Combinatorica でネットワーク解析を学ぼう

前口上

この文書は吉野がネットワーク解析について学ぶために作成した私的なメモである．従って，このメモの誤りによって生じたいかなる不都合もその責任はメモに頼った人が責任を負うべきものであり，吉野は一切の責任を持たない．しかし，このメモを更に発展させてほしいという要望は作者のメモ作成の駆動力となるので，誤りの報告・追加の要望・暖かい激励の言葉は takashiあっとまーくrandom-walk.org に送信してみる価値はあるかもしれない（「あっとまーく」は半角の@で）．

このメモを書くきっかけになったのは Watts and Strongatz と Albert and Barabasi である（参考文献参照）．これらの論文の内容を自分なりに確認することと Combinatorica について学ぶことがこのメモの主題である．そのため，一般のグラフ理論の本よりもかなり偏った内容になっていると思われる．そのあたりは個人的な趣味の話なので勘弁してほしい（汗）それにずっと作りかけのままだし．次の目標はランダムグラフの相転移あたりかもしれない．

この文書の更新履歴は以下のとおり．

- 2003/12/10 Ver. 1.0
- 2004/02/20 Ver. 1.1 (パーコレーションを追記）
- 2004/08/21 Ver. 1.2 (ゲーム理論の部分を追記，Small world 部分を変更）
- 2004/10/06 Ver. 1.3 (複数の場所でミスを発見，small world 部分に正方格子の場合の話題を追加)
- 2005/01/10 Ver. 1.4 (リクエストにより Scale Free Network のプログラムを追加，説明文のミスを修正)
- 2005/01/24 Ver. 1.5 (パーコレーション部分を追加・修正）
- 2006/10/19 Ver. 1.6（リクエストにより Scale Free Network のグラフィックスに正多角形バージョンを追加．説明文のミスを修正．スタイルを修正）

久しぶりに眺めてみました．今となっては恥ずかしい幾つかのミスや表記があります．修正できるところは修正しましたが，今の自分ならこうは書かないなという部分も多いです．コメントをいただけると修正できてうれしいです．単純なことでもいいので何かありましたらご指摘願います．（2006/10/19 追記）

テクニック

基本的なこと

■ パッケ-ジのロード

はじめに Combinatorica パッケージをロードしなくてはならない.

```
Off[General::"spell1"]
```

```
<< DiscreteMath`Combinatorica`
```

とする．これは,毎回の作業の始まりに一度だけ行えばよい手続きである．頻繁に使うようになったら，*Mathematica* を起動したときに自動的に読み込むようにしておくとよいかもしれない．

■ Graph オブジェクト

Combinatorica でのグラフの内部表現は変えない方が，用意されているさまざまな関数を使いまわすことができるので好都合である．それがどのような要素で構成されているのかを確認するために，まず，5つの点からなる完全グラフ K_5を作ることにする．完全グラフというのが何かは後で考えることにしよう．

```
k5 = CompleteGraph[5]
```

```
-Graph:<10, 5, Undirected>-
```

このように,残念ながら作られたグラフがどのようなものなのかは,その表示を見るだけでは本当の概要（点と辺の数そして有向・無向）しかわからない. FullForm を用いてすべて表示すると，このオブジェクトはList の組み合わせからなることがわかる．

```
k5 // FullForm
```

```
Graph[List[List[List[1, 2]], List[List[1, 3]], List[List[1, 4]],
  List[List[1, 5]], List[List[2, 3]], List[List[2, 4]], List[List[2, 5]],
  List[List[3, 4]], List[List[3, 5]], List[List[4, 5]]],
 List[List[List[0.30901699437494745`, 0.9510565162951535`]],
  List[List[-0.8090169943749473`, 0.5877852522924732`]],
  List[List[-0.8090169943749476`, -0.587785252292473`]],
  List[List[0.30901699437494723`, -0.9510565162951536`]],
  List[List[1.`, 0]]]]
```

グラフを表現するために Combinatorica **が保持しているデータは連結に関するデ**-**タと位置に関するデータであることがわかる．ヘルプを参照すると，**

```
?Graph
```

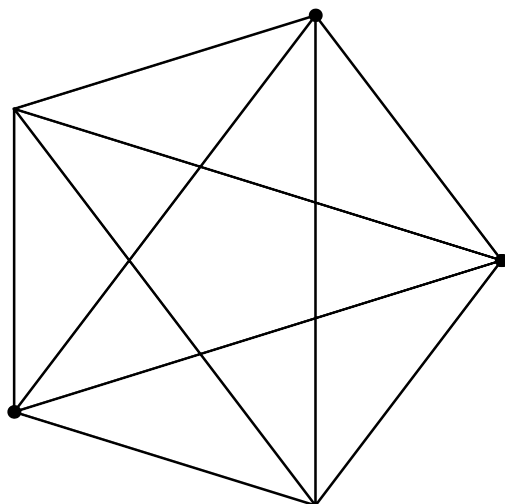
```
Graph[e, v, opts] represents a graph object where e is the list of
   edges annotated with graphics options, v is a list of vertices
   annotated with graphics options, and opts is a set of global
   graph options. e has the form {{{i1, j1}, opts1}, {{i2, j2},
   opts2},...}, where {i1, j1}, {i2, j2},... are edges of the
   graph and opts1, opts2,... are options that respectively apply
   to these edges. v has the form {{{x1, y1}, opts1}, {{x2, y2},
   opts2},...}, where {x1, y1}, {x2, y2},... respectively denote the
   coordinates in the plane of vertex 1, vertex 2,... and opts1,
   opts2,... are options that respectively apply to these vertices.
   Permitted edge options are EdgeWeight, EdgeColor, EdgeStyle,
   EdgeLabel, EdgeLabelColor, and EdgeLabelPosition. Permitted
   vertex options are VertexWeight, VertexColor, VertexStyle,
   VertexNumber, VertexNumberColor, VertexNumberPosition, VertexLabel,
   VertexLabelColor, and VertexLabelPosition. The third item in a
   Graph object is opts, a sequence of zero or more global options
   that apply to all vertices or all edges or to the graph as a
   whole. All of the edge options and vertex options can be used
   as global options also. If a global option and a local edge
   option or vertex option differ, then the local edge or vertex
   option is used for that particular edge or vertex. In addition to
   these options, the following two options can also be specified
   as part of the global options: LoopPosition and EdgeDirection.
   Furthermore, all the options of the Mathematica function Plot
   can be used as global options in a Graph object. These can be
   used to specify how the graph looks when it is drawn. Also, all
   options of the graphics primitive Arrow can also be specified
   as part of global graph options. These can be used to affect
   the look of arrows that represent directed edges. See the usage
   message of individual options to find out more about values
   these options can take on. Whether a graph is undirected or
   directed is given by the option EdgeDirection. This has default
   value False. For undirected graphs, the edges {i1, j1}, {i2,
   j2},... have to satisfy i1 <= j1, i2 <= j2,... and for directed
   graphs the edges {i1, j1}, {i2, j2},... are treated as ordered
   pairs, each specifying the direction of the edge as well. 詳細
```

