修士副テーマレポート

副テーマ研究題目： Pythonによるデータ分析基礎

氏　　名： KIM JAEHO

学生番号：1910080

副テーマ指導教員：谷澤 俊弘(Tanizawa Toshihiro)

令和元年10月

北陸先端科学技術大学院大学

目的

Graph-toolパッケージを理解して応用する。これを用いていろんなネットワークを視覚化し、分析ツールを利用する。そして、それらを本文で見せて、Graph-toolやネットワーク分析が初めな人が読んでも同じことができるようにマニュアルを作る。

はじめ

Graph-toolはイングランドのTiago P. Peixoto教授がC++をベースにしてpythonで作った複雑ネットワーク分析モジュールである。このモジュールではネットワーク生成、リワイヤリング、いろんなネットワークプロパティ計算、視覚化などができる。これらは他のネットワーク分析ツール(Ex. NetworkX、Pajek、etc)でもできるが、ここでGraph-toolが持つ長所は多い確率的ブロックモデルに対する関数、大規模での速い計算速度、強力なグラフ視覚化能力である。しかし、インストールための環境構築がかなり難しいのが最初の問題である。Dockerを使うとこれは比較的にたやすくなるが、Graph-toolを他pythonプログラム(Ex. TensorFlow)と連動する定型化した方法がないことが新たな問題になる。これはもともとこのモジュールがTiago教授が自身の研究のために作ったものからである。使い方が不便とは言ってもそれだけを解決すれば、Graph-toolは強力なネットワーク分析ツールになることには間違いない。ゆえに、この文書ではインストール方法とこのモジュールの基本的な使い方について述べる。

インストール

まずはDockerのインストールから始まる。Docker内には既にいろんなモジュール()があるのでインストールした後、Docker CMD窓でコマンドを入力すればそれでGraph-toolを設置することができる。構築環境はDebianである。

1. Docker インストール

```
# 正確にはDocker Engineをインストールする。そのためにまずDocker Repositoryをインストールする必要がある。その後、Repositoryからdockerをインストールする

# 最初にパッケージリストを更新する

$  sudo apt update

# HTTPSを通してRepositoryを使うためのパッケージをインストールする

$  sudo  apt  install  apt-transport-https  ca-certificates  curl  gnupg2  software-properties-common

# システムにRepositoryための公式GPG Keyを与える

$  curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
```

```
# AptにRepositoryを追加する
$  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com
  /linux/debian $(lsb_release -cs) stable"
# これができたらdockerをインストールすればいい
# パッケージリストを更新する
$  sudo apt update
# 最新バージョンをインストールする
$  sudo apt install docker-ce-cli containerd.io
# インストールが完了されたかを確認する
$  sudo systemctl status docker
```

2. Graph-tool インストール

```
# dockerが設置されていると、Graph-toolインストールはとても簡単である
$  docker pull tiagopeixoto/graph-tool
```

実行

IDEを使わない実行法は次である。

```
# 普通にDocker CMD窓で実行したいなら
$  docker run -it -u user -w /home/user tiagopeixoto/graph-tool ipython
# CMD窓でグラフも表したいならdisplay変数を指定する必要がある。ゆえに、実行するとき下のコマンドを入力する。上のコマンドではグラフなど描くのはできない。
$   docker run -ti -u user -w /home/user --rm -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp/.X11-unix tiagopeixoto/graph-tool ipython
# もし、Xサーバーに関するエラーがでると、
$  xhost +local:
```

次からはGraph-toolの各種関数について述べる。そのため、モジュールをインポートするコマンドは次であり、これ以降説明する関数は全部gtとプレフィクスする。

```
from graph_tool.all import *
import graph_tool.all as gt
```

なおかつ、本文ではnumpyモジュールを頻繁に使う。それはnpとプレフィックスした。

```
import numpy as np
```

ネットワーク生成

なにもないグラフオブジェクトを作って、ノードとリンクをいちいち追加する方法でネットワークを作られるが、Graph-toolにはいろんな生成関数が存在する。その中ではrandom_graphというコマンドをよく使われる。これを用いてある次数分布P(k)を持つネットワークを作ることができる。以下ではその例をいくつか挙げる。

1. Regular network

全てのノードが同じ次数を持つネットワーク

方向性なしのグラフで、Nは全体ノードの数、mはノード1つがもつリンク数である。

gt.random_graph(N、 lambda : m、 directed=False)

