

def remove_half_edges(n, k):
    """
    Randomly removes half the total number of edges from a k-circulant graph
    with n vertices.

    A k-circulant graph with n vertices is simply a ring lattice with n
    nodes, each of which is connected to its k neighbours. Such a graph
    is also referred to as an (n, k) ring lattice. Such random removal is
    used in "Model 4" of Latora & Marchiori [2].

    INPUT:
        n -- the number of vertices.
        k -- the number of per-vertex degree (must be an even integer).

    OUTPUT:
        An (n, k) ring lattice with half of the total number of its edges
        removed.
    """
    from sage.misc.prandom import choice

    adj = [a for a in xrange(1, k/2+1)]
    G = graphs.CirculantGraph(n, adj)

    # remove half the total number of edges from G
    nEdges = list(G.edge_iterator(labels = False))
    elimTotal = G.size() / 2
    elim = 0
    while elim < elimTotal:
        edge = choice(nEdges)
        G.delete_edge(edge[0], edge[1])
        while not G.is_connected():
            G.add_edge(edge[0], edge[1])
            edge = choice(nEdges)
            G.delete_edge(edge[0], edge[1])
        nEdges = list(G.edge_iterator(labels = False))
        elim += 1

    return G

# As used by Watts & Strogatz [1] and Latora & Marchiori [2].
n = 1000

# As used by Latora & Marchiori [2] in their "Model 4".
k = 6
```

```

# Also known as the number of rewiring probabilities. This number depends
# on how many rewiring probability points you want to use.
nTimes = 37

for i in xrange(nTimes):
    print "[%s] generating network" % (i+1)
    g = remove_half_edges(n, k)
    print "[%s] converting network to R code" % (i+1)
    to_r(i+1, g)
```

B.2 The script rewire-lattices.r

```
# ---------------------------------------------------------------------------
# rewire-lattices.r
# Copyright (C) 2009 Minh Van Nguyen <nguyenminh2@gmail.com>
#
# This R script can be used to rewire (n, k) ring lattices that have had
# 50 percent of their total number of edges removed. The rewiring process
# is per Watts & Strogatz [2]. The resulting rewired networks are used
# in "Model 4" of Latora & Marchiori [1]. Before running this script, make
# sure that a directory named "networks-dat" exists in the current directory.
# For more information about R, please visit www.r-project.org.
#
# REFERENCES:
# [1] V. Latora & M. Marchiori. Economic small-world behavior in weighted
#     networks. The European Physical Journal B, 32(2):249--263, 2003.
#
# [2] D. Watts & S.H. Strogatz. Collective dynamics of "small-world"
#     networks. Nature, 393(4):440--442, 1998.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
# ---------------------------------------------------------------------------


# Some housekeeping before generating random Latora-Marchiori networks.
# Clear memory, removing (almost) everything in the working environment
# without any warning. Be careful with what you wish for.
rm(list = ls())
# For social network analysis
library("sna")
# For various graph-theoretic operations, in particular, weighted shortest
# paths.
library("igraph")


### Functions


# Simulates a circular list. We are only interested in the index i of a
# member of this list, which has n members. One way to conceptualize this
# list is to visualize all n members as arranged in a cycle graph, in
# which each member i has an edge connecting it to i + 1, and an edge
# connecting it to i - 1. Another way to think about this function is to
# interpret it as a simple implementation of the group Z/Zn, where only
# the index of each i in Z/Zn is returned.
#
# INPUT:
#    index -- integer; index of an element in this circular list.
#    length -- integer > 0; the number of elements in this circular list.
#       FIXME: Maybe it's a good idea to implement the case where
#       length < 0, or provide some sanity checking to take care of that
#       possibility.
#
# OUTPUT:
#    If 0 < index <= length, then return index. If index > length, then
#    return index mod length. Else index < 0, so return index mod length.
#    If length is 0, then return NaN (not a number).
#
# AUTHOR:
#    Rodolfo Garcia-Flores
#    Documentation by Minh Van Nguyen <nguyenminh2@gmail.com>
#
returnIndex <- function(index, length) {
  if ((index > 0) && (index <= length)) {
    index
  }
  else if (index > length) {
    index - length * floor(index / length)
  }
  else {
    (length + index) + length * floor(-1 * index / length)
  }
}