と，辺と点座標のリスト＋オプションであることがわかる．

■ グラフの表示

上記のように，グラフオブジェクトを作成するだけでは，その構成を確認するのは難しい．グラフを表示するには，

```
ShowGraph[k5];
```



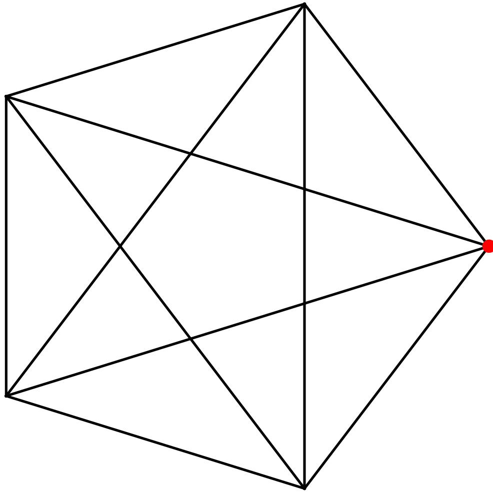
とする． ShowGraph には多くのオプションが用意されている．これを確認する．

```
Options[ShowGraph]
```

```
{VertexColor → GrayLevel[0], VertexStyle → Disk[Normal],
 VertexNumber → False, VertexNumberColor → GrayLevel[0],
 VertexNumberPosition → LowerLeft, VertexLabel → False,
 VertexLabelColor → GrayLevel[0], VertexLabelPosition → UpperRight,
 PlotRange → Normal, AspectRatio → 1, EdgeColor → GrayLevel[0],
 EdgeStyle → Normal, EdgeLabel → False,
 EdgeLabelColor → GrayLevel[0], EdgeLabelPosition → LowerLeft,
 LoopPosition → UpperRight, EdgeDirection → False}
```

例えば点の色を変えたい場合には,

```
ShowGraph[k5, VertexColor → RGBColor[1, 0, 0]];
```



さらに個々の点や線のスタイルのみを変えることもできるのだが,これは本題から外れるので Combinatorica のヘルプや関連サイトで情報収集をしてほしい.

■ 主な解析ツ-ル

点(vertex)の数を数えるには V を用いる．たとえば K_5 の頂点の数は,

```
vk5 = V[k5]
```

```
5
```

で求められる.同様にして，辺（edge）の個数を求めるには， M を用いる．

```
m = M[k5]
```

```
10
```

辺のリストは Edges で得ることができる．

```
Edges[k5]
```

```
{{1, 2}, {1, 3}, {1, 4}, {1, 5},
 {2, 3}, {2, 4}, {2, 5}, {3, 4}, {3, 5}, {4, 5}}
```

隣接の関係（点と点が繋がっているのかいないのか）は，隣接行列を用いて表現される．

```
ToAdjacencyMatrix[k5] // MatrixForm
```

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

数字は繋がっている辺の本数を表している．このグラフは向きを持っていないグラフ（無向グラフ）なので，転置行列も同じ形をしている．対角成分がすべてゼロであることは自分自身へのループが存在してないことを意味している.

頂点の個数は次数と呼ばれる．次数の平均を位数と呼ぶことがある．これは辺の数の２倍を点の数で割ればいいので，

```
2 M[k5] / V[k5]
```

```
4
```

と求められる．個々の点についての位数を求める場合には，隣接行列を用いる．隣接行列は List **の** List **であることを確かめておく．**

```
ToAdjacencyMatrix[k5]
```

```
{{0, 1, 1, 1, 1}, {1, 0, 1, 1, 1},
 {1, 1, 0, 1, 1}, {1, 1, 1, 0, 1}, {1, 1, 1, 1, 0}}
```

これを用いると，それぞれのリストは各点から出ている辺の個数に等しいので，各リストに Plus **を適用することによって，各点の位数が得られる．**

```
Apply[Plus, ToAdjacencyMatrix[k5]]
```

```
{4, 4, 4, 4, 4}
```

2**点間の経路は** ShortestPath **関数で求められる．例えば** k5 **において，点２と点５間の経路を求めたい場合には，**

```
ShortestPath[k5, 2, 5]
```

```
{2, 5}
```

その距離は経路のリストの長さから1**を引けばいいので**,

```
Length[ShortestPath[k5, 2, 5]] - 1
```

```
1
```

で求められる.

すべてのペアについて距離を求めたい場合には， AllPairsShortestPath を用いる（完全グラフの場合には，２点間の距離はすべて 1 である）．

```
AllPairsShortestPath[k5] // MatrixForm
```

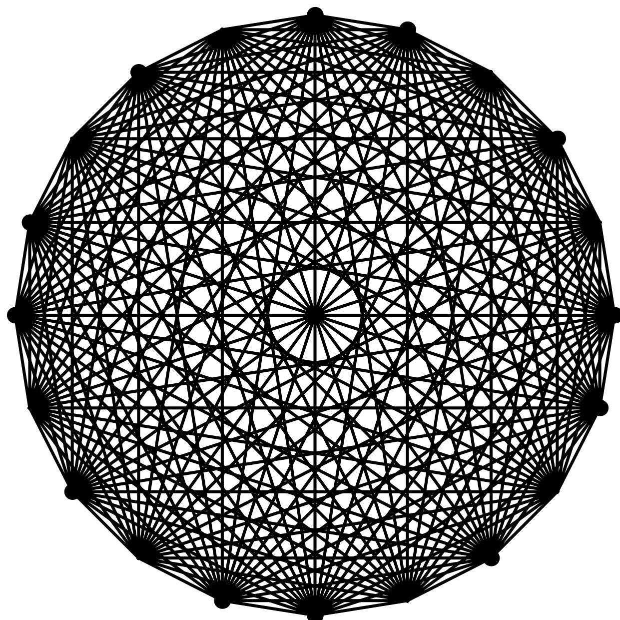
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

グラフの種類

■ 完全グラフ

完全グラフとは自分以外のすべての点と連結しているグラフを言う. n 個の点からなる完全グラフを K_n と表す．上で用いたのは 5 点からなる完全グラフだった. 同じようにして, 20 点からなる完全グラフ K_{20} は,

```
k20 = CompleteGraph[20]; ShowGraph[k20];
```

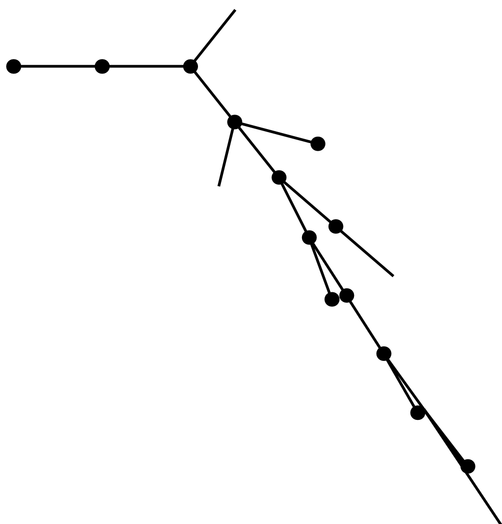


である.

■ 木と星

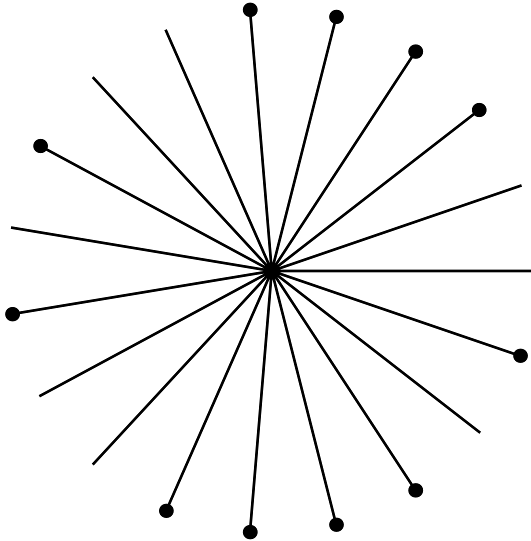
木（Tree）とは閉じたループも孤立した点もひとつも持たないグラフをいう.そのためには (n-1) 本の線で構成されていなければならない．ただし，その方法は一意的でないために， Combinatorica は Tree という関数を持たない．線の配置をランダムに指定しても良い場合には， RandomTree を用いる．たとえば，

```
rt20 = RandomTree[20]; ShowGraph[rt20];
```