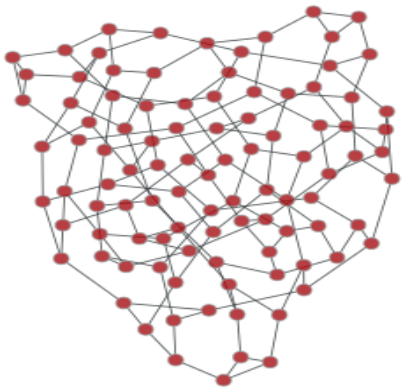


図 1 gt.random_graph(100, lambda: 3, directed=False)

2. Erdős–Rényi(ER) network

2.1 G(N, p) model

方向性なしのグラフで、Nは全体ノードの数、pは2つのノードがつながる確率

gt.random_graph(N、 lambda: np.random.poisson(λ)、 directed=False)

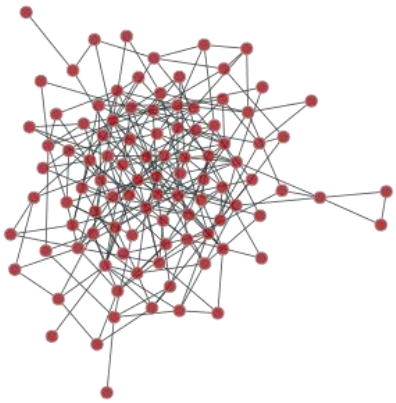


図 2 gt.random_graph(100、lambda: np.random.poisson(5)、 directed=False)

npはnumpyの名付けである。G(N、p)の次数分布は

$$\mathrm{P(k)} = \binom{N-1}{k} p^k (1-p)^{N-1-k} \cong \frac{(Np)^k e^{-Np}}{k!} = \frac{\lambda^k e^{-\lambda}}{k!} (Np = \lambda、\ N\ is\ large)$$

ゆえに、P(k)に平均値λのpoisson分布(Np＝λ)を使えば連結確率pのランダムネットワークを得られる。

2.2 G(N, E) model

方向性なしのグラフで、Nは全体ノードの数、Eは全体リンクの数

gt.random_graph(N、 lambda: m、 model="erdos"、 directed=False)

E＝N·m/2で、Model parameterについては後述する。もし、mが自然数ではなければ小数点以下を捨てて計算することになる。

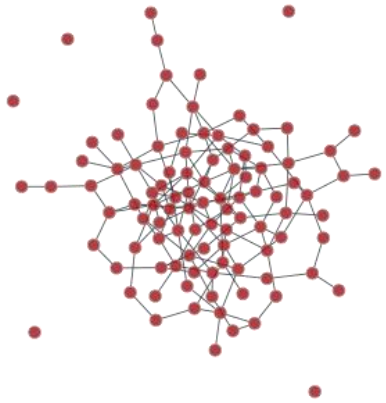


図 3 gt.random_graph(100、lambda: 3、model="erdos"、directed=False)

3. Scale-Free(SF) network

方向性なしのグラフで、Nは全体ノードの数、mは新しく追加されるノードが持っているリンクの数、cはconstant factor、gammaはexponential factor。

gt.price_network(N, m=a, c=None, gamma=b, directed=False, seed_graph=None)

資料のようにあらかじめP(k)を決めてrandom_graphを使ってもよいが、簡単に方向性なしのprice_networkを用いられる。このルーチンのリンクがつなげる確率πはπ＝$C^*(k^b + c)$の上、b＝1で無向の時、P(k)~$k^{-(3+\frac{c}{m})}$である。つまり、gamma=1と設定することでprice_networkでSF networkを描くことができる。もし、m>>1であると、seed_graphのサイズがm以上ではないと$V \geq m$になるまでにmが変化するのでそこに注意してseed_graphを準備する必要がある。それと、この関数はseed_graph=Noneであると、c>0ではノード1つから始まり、それ以外はノード2つがつながっている状態から始まるように作られている。