# The rewiring routine.
#
# INPUT:
#    aMatrix -- an adjacency matrix.
#    aProbability -- double; a probability value p such that 0 < p < 1. This
#       value determines the probability that an edge incident on a vertex
#       is re-wired.
#
# OUTPUT:
#    An adjacency matrix with a number of the vertices re-wired.
#
# AUTHOR:
#    Rodolfo Garcia-Flores
#    Documentation by Minh Van Nguyen <nguyenminh2@gmail.com>
#
rewire <- function(aMatrix, aProbability) {
  currentMat <- aMatrix
  for (i in 1:n) {
    for (j in 1:i) {
      if ((currentMat[i, j] != 0) && (runif(1) < aProbability)) {
        # To vertices different to i and
        # different to those already connected,
        # preferably to nodes that are isolated.
        # This should prevent having isolated regions.
        isolatedNodes <- c(1:n)[colSums(currentMat[,]) == 0]
        nodesAlreadyConnected <- c(1:n)[currentMat[i,] > 0]
        excludedNodes <- c(i, nodesAlreadyConnected)
        notExcludedNodes <- (1:n)[-excludedNodes]

        # A list whose first elements are isolated nodes, the rest are
        # shuffled, not-excluded values.
        validPrioritisedNodes <- c(isolatedNodes,
                                     sample(setdiff(notExcludedNodes,
                                                    isolatedNodes)))

        # Take first node index in list.
        newVertex <- validPrioritisedNodes[1]
        currentMat[i, j] <- 0
        currentMat[j, i] <- 0
        currentMat[i, newVertex] <- 1
        currentMat[newVertex, i] <- 1
      }
    }
  }

  currentMat
}


# The main routine. This is where the network metrics are calculated. For
# the Watts-Strogatz model, the network metrics is comprised of the
# characteristic path length L and the clustering coefficient C. As regards
# the generalization of Watts-Strogatz contained in the paper
# (Latora & Marchiori 2003), the network metrics are the local efficiency
# E_loc, the global efficiency E_glob, and the network cost C_G.
#
# INPUT:
#    regmat -- a regular matrix.
#    probabilities -- a set of re-wiring probabilities.
#
# OUTPUT:
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
latoraMarchioriGraphs <- function(regMat, probs, experiment) {
  for (p in 1:length(probs)) {
    # Reads in an (n, k) ring lattice which has 50 percent of its total
    # number of edges removed. The matrix is read into memory and named "mat".
    source(paste("networks-half-edges-r/graph-",
                 experiment, "-", p, ".r", sep = ""))

    # rewire with p-th probability
    print(c("rewiring with probability ", probs[p]))
    rewiredMat <- rewire(mat, probs[p])
    while (connectedness(rewiredMat) < 1) {
      rewiredMat <- rewire(regMat, probs[p])
    }
    # The number 1000 refers both to the column and row dimensions of the
    # Latora-Marchiori network.
    write.table(1000, file = paste("networks-dat/graph-",
                        experiment, "-", p, ".dat", sep = ""),
                row.names = FALSE, col.names = FALSE)
    write.table(1000, file = paste("networks-dat/graph-",
                        experiment, "-", p, ".dat", sep = ""),
                row.names = FALSE, col.names = FALSE, append = TRUE)
    write.table(probs[p], file = paste("networks-dat/graph-",
                            experiment, "-", p, ".dat", sep = ""),
                row.names = FALSE, col.names = FALSE, append = TRUE)
    write.table(rewiredMat, file = paste("networks-dat/graph-",
                              experiment, "-", p, ".dat", sep = ""),
                row.names = FALSE, col.names = FALSE, append = TRUE)
  }
}


### Start generate Latora-Marchiori networks here

```

```
# Experimental parameters.
# Actual values should be n = 1000, k = 10 or k = 6. When run with these
# values, the script should take a few hours to complete. Test values can be
# n = 20, k = 4
n <- 1000              # vertices
k <- 6                 # edges per vertex, MUST BE EVEN.
nTimes <- 20           # should be 20

# data for a logarithmically-scaled probability vector
numPoints <- 37
minProb <- 1e-4
maxProb <- 1


# logarithmically-scaled probability vector
factor <- (maxProb / minProb)^(1 / (numPoints - 1))
probs <- NULL
for (pt in 1:numPoints) {
  probs[pt] <- minProb * factor^(pt - 1)
}

# Create rewired networks. The rewired networks are written to text files.
for (experiment in 1:nTimes) {
  print(c("Executing experiment ", experiment))
  latoraMarchioriGraphs(matHalfEdges, probs, experiment)
}
```

B.3 The script mat2r.py