という要領である.星(Star)**とは木の中で，ひとつの点が** (n-1) **の線を持ち，すなわち位数が**(n-1)**であり，他の点がひとつしか線を持たない，すなわち位数が** 1 **であるグラフをいう．これはグラフが一意的に決定されるので，** Combinatorica **で関数が用意されている．**

```
st20 = Star[20]; ShowGraph[st20];
```



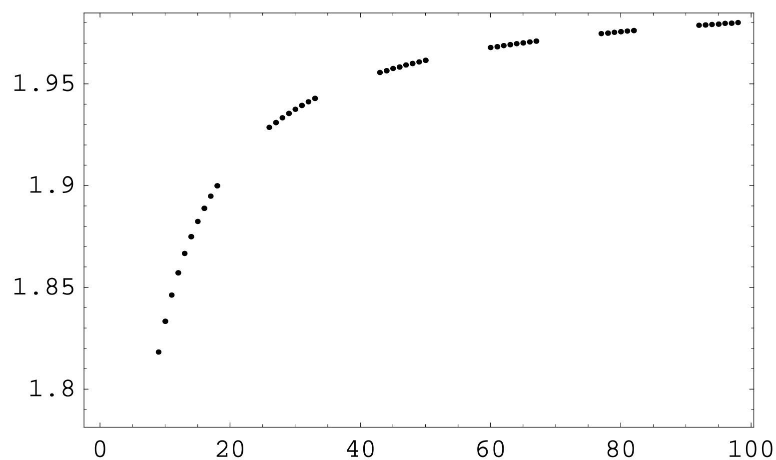
平均距離はほとんどが距離２にあるので，

```
(Plus @@ (AllPairsShortestPath[Star[20]] // Flatten) ) / (20 * 19)
```

$$\frac{19}{10}$$

となる．当然のことながら頂点の数が大きくなるほど２に近づいていき，

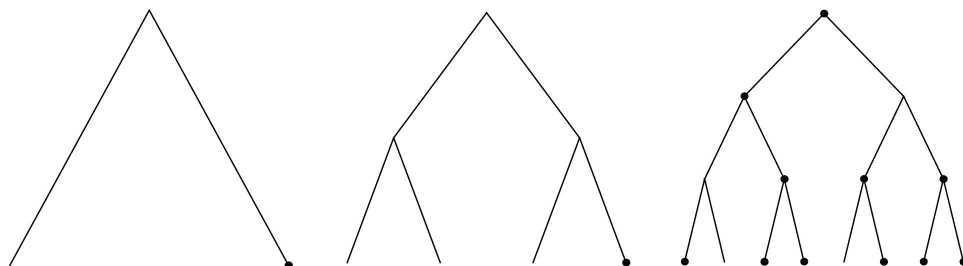
```
Table[(Plus @@ (AllPairsShortestPath[Star[i]] // Flatten) ) / (i * (i - 1)),
  {i, 3, 100}] // ListPlot[#, Frame → True] &
```



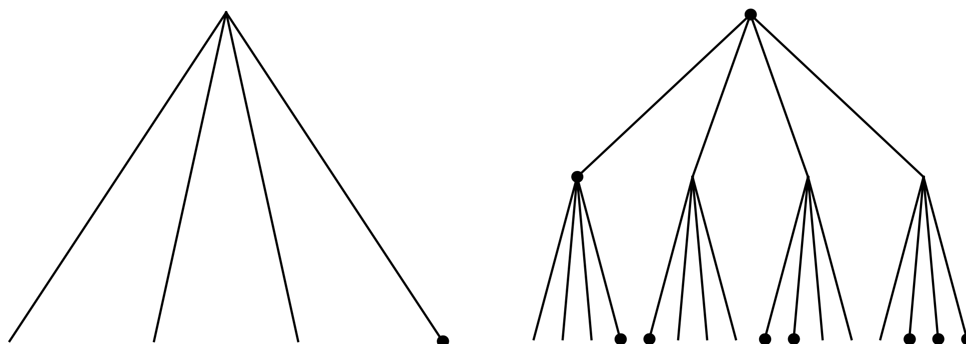

```
- Graphics -
```

規則的な木を作りたい場合には CompleteKaryTree を使うとよい.

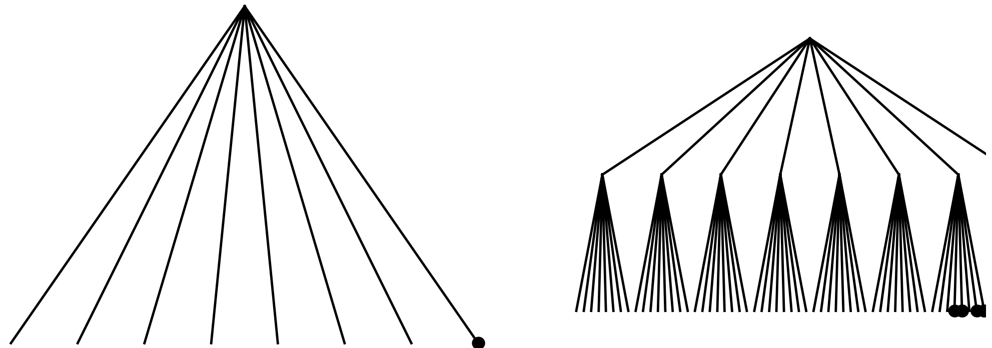
```
ShowGraphArray[Table[CompleteKaryTree[(2^i - 1) / 1, 2], {i, 2, 5}]];
```



```
ShowGraphArray[Table[CompleteKaryTree[(4^i - 1) / 3, 4], {i, 2, 4}]];
```



```
ShowGraphArray[Table[CompleteKaryTree[(8^i - 1) / 7, 8], {i, 2, 4}]];
```



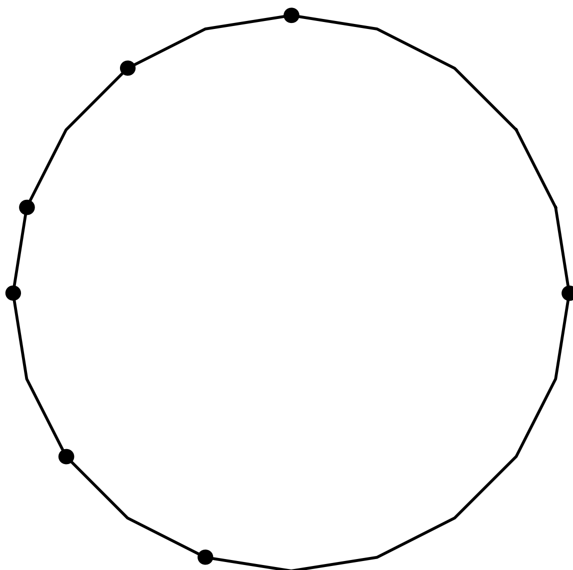
```
Table[(Plus @@ (AllPairsShortestPath[CompleteKaryTree[2^i - 1, 2]] // Flatten) ) /
   (i * (i - 1)), {i, 2, 8}] // N
```

```
{4., 16., 61.3333, 230.4, 857.6, 3181.71, 11803.4}
```

■ サイクル

サイクルとはひとつの閉じたループのみで構成されるグラフである.