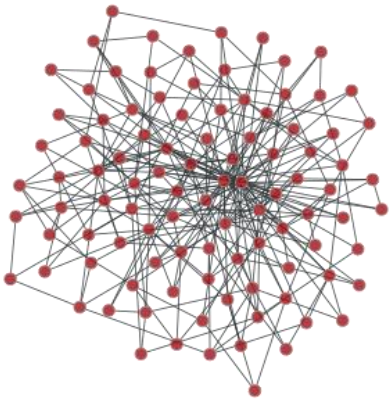


図 4 price_network(100, m=3, c=0, gamma=1, directed=False)

これ以外にlattice、complete_graph、circular_graphもある。Graph-toolはBlock modelの生成、推論、分析を支援するのでこれに関するコマンドやパラメーターは多いが今回は排除する。

これらの作り方はまずノードを配置してリンクをあらかじめ決めた分布に従って配分することの上、かなり決定論的である。ゆえに、P(k)にいろんな分布を持たせることができる。Numpyなどを使って定型化したものを用いてもよいし、それらを結合したMultimodal分布もできる。若しくはSF networkの変型であるBianconi-Barabasi modelみたいな適合度(Fitness)を適用したネットワークもP(k)を分かると表現できる。

例えば、P(k)として

$$\mathrm{P}(\mathrm{k};\mu,\sigma,\lambda)=\frac{1}{2}\cdot\frac{1}{\sqrt{2\pi\sigma^2}}e^{-\frac{(k-\mu)^2}{2\sigma^2}}+\frac{1}{2}\cdot\frac{\lambda^k e^{-\lambda}}{k!}$$

上の式のような正規とポアソン分布の和を代入することもできる。

```
def P(mu,sigma,gamma,p1,p2):
    k = p1*np.random.normal(mu,sigma)+p2*np.random.poisson(gamma)
    return k
g = gt.random_graph(100, lambda: P(4,1,10,0.5,0.5), directed=False)
```

また、Bianconi-Barabasi modelの次数分布P(k)は

$$\mathrm{P}(\mathrm{k})=\mathrm{C}\int\frac{\rho(\eta)}{\eta}\left(\frac{m}{k}\right)^{\frac{C}{\eta}+1}d\eta\quad(\eta:fitness)$$

上のように適合度を掛けたSFネットワークの次数分布の和である。もし、適合度が全部同じなら普通のSFネットワークになる。しかしこれは適合度分布が明確な必要があり、kminとkmaxを決めないと生成がかなり混乱である。ゆえに、下のように作るのがもっとよいと思う。コード中の適合度分布は自由に変えても成立する。

```
# All fitness are positive and int type
def BB_model(N, m, seed = None):
    g = gt.Graph(directed=False)
    fitness = []
    target = []
    if seed == None:
        seed = 0
        First_nodes = seed + m
        for i in range(First_nodes):
            g.add_vertex()
            fitness.append(np.random.randint(1,50))
            target.extend([i]*fitness[i])

    for i in range(N-First_nodes):
        v = g.add_vertex()
        fitness.append(np.random.randint(1,50))
        select_target = []
        for j in range(N-First_nodes):
            temp = target
            select_target.extend(random.sample(temp,1))
            try:
                temp.remove(select_target[j])
            except ValueError:
                pass
        for w in select_target:
            g.add_edge(v,w)
            target.extend([w]*fitness[w])
        target.extend([i+First_nodes]*fitness[i+First_nodes]*m)
    return g, fitness
```