```
# --------------------------------------------------------------------------
# mat2r.py
# Copyright (C) 2009 Minh Van Nguyen <nguyenminh2@gmail.com>
#
# Convert text representation of a Latora-Marchiori network to its R code
# representation. This Python script essentially generates R code to
# represent Latora-Marchiori networks stored in text files. Latora-Marchiori
# networks are (n, k) ring lattices, each with half the total number of its
# edges removed and the resulting network rewired as per Watts & Strogatz [2].
# Such networks are used in "Model 4" of Latora & Marchiori [1]. Before
# running this Python script, make sure that a directory named "networks-r"
# exists in the current directory.
#
# REFERENCES:
# [1] V. Latora & M. Marchiori. Economic small-world behavior in weighted
#     networks. The European Physical Journal B, 32(2):249--263, 2003.
#
# [2] D. Watts & S.H. Strogatz. Collective dynamics of "small-world"
#     networks. Nature, 393(4):440--442, 1998.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
# --------------------------------------------------------------------------

```

```

nExperiments = 20
nProbs = 37
for e in xrange(1, nExperiments + 1):
    print "generate R code for networks in experiment %s" % e
    for p in xrange(1, nProbs + 1):
        inFile = open("networks-dat/graph-"
                      + str(e) + "-" + str(p) + ".dat", "r")
        outFile = open("networks-r/graph-"
                       + str(e) + "-" + str(p) + ".r", "w")
        nrow = int(inFile.readline().strip())
        ncol = int(inFile.readline().strip())
        rewireProb = float(inFile.readline().strip()) # don't write to file
        outFile.write("mat <- matrix(nrow = " + str(nrow)
                      + ", ncol = " + str(ncol) + ")\n")
        for i in xrange(1, nrow + 1):
            row = inFile.readline().strip()
            row = row.replace(" ", ", ")
            row = "c(" + row + ")"
            outFile.write("mat[" + str(i) + ",] <- " + row + "\n")
        inFile.close()
        outFile.close()
```

B.4 The script `network-metrics-lm.r`

```
# ---------------------------------------------------------------------------
# network-metrics-lm.r
# Copyright (C) 2008, 2009 -- Minh Van Nguyen <nguyenminh2@gmail.com>
#
# An R script to compute network metrics of Latora-Marchiori networks.
# That is, this script calculates the local and global efficiencies and
# network cost defined by Latora & Marchiori [1] as generalizations of
# the Watts-Strogatz [2] small world network metrics.
#
# REFERENCES:
# [1] V. Latora & M. Marchiori. Economic small-world behavior in weighted
#     networks. The European Physical Journal B, 32(2):249--263, 2003.
#
# [2] D. Watts & S.H. Strogatz. Collective dynamics of "small-world"
#     networks. Nature, 393(4):440--442, 1998.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
# ---------------------------------------------------------------------------


# Clear memory, removing (almost) everything in the working environment
# without any warning. Be careful with what you wish for.
rm(list = ls())
library("grDevices")  # for colours
library("sna")        # for social network analysis
# For various graph-theoretic operations, in particular, weighted shortest
# paths.
```

```r
library("igraph")


### Functions


# The average efficiency E of a graph G, where the graph has N vertices and
# K edges. This function can also be used to calculate the average
# efficiency even if G is a complete graph on N vertices. Such a complete
# graph is also denoted G^{ideal}, i.e. the ideal case where G has all the
# possible
#
# N(N - 1) / 2
#
# edges. Then the global and local efficiencies are defined in terms of E(G)
# and E(G^{ideal}). These notions of efficiency of a graph are defined in
# the paper (Latora & Marchiori 2003).
#
# INPUT:
#    adjMat -- the adjacency matrix of the underlying graph.
#    weightedGeoMat -- a matrix of weighted geodesics, or weighted
#       shortest paths.
#    level -- whether the average efficiency returned would be used in
#       computing the local or global efficiency. The parameter level can
#       take on either of two string arguments: "global" to indicate that
#       the returned average efficiency is to be used in calculating the
#       global efficiency of a network; and "local" which signifies that
#       the returned average efficiency is to be used in calculating the
#       local efficiency of a network. Default is "global".
#
# OUTPUT:
#    the average efficiency E of the graph G.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
averageEfficiency <- function(adjMat, weightedGeoMat) {
  # The number of vertices in weightedGeoMat. Thus adjMat and weightedGeoMat
  # are both N x N matrices.
  N <- dim(weightedGeoMat)[1]

  # for storing the average efficiency
  E <- NULL

  # check for the case that weightedGeoMat is a 1 x 1 matrix
  if (N == 1) {
    # harmonicSum <- 0
    E <- 0
  }
  else {
    # Compute the harmonic sum of lower triangular matrix of weightedGeoMat,
    # excluding the main diagonal.
    colLimit <- 0
    colStart <- 1
    rowStart <- 2
    harmonicSum <- 0
    for (row in rowStart:N) {
      colLimit <- colLimit + 1
      for (col in colStart:colLimit) {
        # Avoid the case where there's no path between vertices i and j. If
        # no path exists between i and j, then d_ij = +oo, which is positive
        # infinity. As d_ij -> +oo, then (1 / d_ij) -> 0. If adjMat
        # represents a totally isolated graph G, then the average efficiency
        # of G is 0.
        if (weightedGeoMat[row, col] != "Inf") {
          harmonicSum <- harmonicSum + (1 / weightedGeoMat[row, col])
        }
      }
    }