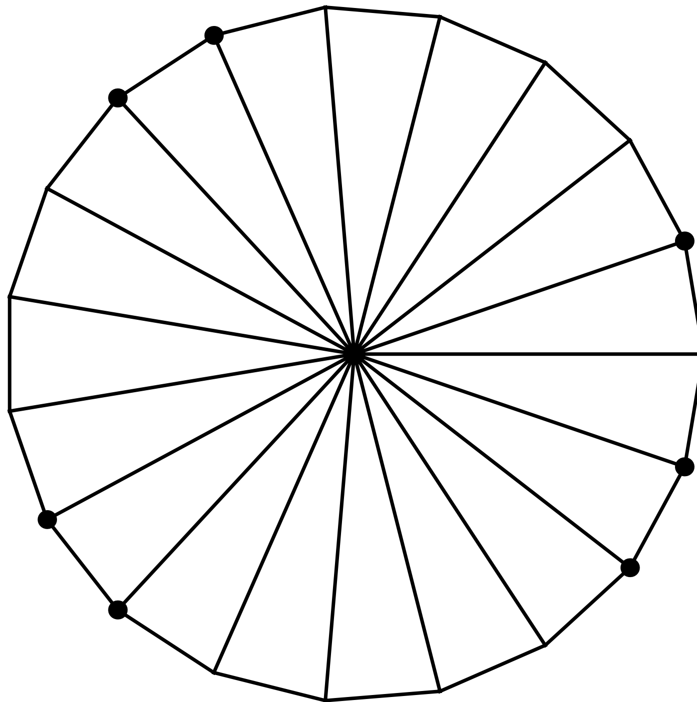
```
cy20 = Cycle[20]; ShowGraph[cy20];
```



■ Wheel

星とサイクルをあわせて車輪（Wheel）が作られる．

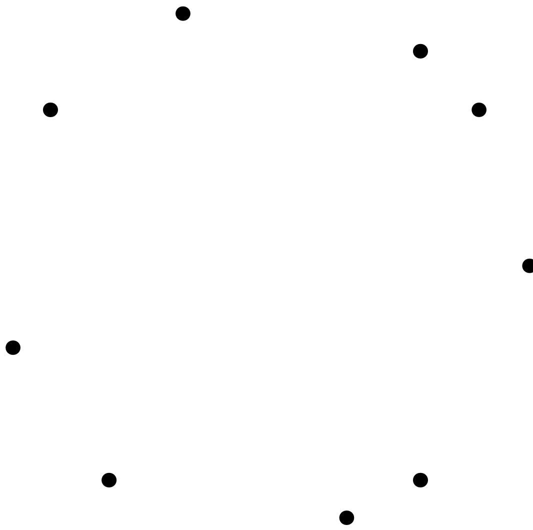
```
whl20 = Wheel[20]; ShowGraph[whl20];
```



■ Empty Graph

Empty graph **とはひとつも線を持たないグラフを言う**.

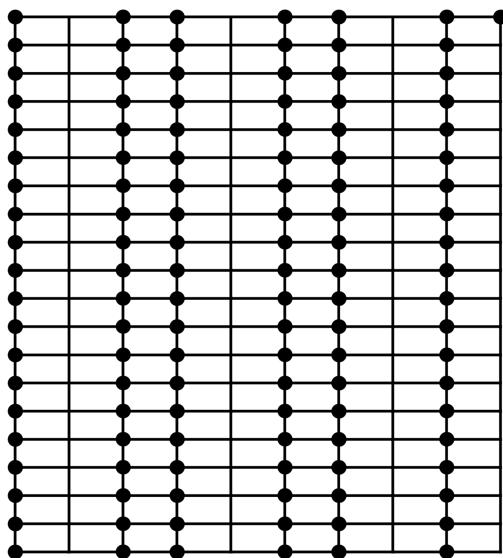
```
em20 = EmptyGraph[20]; ShowGraph[em20];
```



■ グリッドグラフ

視覚的に言えば格子状に配置したグラフである.でも,この言い方はトポロジカルな関係だけを問題にするグラフ理論では適切な言い方ではないかもしれない．でも，直感的には一番納得しやすいのではないかと思う．２次元や３次元の正方格子に相当するグリッドグラフはあるみたいだが，多次元の正方格子に相当するグリッドグラフはないのだろうか．作ろうと思えば作れるのだが…あるとパーコレーション理論との関係を検証できて便利なんだけど…

```
ShowGraph[GridGraph[10, 20]]
```

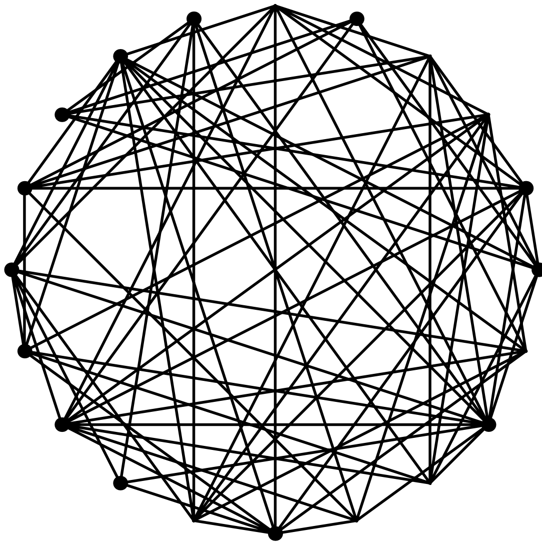


```
- Graphics -
```

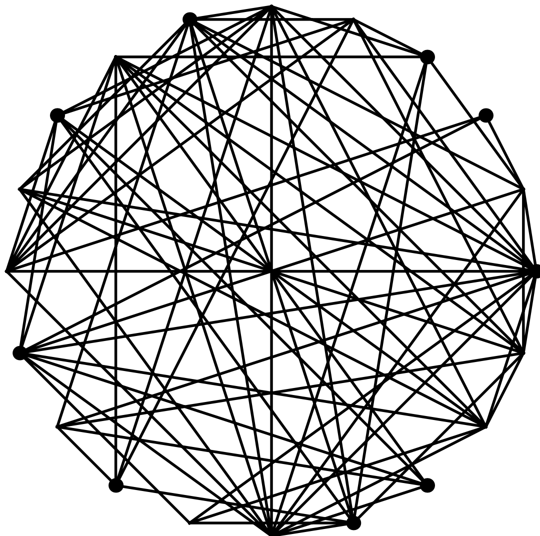
■ ランダムグラフ

ランダムグラフは,このノ-トを書く動機になったグラフである.これはn個の点からなり,そこに含まれる任意の2点をつなぐ線の有無を確率 p で決定することによって作られる.このグラフの持つ性質については次の章でまとめるつもりである．Combinatorica **では** RandomGraph **と**ExactRandomGraph **というふたつの関数が準備されている．前者は確率を後者は陵の本数を与えてグラフを作成する．すなわち，**RandomGraph[n, p] **は** n **個の節点によって構成される完全グラフをのすべての陵を確率** p **で生き残らせるか消去するかを決定することによって与える．それに対して** ExactRandomGraph[n,e] **は**e**本の生き残る辺をランダムに選択する．**RandomGraph **にはさらに向きの有無なども指定できる．**

```
rg2004 = RandomGraph[20, 0.4]; ShowGraph[rg2004];
```



```
(erg2076 = ExactRandomGraph[20, 76]) // ShowGraph;
```