Code 1 Pseudocode of BB network

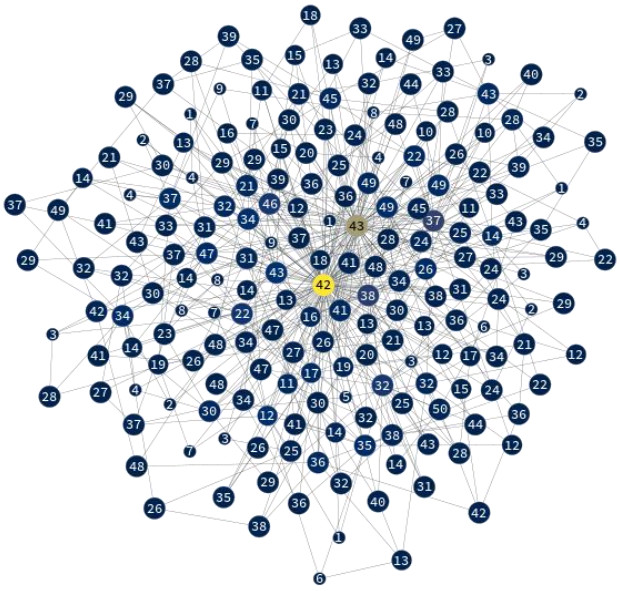


図 5 Bianconi-Barabasi model; vertex text means fitness

Graph-toolには2つのグラフを合わせるgraph_unionもある。しかしこれは、もし両グラフが同じ名前のinternal propertyを持っていると、 合わせる時に1つのpropertyが消える問題がある。例えば、g1グラフには”vcolor=red”、g2グラフには”vcolor=blue”が指定されていると合わせる時、vcolorがどちらの1つになる。

リワイヤリング

random_graphはrandom_rewireのパラメーターも使える。その中でmodelとedge_probsパラメーターを使ってrewiringされたグラフを作ることができる。先述べたようにGraph-toolはグラフをmodule化(block化)して分析できるのが長所なので、それに関するパラメーターもかなりあるがそれを除く。

modelには次のようなoptionがある。

	リンク変化	Degree sequence変化	Degree correlation変化
Erdos	Random	有り	有り
Configuration	Random	なし	有り
Constrained-configuration	Random	なし	なし
Probabilistic-configuration	Edge_probs	なし	有り

表 1 ‘model’ parameter of ‘random_rewire’ routine

Erdosは先G(N,E)modelで使ったもので、リンク繋がりを完全に無作為的に交ぜる。Constrained-configurationを用いるとsequenceとcorrelationを同じくしてリンクを変えることなので、ネットワークの異形同質体を求められる。Probabilistic-configurationを用いると決めたEdge_probsを基にしてrewiringできる。しかし、Edge_probsはdegree=kであるノードとdegree=k’であるノードがつながって

いる確率($k_{nn}(k)$)なので、主にassortativityを調整することの上、例えば、隣接行列・Laplacian行列ベースのrobustness measureやFVSなどを基にしてrewiringすることはこれではできない、全く別の課題である。

ネットワークプロパティ

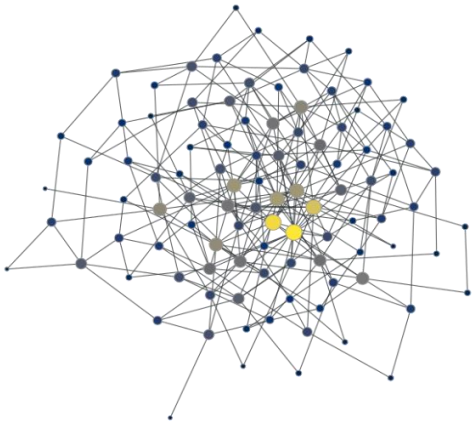
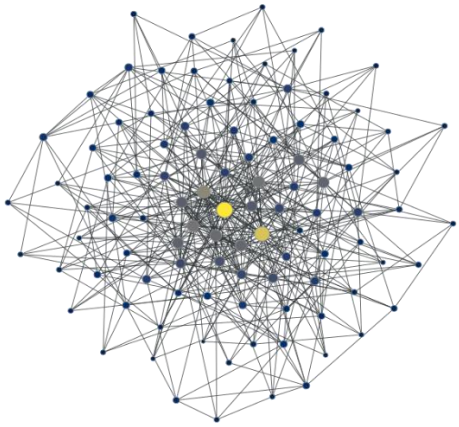
Graph-toolはネットワークのいろんなプロパティを計算し、それを視覚化することができる。これのルーチンやそのパラメーターは単純で得られる結果もわかりやすいのでここではそれらをテーブルでまとめて簡単に紹介する。

Centrality

Basic Routine	Return
gt.betweenness(g)	[array, array]=(vertex_betweenness, edge betweenness)
gt.closeness(g)	Vertex_closeness
gt.eigenvector(g)	(Number, array)=(Largest_eigenvalue, eigenvector)
gt.katz(g)	Katz_centrality
gt.hits(g)	(Largest_eigenvalue, authority_centrality, hub_centrality)
gt.pagerank(g)	Pagerank_centrality