    # compute average efficiency
    E <- harmonicSum / (N * (N - 1))
  }

  E
}


# The global efficiency E_glob of a graph G, where the graph has N vertices
# and K edges. The notion of global efficiency of a graph is defined in the
# paper (Latora & Marchiori 2003). See also the function averageEfficiency,
# which defines the average efficiency of G. The measure E_glob is defined
# as
#
# E_glob = E(G) / E(G^{ideal})
#
# where G^{ideal} is the complete graph on N vertices. Thus E_glob is a
# ratio of the average efficiencies of two types of graphs: (1) the average
# efficiency of G itself; (2) the average efficiency of the complete graph
# on N vertices, which is the number of vertices in G.
#
# INPUT:
#    adjMat -- an N x N adjacency matrix of G.
#    weightMat -- an N x N weight matrix of G. See the function weightMatrix
#       for further details.
#
# OUTPUT:
#    the global efficiency E_glob of the graph G.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
globalEfficiency <- function(adjMat, weightMat) {
  # The number of vertices in the underlying graph G. Thus the column and
  # row dimensions must be equal.
  N <- dim(adjMat)[1]

  # matrix of weighted geodesics for G
  weightedGeoMat <- weightedGeodesics(adjMat, weightMat)
  # average efficiency of G
  aveEfficiency <- averageEfficiency(adjMat, weightedGeoMat)

  # Construct the adjacency matrix of a complete graph on N vertices. By
  # definition of complete graphs, a complete graph K_n and its geodesic
  # matrix G_dist are equivalent. That is, K_n and G_dist are copies of
  # each other. Also, G_dist has 1 everywhere, and 0 on the main diagonal.
  gIdealMat <- matrix(1, nrow = N, ncol = N)
  for (i in 1:N) {
    gIdealMat[i, i] <- 0
  }

  # matrix of weighted geodesics for G^{ideal}
  weightedGeoMat <- weightedGeodesics(gIdealMat, weightMat)
  # average efficiency of G^{ideal}
  aveEfficiencyGIdeal <- averageEfficiency(gIdealMat, weightedGeoMat)

  # compute the global efficiency
  Eglob <- aveEfficiency / aveEfficiencyGIdeal
}


# Let adjMat be an adjacency matrix of an undirected graph G. For a given
# vertex i of G, find the indices of the immediate neighbours of i.
#
# INPUT:
#    adjMat -- an adjacency matrix of an undirected graph G.
#    i -- a vertex of G.
#
# OUTPUT:
#    a vector containing vertices that are immediate neighbours of i.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
immediateNeighbours <- function(adjMat, i) {
  # The column dimension of adjMat. As adjMat is an adjacency matrix, it
  # doesn't matter if we get either of its row or column dimensions.
  colNum <- dim(adjMat)[2]

  # for storing indices of the immediate neighbours of vertex i
  neighIndex <- c()

  # find indices of the immediate neighbours of vertex i
  for (col in 1:colNum) {
    if (adjMat[i, col] == 1) {
      neighIndex <- c(neighIndex, col)
    }
  }

  neighIndex
}


# The adjacency matrix of G_i. If G is an undirected graph and i is a
# vertex of G, then G_i is the subgraph of neighbours of i, excluding i
# itself.
#
# INPUT:
#    aMat -- the adjacency matrix of the graph G.
#    i -- the index of the vertex whose neighbours we want to consider.
#       Let r and c be the row and column dimensions of aMat, respectively.
#       Then 1 < i < r or 1 < i < c.
#
# OUTPUT:
#    an adjacency matrix of G_i. If i is an isolated vertex, then return
#    an n x n zero matrix. If i is not isolated but all vertices in G_i are
#    isolated from each other, then return an n x n zero matrix. Else we know
#    that i is not isolated and there is a pair of vertices in G_i that
#    is connected by an edge; in this case, return an n x n matrix where
#    n > 0.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
neighboursAdjMat <- function(aMat, i) {
  # The column dimension of aMat. As aMat is an adjacency matrix, it
  # doesn't matter if we get either of its row or column dimensions.
  colNum <- dim(aMat)[2]