RandomGraph は与えられた確率に従ってグラフを作るだけなので，作られた稜の数が一定ではない．

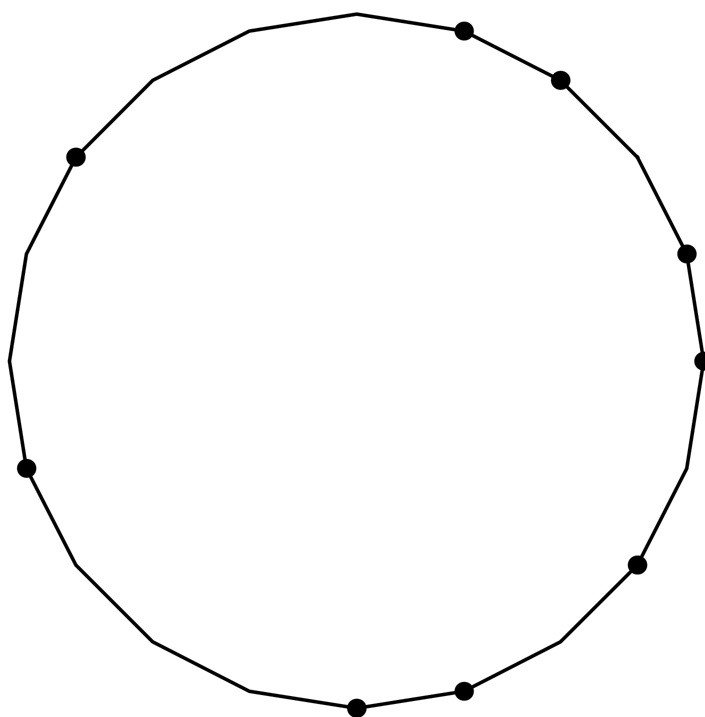
```
Map[M, {rg2004, erg2076}]
```

```
{76, 76}
```

■ Circulant グラフ

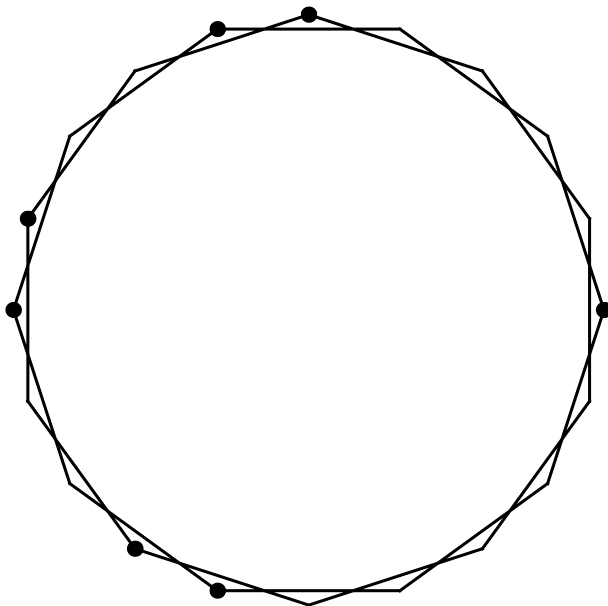
Circulant グラフは周期的に連結するグラフを作る．Circulant[n,1]は点の数が n で隣と連結するグラフ，すなわち Cycle グラフである．

```
CirculantGraph[20, 1] // ShowGraph;
```



Circulant[n,k]は k だけ飛んだ先にある点と周期的に連結するグラフを作る．

```
CirculantGraph[20, 2] // ShowGraph
```

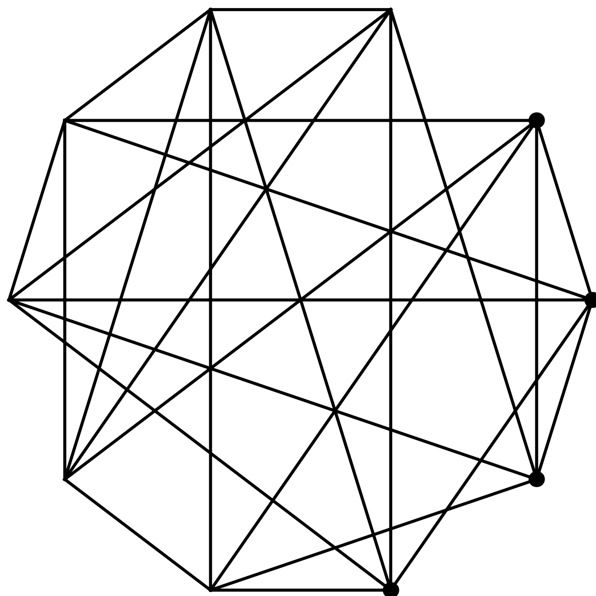


```
- Graphics -
```

■ 正則グラフ

すべての節点の位数が等しいグラフを正則グラフという．位数が k **である正則グラフを**k**次の正則グラフという．** Combinatorica **には対称性の少ない連結，すなわち連結の仕方がランダム（ヘルプの言葉を借りれば「準ランダム」）な正則グラフを作る関数** RegularGraph **が用意されている．**

```
ShowGraph[RegularGraph[5, 10]]
```

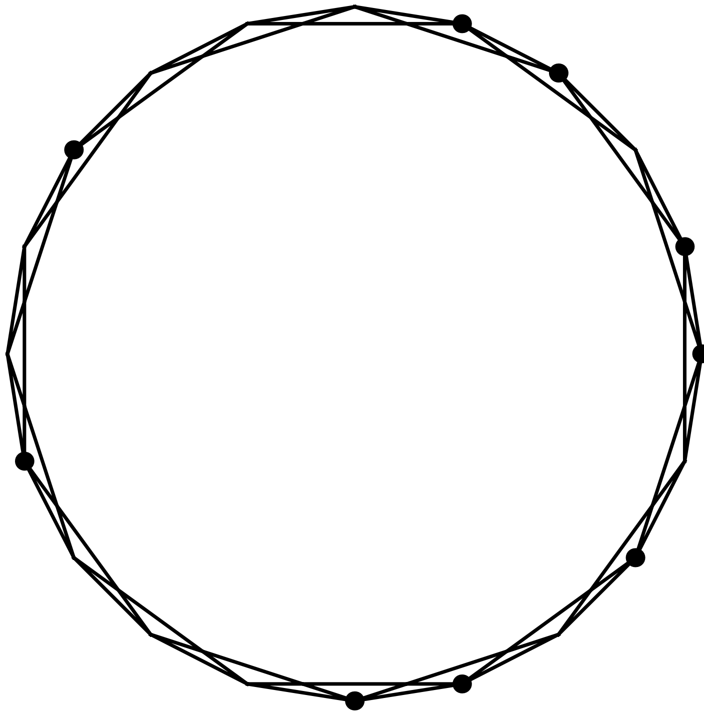


```
- Graphics -
```

これは呼び出されるたびに異なる正則グラフを作る．最も対称性のよい正則グラフを作るには，CirculantGraphを重ね合わせればよいので，以下の関数を使う．

```
makeSymmetricalRegularGraph[n_, k_] :=
 Apply[GraphSum, Map[CirculantGraph[n, #] &, Range[1, k / 2]]]
```

```
makeSymmetricalRegularGraph[20, 4] // ShowGraph
```



```
- Graphics -
```

```
?LineGraph
```

LineGraph[g] constructs the line graph of graph g. 詳細

自分でグラフを作るには

既成のグラフだけで足りることばかりではないと思われる．そこで，点や線を付けたり消したりする方法を知っておくと便利である.