表 2 Graph-tool's Network centrality routines

ルーチンはその中心性の名前と同じである。しかしそれらのリターンは個別に差がある。一般的に各ノードの中心性が。各ルーチンにはいろんなパラメーターがあるが、普通は分析したいグラフだけを入ればいい。下はランダムグラフとSFネットワークでの中心性を表したものである。

	gt.random_graph(100, lambda: np.random.poisson(5), directed=False)	gt.price_network(100, m=5, c=0, gamma=1, directed=False)
Betweenness centrality		

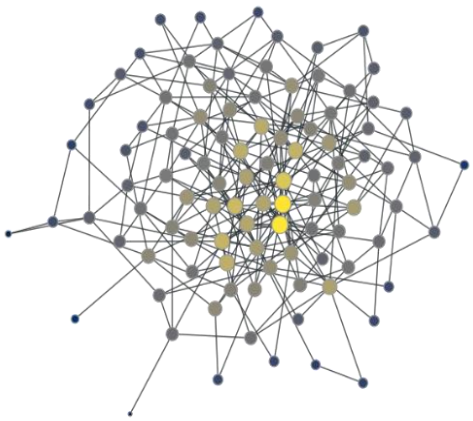
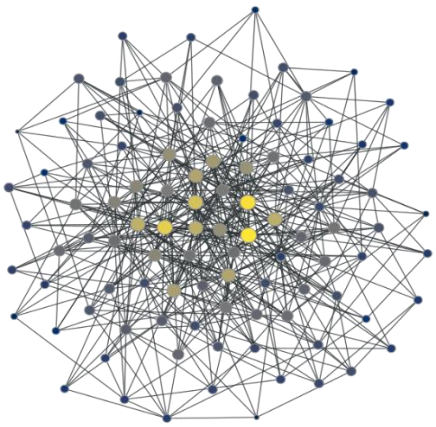
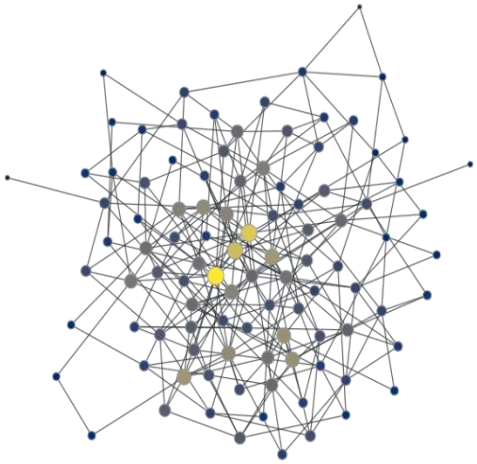
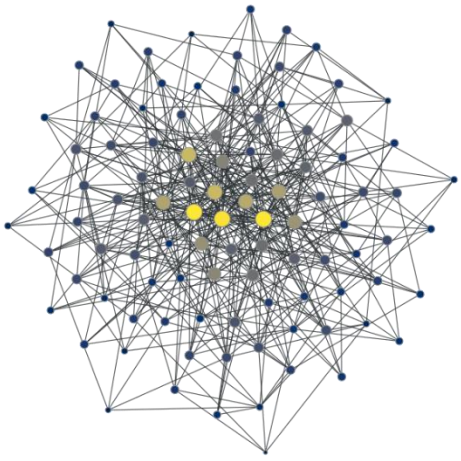
Closeness centrality		
Eigenvector centrality		

表 3 Examples of Table2; Yellow node has higher centrality

Spectral properties

Name	Basic Routine	Return
Adjacency matrix	gt.adjacency(g)	CSR matrix
Laplacian matrix	gt.laplacian(g)	CSR matrix
Non-backtracking matrix	gt.hashimoto(g)	CSR matrix

表 4 Graph-tool's matrix routines

Topological properties

What we can get	Basic Routine	Return
Shortest path from source to target	gt.shortest_path(g,source,target)	(vertex lists, edge lists)
Pseudo diameter of graph and Two end point vertex of diameter	gt.pseudo_diameter(g)	(Number, tuple)=(diameter, end points(pair))
Find subgraph in main graph	gt.subgraph_isomorphism(sub,g)	Vertex sets
Minimum spanning tree of graph	gt.min_spanning_tree(g)	Tree sets(Edge)

Route of Traveling salesman tour	gt.tsp.tour(g, source)	Vertex set
Visualized giant component	gt.extract_largest_component(g)	graph
k-core of each vertex	gt.kcore_decomposition(g)	Vertex property