  # an adjacency matrix of the neighbours of vertex i
  neighAMat <- NULL

  # for storing indices of the immediate neighbours of vertex i
  neighIndex <- NULL

  # check if i is an isolated vertex
  if (sum(aMat[i, ]) == 0) {
    # If i is an isolated vertex, then return an n x n zero matrix, which
    # is of the same dimensions as those of aMat.
    neighAMat <- matrix(0, nrow = dim(aMat)[1], ncol = dim(aMat)[2])
  }
  # now we know that i is connected to at least another vertex
  else {
    # get indices of the immediate neighbours of i
    neighIndex <- immediateNeighbours(aMat, i)

    # The variables neighRowIndex and neighColIndex should be vectors of
    # equal length. Let neighRowIndex be of length n, then neighColIndex
    # also has length n. For k = 1,...,n, neighRowIndex[k] and
    # neighColIndex[k] refer to vertices that are immediate neighbours of
    # vertex i, and such that neighRowIndex[k] and neighColIndex[k] are
    # connected by an (undirected) edge.
    neighRowIndex <- NULL
    neighColIndex <- NULL
    for (row in 1:length(neighIndex)) {
      for (col in row:length(neighIndex)) {
        if (aMat[neighIndex[row], neighIndex[col]] == 1) {
          neighRowIndex <- c(neighRowIndex, neighIndex[row])
          neighColIndex <- c(neighColIndex, neighIndex[col])
        }
      }
    }

    # If i is not an isolated vertex, then the length of the vector
    # neighIndex is > 0. Let G_i be the subgraph of the neighbours of i. If
    # all vertices of G_i are isolated, then each of the vectors
    # neighRowIndex and neighColIndex has a length of zero. In this case,
    # neighAMat is a 2 x 0 matrix.
    neighAMat <- matrix(0, nrow = 2, ncol = length(neighRowIndex))
    neighAMat[1,] <- neighRowIndex
    neighAMat[2,] <- neighColIndex

    # check if G_i is totally isolated
    if (dim(neighAMat)[2] == 0) {
      # If G_i is totally isolated, then return an n x n zero matrix, which
      # is of the same dimensions as those of aMat.
      neighAMat <- matrix(0, nrow = dim(aMat)[1], ncol = dim(aMat)[2])
    }
    # now we know that at least one pair of vertices in G_i are connected
    else {
      neighAdjMat <- matrix(0, nrow = dim(aMat)[1], ncol = dim(aMat)[2])
      for (col in 1:dim(neighAMat)[2]) {
        neighAdjMat[neighAMat[1, col], neighAMat[2, col]] <- 1
        neighAdjMat[neighAMat[2, col], neighAMat[1, col]] <-  1
      }
      neighAMat <- neighAdjMat
    }
  }

  neighAMat
}


# The local efficiency E_loc of a graph G, where the graph has N vertices
# and K edges. The notion of local efficiency of a graph is defined in the
# paper (Latora & Marchiori 2003). See also the function averageEfficiency,
# which defines the average efficiency of G. The measure E_loc is defined
# as
#
# E_loc = (1/N) \sum_{i \in G} E(G_i) / E(G^{ideal}_i)
#
# where G_i is the subgraph of neighbours of vertex i, and G^{ideal}_i is
# the complete graph on N_i, which is the number of vertices in G_i. Note
# that G_i excludes the vertex i, and only considers the graph formed by
```

```r
# its immediate neighbours.
#
# INPUT:
#    aMat -- the adjacency matrix of the graph G. If G is an undirected
#       graph, then aMat is symmetric about the main diagonal.
#
# OUTPUT:
#    the local efficiency E_loc of the graph G.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
localEfficiency <- function(aMat, weightMat) {
  # The number of vertices in the underlying graph G. Thus the column and
  # row dimensions must be equal.
  N <- dim(aMat)[1]

  # summing the ratios (EGi / EIdealGi) for all vertices i
  cumSum <- 0           # the cumulative sum
  EGi <- 0              # average efficiency of G_i
  EIdealGi <- 0         # average efficiency of G^{ideal}_i
  idealGi <- 0          # adjacency matrix of G^{ideal}_i
  for (i in 1:N) {
    # adjacency matrix of G_i
    neighI <- neighboursAdjMat(aMat, i)
    # matrix of weighted geodesics for G_i
    weightedGeoMat <- weightedGeodesics(neighI, weightMat)
    # average efficiency of G_i
    EGi <- averageEfficiency(neighI, weightedGeoMat)