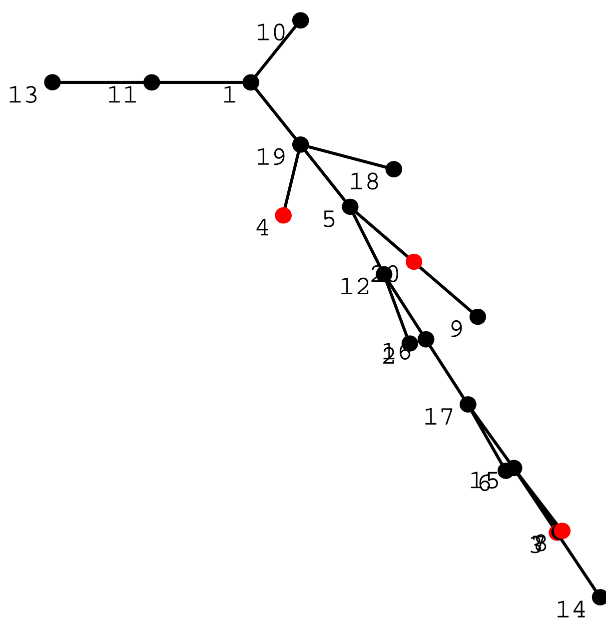
グラフ g に新しい辺を加えるには AddEdges[g, edgeList] を用いる．ここで edgeList は連結するふたつの点のリストである．例えば，前述のランダムツリーに線をひとつ加えてみよう．初めに，その構成を確認しておこう．隣接行列を使って隣接の様子を再確認する．

```
ToAdjacencyMatrix[rt20] // MatrixForm
```

$$
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

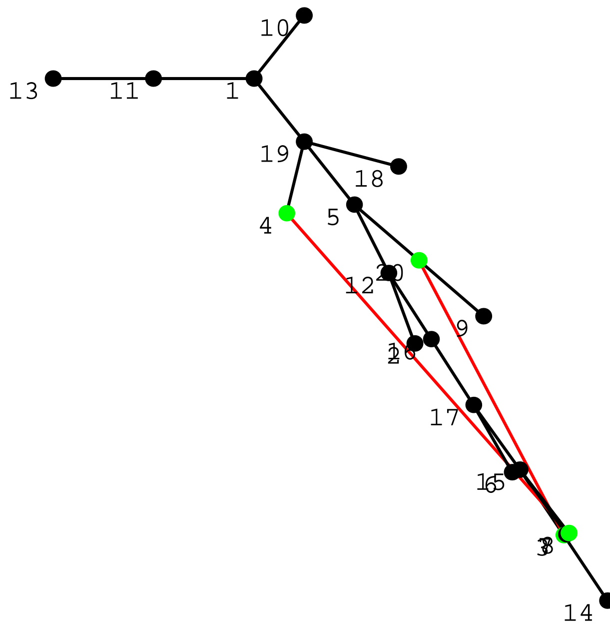
この行列の要素のうち値がゼロであるものが隣接していないことを意味している．例えば節点 4と接点8，接点3と接点20がそれである．

```
ShowGraph[
  SetGraphOptions[rt20, {{4, 8, 3, 20 , VertexColor → Red}}], VertexNumber → On];
```



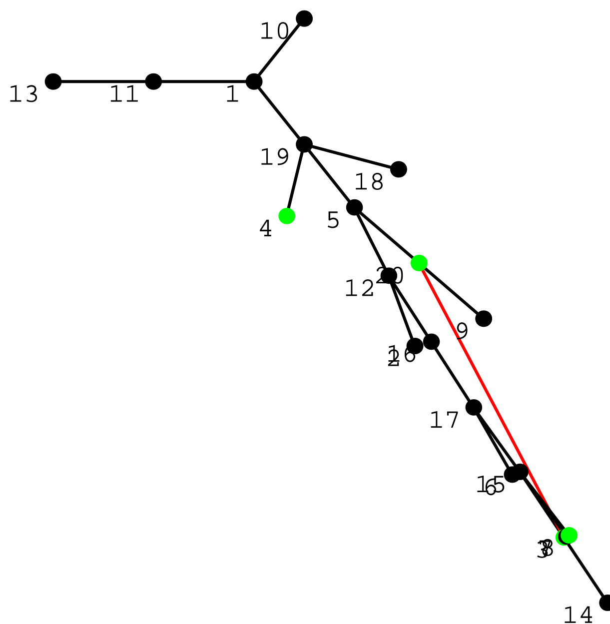
が隣接させる前のグラフである．AddEdge を用いて，２点を隣接させてみる．

```
nrt20 = AddEdges[rt20, {{4, 8}, {3, 20}}];
ShowGraph[SetGraphOptions[nrt20, {{4, 8 , 3, 20, VertexColor → Green},
   {{4, 8}, {3, 20}, EdgeColor → Red}}], VertexNumber → On];
```



逆に削除することもできる.

```
(nnrt20 = DeleteEdges[nrt20, {{4, 8}}]);
ShowGraph[SetGraphOptions[nnrt20, {{4, 8 , 3, 20, VertexColor → Green},
    {{4, 8}, {3, 20}, EdgeColor → Red}}], VertexNumber → On];
```



同様にして点の追加も行える.

```
AddVertex[nnrt20] //
  ShowGraph[SetGraphOptions[#, {{4, 8, 3, 20 , VertexColor → Green},
      {{4, 8}, {3, 20}, EdgeColor → Red}}], VertexNumber → On] &;
```

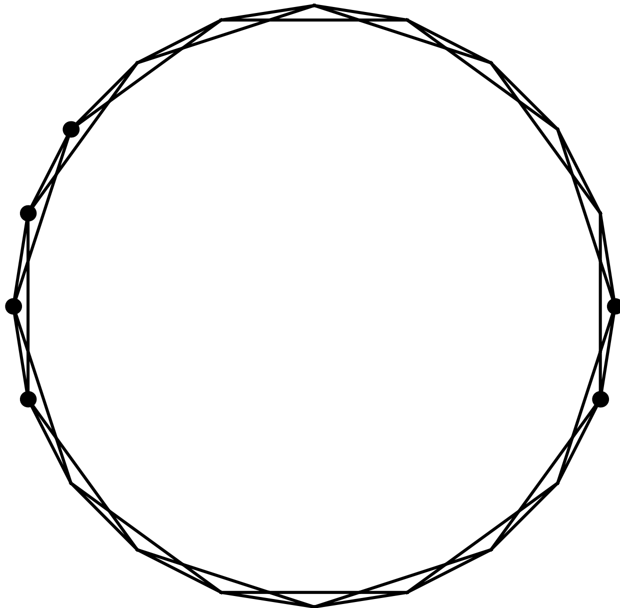
点を削除することも可能である.

```
DeleteVertex[nnrt20, 20] //
  ShowGraph[SetGraphOptions[#, {{4, 8 , 3, 20, VertexColor → Green},
      {{4, 8}, {3, 20}, EdgeColor → Red}}], VertexNumber → On] &;
```

大まかな形を作っておいて,そこから副グラフを作ることで新たなグラフを作ることもできる.

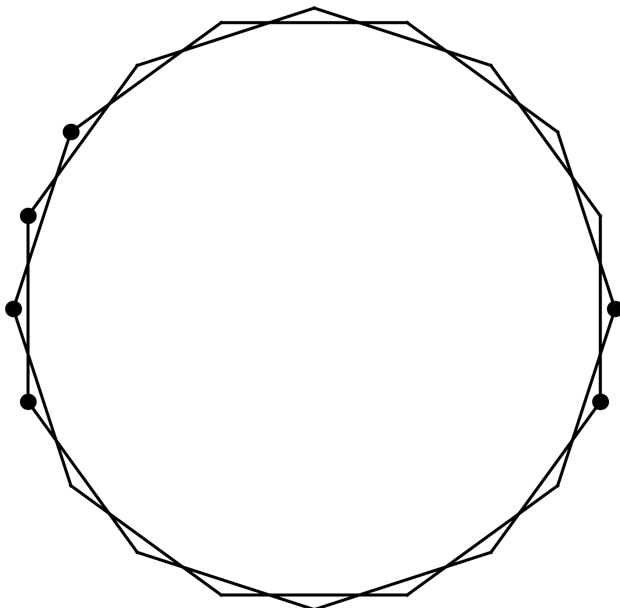
ふたつ以上ののグラフの和集合を作る．

```
(reg20 = GraphSum[CirculantGraph[20, 2], Cycle[20]]) // ShowGraph;
```