表 5 Graph-tool's Network Topology routines

Coefficient & Etc

What we can get	Basic Routine	Return
Assortativity coefficient	gt.assortativity(g, deg = "in/out/total")	coefficient
Scalar assortativity coefficient	gt.scalar_assortativity(g, deg = "in/out/total")	coefficient
Clustering coefficient of whole graph	gt.global_clustering(g)	Vertex sets
Degree distribution of graph	gt.vertex_hist(g, vertex property)	[array, array]=(bin counts, bin edges)

表 6 Graph-tool's coefficient and other routines

表4~6はほとんどのネットワーク特性ルーチンと返還されるパラメーター形式である。これら以外にダイナミック特性やこのツールの強みであるネットワーク推論もあるが本文では扱わない。各ルーチンのハイパーパラメーターは全部方向性なし、重みなしと仮定したので、書かれたもの以外のハイパーパラメーターも多くある。

パーコレーション

Graph-toolではネットワークのロバストネスとレジリエンスを分析するためのノードやリンクアタックを再現することができる。これはパーコレーションと同値である。ここではGraph-toolで定義されたパーコレーションルーチンを用いて、表3で使ったネットワークでランダム攻撃とターゲット攻撃をする方法とその結果を見せる。

Name	Basic Routine	Return
Vertex percolation	gt.vertex_percolation(g, vertices)	array, property map = GCC size, vertex property map
Edge percolation	gt.edge_percolation(g, edges)	array, property map = GCC size, vertex property map

表 7 Graph-tool's percolation routines

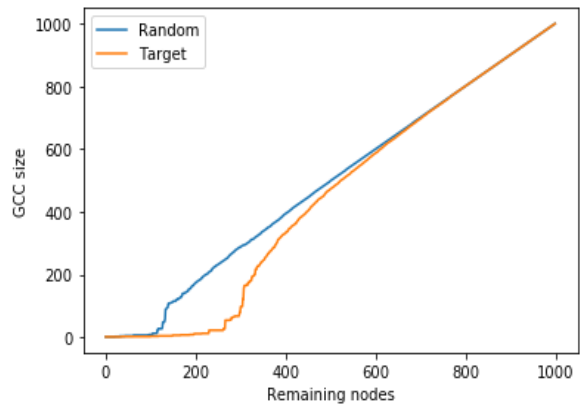
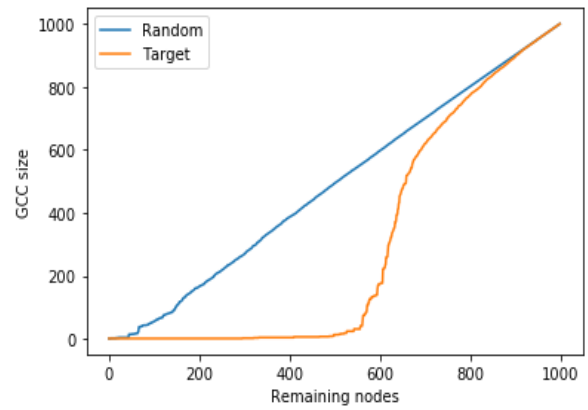
gt.random(1000, m=10, model='erdos', directed="False"	gt.prices_network(1000, m=5, c=0, gamma=1, directed="False"
	

表 8 GCC size after random attack and targeted attack; SF network is weaker to targeted attack than random network

```
L = list(g.vertices())
np.random.shuffle(L)
vertices_random = L
# percolationルーチンのパラメーター"vertices"はリストを逆順で読む。つまり、[1,2,3,4]なら後ろから4番ノード、3番ノード、。。。の順でノードを消す。
vertices_sorted = sorted([ v for v in g1.vertices()])
# ランダム攻撃の結果
sizes1, comp = gt.vertex_percolation(g, vertices_random)
# ターゲット攻撃の結果
sizes2, comp = gt.vertex_percolation(g, vertices_sorted)
```