    # Construct the adjacency matrix of a complete graph on K_i vertices. By
    # definition of complete graphs, a complete graph K_n and its geodesic
    # matrix G_dist are equivalent, provided that K_n is unweighted.
    idealGi <- matrix(0, nrow = N, ncol = N)
    neighIndex <- immediateNeighbours(aMat, i)
    for (j in 1:length(neighIndex)) {
      for (k in j:length(neighIndex)) {
        idealGi[neighIndex[j], neighIndex[k]] <- 1
        idealGi[neighIndex[k], neighIndex[j]] <- 1
      }
      # do this since we want zeros along the main diagonal
      idealGi[neighIndex[j], neighIndex[j]] <- 0
    }

    # matrix of weighted geodesics for G^{ideal}_i
    weightedGeoMat <- weightedGeodesics(idealGi, weightMat)
    # average efficiency of G^{ideal}_i
    EIdealGi <- averageEfficiency(idealGi, weightedGeoMat)

    # Prevent division by zero, which is possible when EIdealGi = 0. If
    # both EGi and EIdealGi are zero, then we get (0 / 0), which returns
    # a NaN for "not a number". CAUTION: we need to consider four cases:
    #
    # EGi  EIdealGi
    # -------------
    # 0    0       <- (EGi / EIdealGi) = 0 because we say so
    # 0    y1      <- (EGi / EIdealGi) = 0
    # x1   0       <- Is it possible to get this case?
    # x2   y2      <- (EGi / EIdealGi) \in RR\{0}
    #
    # where x1, x2, y1, y2 \in RR are non-zero and RR is the set of
    # real numbers.
    if (EGi == 0) {
      cumSum <- cumSum + 0
    }
    else {
      cumSum <- cumSum + (EGi / EIdealGi)
    }
  }

  Eloc <- cumSum / N
}


# The cost of a network G with N vertices and K edges. For now, we assume
# that G is an undirected graph so that its adjacency matrix is symmetric
# about the main diagonal. The generalization of the Watts-Strogatz model
# contained in (Latora & Marchiori 2003) considers directed as well as
# undirected graphs.
#
# INPUT:
#    adjMat -- the adjacency matrix of G. This adjacency matrix must have
#       the same dimensions as the matrix of distances of G.
#    distMat -- the matrix of distances between pairs of vertices. This
#       distance matrix has the same dimensions as the adjacency matrix of
#       G.
#
# OUTPUT:
#    the cost of the network G.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
networkCost <- function(adjMat, distMat) {
  netCost <- 0
  numerator <- 0
  denominator <- 0
  N <- dim(adjMat)[1]

  for (row in 2:N) {
    for (col in 1:(row - 1)) {
      numerator <- numerator + (adjMat[row, col] * distMat[row, col])
      denominator <- denominator + (distMat[row, col])
    }
  }

  netCost <- numerator / denominator
}


# The weight matrix of a ring lattice G that has N vertices. This weight is
# defined in terms of the Euclidean distance between pairs of nodes. If i
# and j are vertices of G, then the distance between i and j is
#
# l_ij = [2 * sin(|i - j| pi / N)] / [2 * sin(pi / N)]
#      = sin(|i - j| pi / N) / sin(pi / N)
#
# The distance between each pair of neighbouring vertices is l_ij = 1 and
# the distance from i to itself is trivially l_ii = 0. The weight matrix
# of G is denoted {l_ij}, which has zero along the main diagonal and is
# symmetric about this diagonal.
#
# INPUT:
#    n -- an integer > 0; this is the number of vertices of the ring
#       lattice G
#
# OUTPUT:
#    the weight matrix {l_ij} of G. If n <= 0, then return NULL.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
```

```
weightMatrix <- function(n) {
  weightMat <- NULL

  if (n > 0) {
    # construct an n x n matrix with zero everywhere
    weightMat <- matrix(0, nrow = n, ncol = n)

    # Calculate the Euclidean distances on the ring lattice. Perhaps we
    # need only to consider either of the lower triangular or upper
    # triangular matrices, excluding the main diagonal.
    for (row in 1:n) {
      for (col in 1:n) {
        if (row != col) {
          # numerator <- 2 * sin((abs(row - col) * pi) / n)
          # denominator <- 2 * sin(pi / n)
          numerator <- sin((abs(row - col) * pi) / n)
          denominator <- sin(pi / n)
          weightMat[row, col] <- numerator / denominator
        }
      }
    }
  } else {
    weightMat <- NULL
  }

  weightMat
}


# A matrix of weighted shortest paths for a weighted ring lattice G. The
# lattice G has N vertices and a degree of k per vertex.
#
# INPUT:
#    adjMat -- the adjacency matrix of G. If G is undirected, then adjMat
#       is symmetric about the main diagonal. The adjacency matrix of G
#       must have the same dimensions as the weight matrix of G.
#    weightMat -- a matrix of edge weights. This is an N x N matrix,
#       where N is the number of vertices in G. If G is undirected, then
#       weightMat is symmetric about the main diagonal. The adjacency
#       matrix of G must have the same dimensions as the weight matrix of G.
#
# OUTPUT:
#    an N x N matrix of weighted shortest paths. If G is totally isolated,
#    then return an N x N matrix with +oo everywhere, and zero along the
#    main diagonal. Else G has an (undirected) edge connecting a pair of
#    its vertices, so we return an N x N matrix of weighted shortest paths.
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
weightedGeodesics <- function(adjMat, weightMat) {
  N <- dim(adjMat)[1]
  weightedGeo <- NULL