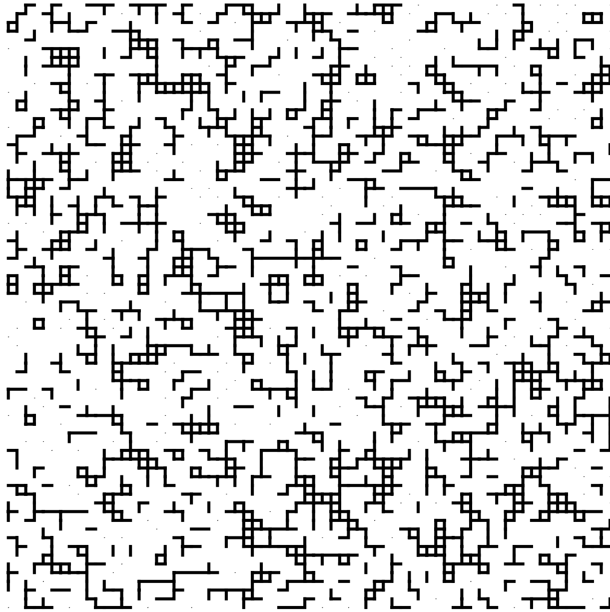
あるグラフから一部（副グラフだ）を取り除く．

```
GraphDifference[reg20, Cycle[20]] // ShowGraph;
```

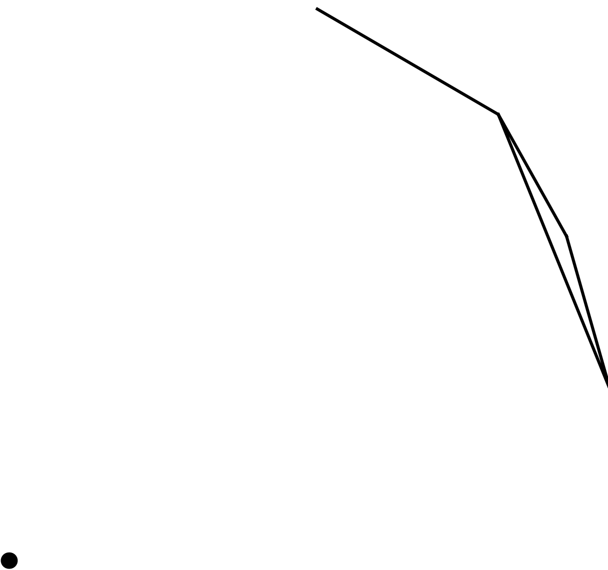


副グラフを作る．副グラフを作ることは生き残りを選ぶことといえるかもしれない．もとのグラフと生き残る点のリストを与えて，その他の点とその点に繋がっていた辺を除くことができる．

```
s = InduceSubgraph[GridGraph[70, 70], RandomSubset[Range[4900]]];
ShowGraph[s, VertexStyle -> Disk[0]];
```



```
InduceSubgraph[reg20, {1, 2, 3, 5, 10}] // ShowGraph;
```



解析の方法

グラフ（またはネットワーク）を特徴付ける量には上に述べたような点や辺の数や位数などの他にもさまざまな量がある．その中でもネットワークの構造解析を行うために使われることが多い，直径（diameter)**や平均最短経路，そしてクラスター係数（**clustering coefficient**）の** *Mathematica* **による計算方法について説明する．**

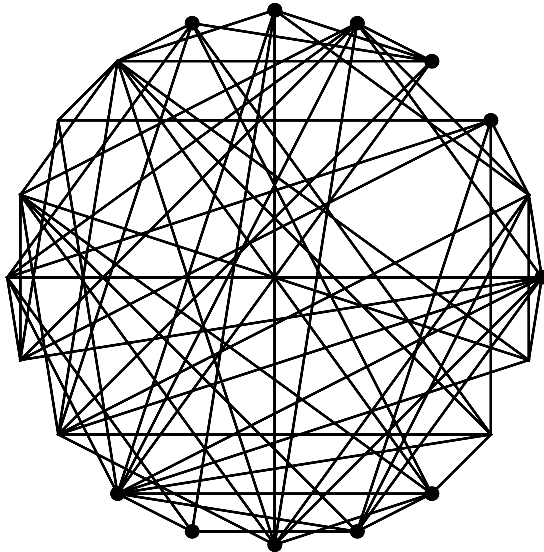
■ 直径

以下の文は Diameter **が実装されていたことをを知らないときにかいたもの．**

直径は既存の辺を通って２点間を連結する最小値（最短経路）の最大値である．最短経路は， AllPairs-ShortestPath**を用いて計算されるので，その最大値を求めればよい．**

ネットワークの構造解析という観点から言うと，直径が小さいことは遠くに行くまでにかかる手数が少なくて済むことを示しているので，小さいほどそのネットワークが密であると言うことができる．

```
rg2004 = RandomGraph[20, 0.4]; ShowGraph[rg2004];
```



```
AllPairsShortestPath[rg2004] // MatrixForm
```

```
0 1 2 1 1 2 2 2 2 2 2 1 2 2 1 1 2 2 1 1
1 0 3 2 2 2 2 1 2 1 1 2 2 2 2 1 2 1 2 2
2 3 0 1 1 1 1 2 2 2 2 1 1 2 2 2 2 2 2 2
1 2 1 0 2 2 2 2 1 1 1 2 1 2 1 2 2 2 2 1
1 2 1 2 0 2 1 2 2 2 3 2 1 1 1 2 2 2 2 2
2 2 1 2 2 0 1 2 2 1 2 1 2 2 2 2 1 1 2 2
2 2 1 2 1 1 0 1 1 2 2 1 2 2 1 2 1 1 1 2
2 1 2 2 2 2 1 0 2 2 1 2 1 2 2 2 2 2 2 2
2 2 2 1 2 2 1 2 0 2 1 1 2 2 1 1 1 2 1 2
2 1 2 1 2 1 2 2 2 0 1 2 1 1 2 2 2 2 2 1
2 1 2 1 3 2 2 1 1 1 0 2 2 2 2 2 2 2 2 1
1 2 1 2 2 1 1 2 1 2 2 0 2 3 1 2 2 1 2 1
2 2 1 1 1 2 2 1 2 1 2 2 0 1 2 1 1 1 1 1
2 2 2 2 1 2 2 2 2 1 2 3 1 0 2 1 2 2 2 2
1 2 2 1 1 2 1 2 1 2 2 1 2 2 0 2 1 2 2 1
1 1 2 2 2 2 2 2 1 2 2 2 1 1 2 0 1 2 2 1
2 2 2 2 2 1 1 2 1 2 2 2 1 2 1 1 0 1 2 2
2 1 2 2 2 1 1 2 2 2 2 1 1 2 2 2 1 0 2 2
1 2 2 2 2 2 1 2 1 2 2 2 1 2 2 2 2 2 0 1
1 2 2 1 2 2 2 2 2 1 1 1 1 2 1 1 2 2 1 0
```

```
AllPairsShortestPath[rg2004] // Flatten // Max
```

```
3
```

（Max は Flatten 属性を持っているから Flatten は省けるか）以下のような関数を作っておけば便利かもしれない（これも Flatten が省けるな）．

```
calcDiameter[g_] := Max[Flatten[AllPairsShortestPath[g]]]
```

```
calcDiameter[rg2004]
```

```
3
```

```
dlist = Table[{p, Mean[Map[calcDiameter, Table[RandomGraph[30, p], {50}]]]},
   {p, 0.1, 0.9, 0.05}];
```

```
dlist // Short
```

```
{{0.1, ∞}, {0.15, ∞}, {0.2, ∞}, ≪11≫ , {0.8, 2}, {0.85, 2}, {0.9, 2}}
```