Code 2 Pseudocode of percolation

Application

ここでは実際データをGraph-toolで扱って、ネットワークプロパティを見せる。使ったサンプルは4つで、(1)Friendship relation with Brightkite(location-based social network)(58228ノード、214078リンク)(Undirected, Unweighted)、(2)Interaction of Human-protein pairs(1706ノード、6207リンク)(Directed, Unweighted)、(3)Hyperlinks from Stanford University webpage(281903ノード、2312497リンク)(Directed, Unweighted)、(4)Co-purchase relation in Amazon(334863ノード、925872リンク)(Undirected, Unweighted)である。これらは全部KONECT(konect.uni-koblenz.de)で参照したデータである。分析結果は(1)Degree distribution、(2)Assortativity、(3)Global Clustering Coefficient、(4)Betweenness Centrality、(5)Closeness centrality、(6)Giant Connected Component Sizeである。

それと、他のプログラムとの性能を比べられるように各プロパティの計算時間も見せる。数値後ろの括弧がそれである。仕様はIntel Xeron, 2.80GHz, 4coresである。そして、(1)のネットワークのAssortativityを増加するようにリワイヤリングして、その前後のロバストネスを見せる。

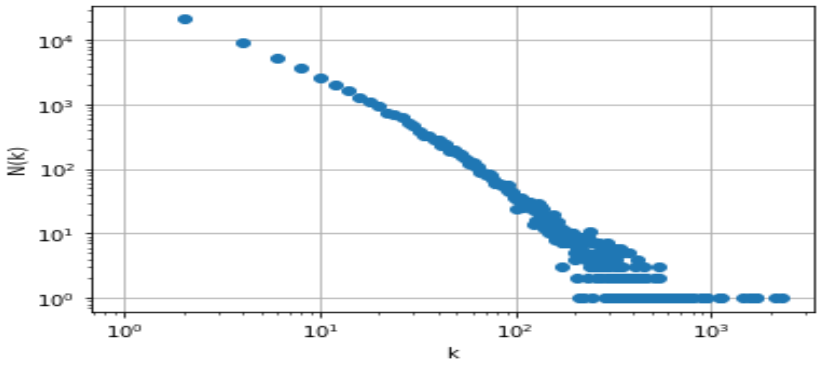
Degree distribution	
Assortativity	0.018(0.086s)
Clustering Coefficient	0.110(0.518s)
Maximum Betweenness Centrality	0.118(519.407s)
Maximum Closeness centrality	1.0(304.820s)
GCC Size	56739(0.026s)

表 9 Network properties of social network(58228ノード、214078リンク)

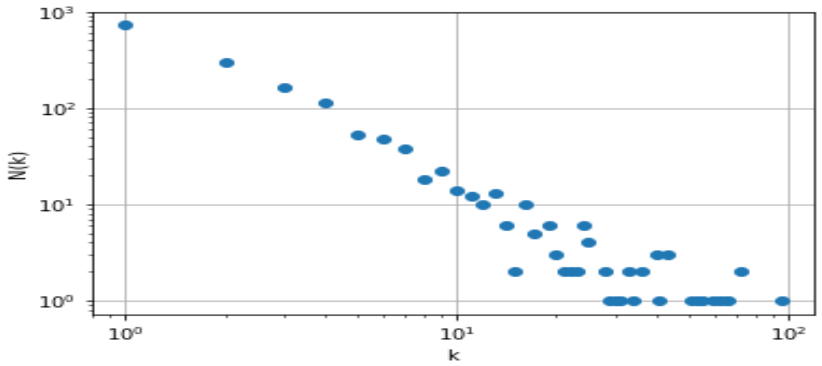
Degree distribution	
Assortativity	-0.188(0.037s)
Clustering Coefficient	0.006(0.133s)
Maximum Betweenness Centrality	0.068(0.111s)
Maximum Closeness centrality	0.343(0.251s)
GCC Size	1615(0.003s)

表 10 Network properties of Human-protein pairs(1706ノード、6207リンク)

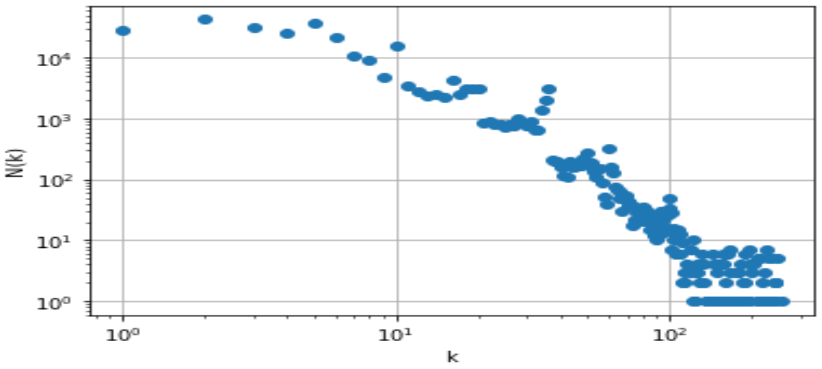
Degree distribution	
Assortativity	0.046(0.119s)
Clustering Coefficient	0.403(0.110s)
Maximum Betweenness Centrality	0.259(4233s)
Maximum Closeness centrality	0.118(3364s)
GCC Size	255265(0.106s)