  # Check for totally isolated graphs. The graph G represented by adjMat is
  # totally isolated if all its vertices are isolated from each other. For
  # a totally isolated graph G of dimensions N x N, its corresponding
  # matrix of weighted geodesics is an N x N matrix with +oo (positive
  # infinity) everywhere, and zero along the main diagonal.
  if (sum(adjMat) == 0) {
    weightedGeo <- matrix(Inf, nrow = N, ncol = N)
    for (i in 1:N) {
      weightedGeo[i, i] <- 0
    }
  }
  # Now we know that G is not totally isolated, so at least one pair of
  # vertices in G is connected by an (undirected) edge. Then proceed to
  # find the matrix of weighted geodesics corresponding to G.
  else {
    colLimit <- 0
    colStart <- 1
    rowStart <- 2
    startVertex <- c()
    endVertex <- c()
    edgeWeight <- c()

    # As weightMat is symmetric about the main diagonal, we only need to
    # consider its lower (or upper) triangular matrix, excluding entries
    # along the main diagonal.
    for (row in rowStart:N) {
      colLimit <- colLimit + 1
      for (col in colStart:colLimit) {
        if (adjMat[row, col] == 1) {
          startVertex <- c(startVertex, row)
          endVertex <- c(endVertex, col)
          edgeWeight <- c(edgeWeight, weightMat[row, col])
        }
      }
    }

    e <- c()
    for (i in 1:length(startVertex)) {
      e <- c(e, startVertex[i], endVertex[i], edgeWeight[i])
    }
    emat <- matrix(nc = 3, byrow = TRUE, e)
    for (row in 1:dim(emat)[1]) {
      emat[row, 1] <- emat[row, 1] - 1
      emat[row, 2] <- emat[row, 2] - 1
    }
    g <- add.edges(graph.empty(N, directed = FALSE),
                   t(emat[, 1:2]), weight = emat[, 3])
    weightedGeo <- shortest.paths(g)
  }

  weightedGeo
}


# The main routine. This is where the network metrics are calculated. For
# the Watts-Strogatz model, the network metrics is comprised of the
# characteristic path length L and the clustering coefficient C. As regards
# the generalization of Watts-Strogatz contained in the paper
# (Latora & Marchiori 2003), the network metrics are the local efficiency
# E_loc, the global efficiency E_glob, and the network cost C_G.
#
# INPUT:
#    regmat -- a regular matrix.
#    probabilities -- a set of re-wiring probabilities.
#    weightMat -- a matrix of edge weights.
#    experiment -- n-th experiment
#
# OUTPUT:
#     Network metrics using the measures described in Latora & Marchiori [1].
#
# AUTHOR:
#    Minh Van Nguyen <nguyenminh2@gmail.com>
#
calculateNetworks <- function(regmat, probabilities, weightMat, experiment) {
  # set of adjacency matrices
  nets <- array(NA, dim = c(length(probabilities) + 1, n, n))
  # first matrix is regular matrix
  nets[1,,] <- regmat[,]

  # read in rewired networks for the specified experiment number
  print("read in rewired networks")
  counter <- 1
  for (p in 1:length(probabilities)) {
    # Read in rewired network into memory and the resulting object is named
    # "mat".
    source(paste("networks-r/graph-", experiment, "-", p, ".r", sep = ""))
    nets[counter + 1,,] <- mat[,]
    counter <- counter + 1
  }

  # This section is for the Latora-Marchiori generalization.
  # Global and local efficiencies, and network cost
  Eglob <- NULL
  Eloc <- NULL
  netCost <- NULL
  for (counter in 1:(length(probabilities) + 1)) {
    print(c("metrics for n-th rewiring probability ", counter))
    Eglob[counter] <- globalEfficiency(nets[counter,,], weightMat)
    Eloc[counter] <- localEfficiency(nets[counter,,], weightMat)
    netCost[counter] <- networkCost(nets[counter,,],
                                    weightedGeodesics(nets[counter,,],
                                                      weightMat))
  }

  # the structure to return
  result <- cbind(Eglob[1] / Eglob,
                  Eloc / Eloc[1],
                  netCost / netCost[1])
}


# Simulates a circular list. We are only interested in the index i of a
# member of this list, which has n members. One way to conceptualize this
# list is to visualize all n members as arranged in a cycle graph, in
# which each member i has an edge connecting it to i + 1, and an edge
# connecting it to i - 1. Another way to think about this function is to
# interpret it as a simple implementation of the group Z/Zn, where only
# the index of each i in Z/Zn is returned.
#
# INPUT:
#    index -- integer; index of an element in this circular list.
#    length -- integer > 0; the number of elements in this circular list.
#       FIXME: Maybe it's a good idea to implement the case where
#       length < 0, or provide some sanity checking to take care of that
#       possibility.
#
# OUTPUT:
#    If 0 < index <= length, then return index. If index > length, then
#    return index mod length. Else index < 0, so return index mod length.
#    If length is 0, then return NaN (not a number).
#
# AUTHOR:
#    Rodolfo Garcia-Flores
#    Documentation by Minh Van Nguyen <nguyenminh2@gmail.com>
#
returnIndex <- function(index, length) {
  if ((index > 0) && (index <= length)) {
    index
  }
  else if (index > length) {
    index - length * floor(index / length)
  }
  else {
    (length + index) + length * floor(-1 * index / length)
  }
}


### Main script


# Actual values should be n = 1000, k = 10. When run with these values,
# the script should take a few hours to complete. Test values can be
# n = 20, k = 4
n <- 1000             # vertices
k <- 6                # edges per vertex, MUST BE EVEN.
nTimes <- 20          # should be 20

# data for a logarithmically-scaled probability vector
nPoints <- 37
minProb <- 1e-4
maxProb <- 1

# output file name
summaryFileName <- "small-world-summary.txt"

# connectivity matrix of a regular network with no loops
regularMatrix <- matrix(0, nrow = n, ncol = n)
for (i in 1:n) {
  for (index in (k/2):1) {
    # Get right the indexes.
    jplus  <- returnIndex(i + index, n)
    jminus <- returnIndex(i - index, n)
    regularMatrix[i, jplus] <- regularMatrix[i, jminus] <- 1
  }
}

# logarithmically-scaled probability vector
factor <- (maxProb / minProb)^(1 / (nPoints - 1))
probs <- NULL
for (pt in 1:nPoints) {
  probs[pt] <- minProb * factor^(pt - 1)
}

# variable aliases
numMeasures <- 3   # how many measures
GE <- 1            # global efficiency
LE <- 2            # local efficiency
NC <- 3            # network cost

# weight matrix of the ring lattice
weightMat <- weightMatrix(n)

# first probability is zero, i.e. the regular matrix
results <- array(NA, dim = c(nTimes, length(probs) + 1, numMeasures))

for (experiment in 1:4) {
  print(c("network metrics for experiment ", experiment))
  results[experiment,,] <- calculateNetworks(regularMatrix,
                                             probs,
                                             weightMat,
                                             experiment)
}

# averages
GEmeans <- colSums(results[,,GE]) / nTimes
LEmeans <- colSums(results[,,LE]) / nTimes
NCmeans <- colSums(results[,,NC]) / nTimes

# write table of results
```

```
summary <- cbind(c(0, probs), GEmeans, LEmeans, NCmeans)
write.table(summary, file = summaryFileName, sep = "\t")

# plot variables
# insert plotting code here
```

References

[1] B. Bollobás. *Random Graphs.* Cambridge University Press, 2nd edition, 2001.

[2] R. D. Castro and J. W. Grossman. Famous trails to Paul Erdös. *Mathematical Intelligencer*, 21:51–63, 1999.

[3] P. Erdös and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959.

[4] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and functions using NetworkX. In G. Varoquaux, T. Vaught, and J. Millman, editors, *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA, USA, 2008. `http://networkx.lanl.gov`.

[5] M. D. Intriligator. *Mathematical Optimization and Economic Theory.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1971.

[6] T. Kaihara. Multi-agent based supply chain modelling with dynamic environment. *International Journal of Production Economics*, 85:263–269, 2003.

[7] V. Latora and M. Marchiori. Economic small-world behavior in weighted networks. *The European Physical Journal B*, 32(2):249–263, 2003.

[8] S. Milgram. The small world problem. *Psychology Today*, 2:60–67, 1967.

[9] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2009. ISBN 3-900051-07-0, `http://www.r-project.org`.

[10] W. A. Stein et al. *Sage Mathematics Software (Version 3.2.3).* The Sage Development Team, 2009. `http://www.sagemath.org`.

[11] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.