表 11 Network properties of Hyperlinks(281903ノード、2312497リンク)

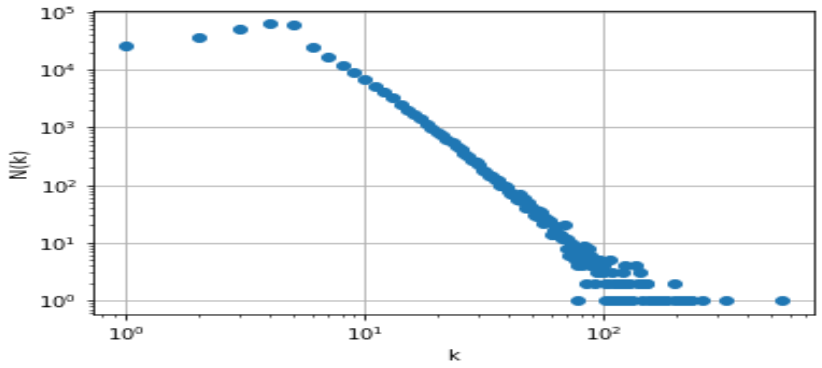
Degree distribution	
Assortativity	-0.059(0.126s)
Clustering Coefficient	0.205(0.248s)
Maximum Betweenness Centrality	0.032(17576s)
Maximum Closeness centrality	0.137(10449s)
GCC Size	334863(0.125s)

表 12 Network properties of Amazon purchase(334863ノード、925872リンク)

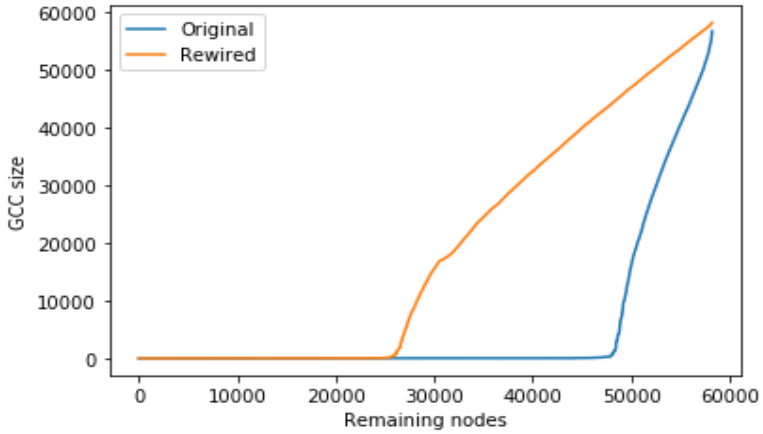


図 6 Robustness of rewired social network; Assortativity of rewired one is 0.126 (Original's one is 0.018). We can realize that rewired network has bigger robustness than Original

おわり

本文ではGraph-toolのインストールから始めてそれを用いてネットワークを分析する方法について述べた。その結果、Graph-toolはかなり多いネットワークための関数を整えており、それを適用する方法も簡単で、かなり高速であることを分かった。本文は一次的なマニュアルであり、次のチャンスがあれば確率的ブロックモデルなどを含めて本文では述べなかったものについて使い方と応用法に扱いたい。

Reference

[1] Pythonと複雑ネットワーク分析―関係性データからのアプローチ―（ネットワーク科学の道具箱II）第1章、共著：林 幸雄、谷澤 俊弘、鬼頭 朋見、岡本 洋、近代科学社、2019年11月出版予定

[2] Peixoto T. P. Graph-tool's documentation. (2019, Jul 13). Retrieved Oct.11, 2019 from https://graph-tool.skewed.de/static/doc/index.html