ランダムグラフのリストを作って，直径の平均値の確率依存性を調べてみる．

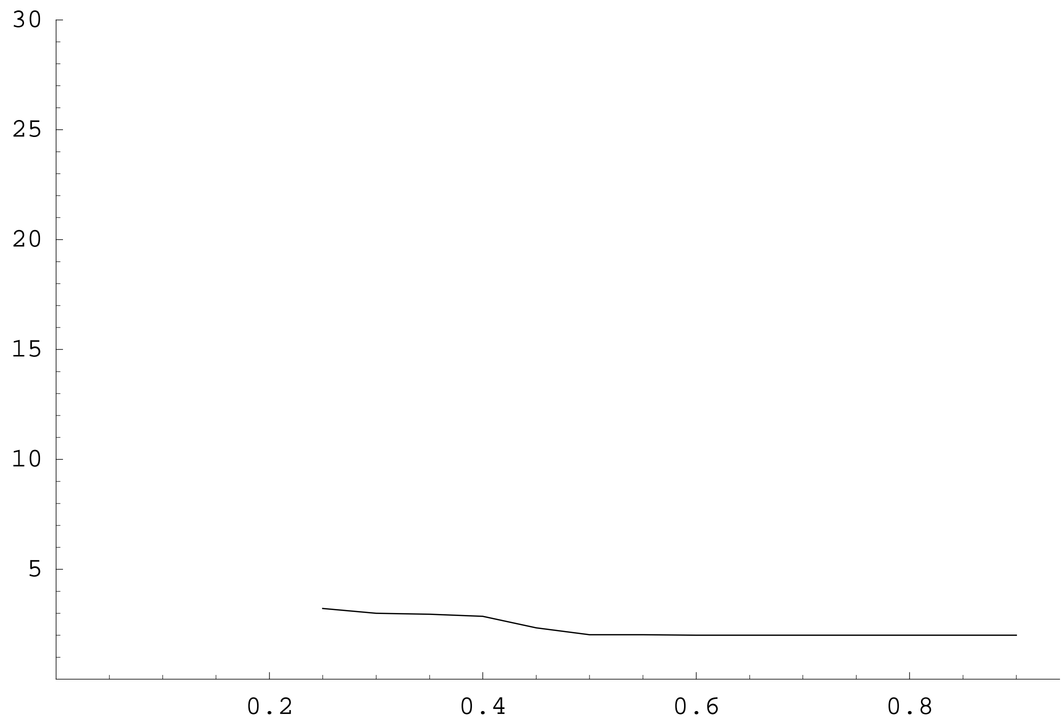
```
ListPlot[dlist, PlotJoined → True, PlotRange → {{0, 1}, {0, 30}}];
```

Graphics::gptn : {0.1, ∞}の座標∞は浮動小数点数ではありません． 詳細

Graphics::gptn : {0.15, ∞}の座標∞は浮動小数点数ではありません． 詳細

Graphics::gptn : {0.2, ∞}の座標∞は浮動小数点数ではありません． 詳細

General::stop : 計算中，Graphics::gptnのこれ以上の出力は表示されません． 詳細



以下はDiameter を知った後のコメント．

おそらくDiameter の実装は上の実装とほとんど変わらないのではないかと思われる．大きなネットワークだと時間がかかりすぎるのが難点だ．

■ 平均最短経路

直径と似たような性質を議論できる量として，平均最短経路がある．これは２点間の最短経路の平均である．すなわち，AllPairsShortestPath の要素を２点を選ぶ組み合わせ数の２倍で割った値である．

```
Apply[Plus, Flatten[AllPairsShortestPath[rg2004]]] /
  (V[rg2004] * (V[rg2004] - 1)) // N
```

```
1.64737
```

従って以下の関数で表現できる．

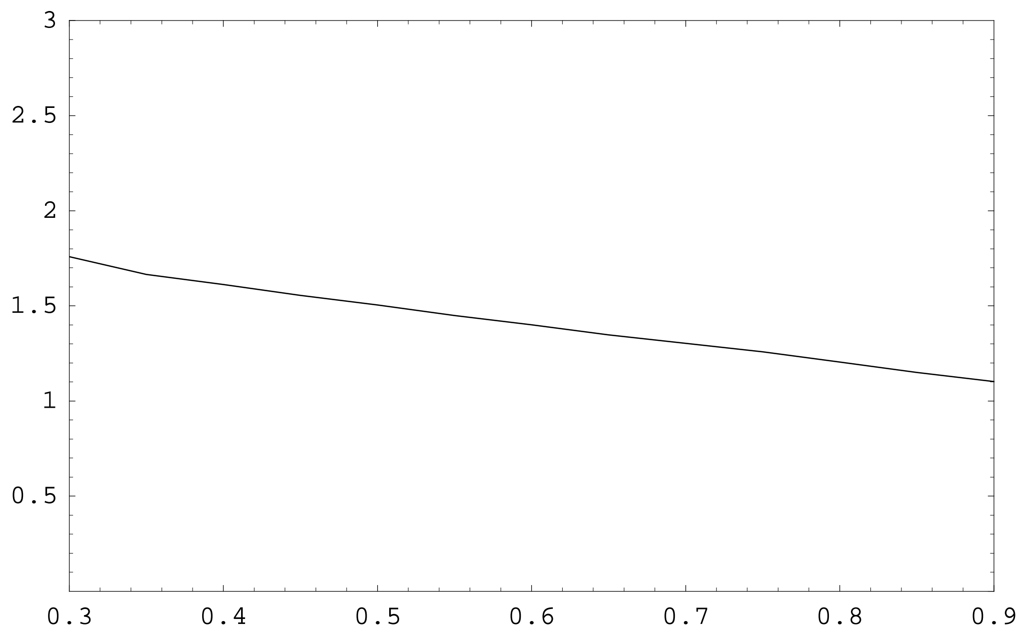
```
calcMeanPath[g_Graph] :=
 Apply[Plus, Flatten[AllPairsShortestPath[g]]] / (V[g] * (V[g] - 1))
```

これも，ランダムグラフのリストを作って，平均値の確率依存性を調べてみる．

```
(mplist = Table[{p, Mean[Map[calcMeanPath, Table[RandomGraph[30, p], {50}]]]},
    {p, 0.3, 0.9, 0.05}]) // Timing
```

$$\left\{36.641\ \text{Second}, \left\{\left\{0.3, \frac{19123}{10875}\right\}, \left\{0.35, \frac{36223}{21750}\right\}, \left\{0.4, \frac{2337}{1450}\right\}, \left\{0.45, \frac{6767}{4350}\right\}, \left\{0.5, \frac{1091}{725}\right\}, \left\{0.55, \frac{6307}{4350}\right\}, \left\{0.6, \frac{15229}{10875}\right\}, \left\{0.65, \frac{9769}{7250}\right\}, \left\{0.7, \frac{14167}{10875}\right\}, \left\{0.75, \frac{27373}{21750}\right\}, \left\{0.8, \frac{13106}{10875}\right\}, \left\{0.85, \frac{25021}{21750}\right\}, \left\{0.9, \frac{11986}{10875}\right\}\right\}\right\}$$

```
ListPlot[mplist, PlotJoined → True, PlotRange → {{0.3, 0.9}, {0, 3}}, Frame → True];
```



■ クラスター係数

クラスター係数（clustering coefficient**）は周囲の点との関係の深さを表す指標である．位数** k **の点は連結しているすべての点が互いに連結しているとき，** k(k-1)/2**個の辺で連結されることになる．この** k **個の点のうち，実際に連結している辺の数の** k(k-1)/2 **に対する辺をクラスター係数と呼ぶ．この値を隣接行列から求める方法を考える．**

```
adRg2004 = ToAdjacencyMatrix[rg2004];
```

ひとつの行について１がある列をピックアップしてみる．

```
findOne = Position[adRg2004[[1]], 1] // Flatten
```

```
{2, 4, 5, 12, 15, 16, 19, 20}
```

この集合から作られるペアは,

```
KSubsets[findOne, 2]
```

```
{{2, 4}, {2, 5}, {2, 12}, {2, 15}, {2, 16}, {2, 19}, {2, 20}, {4, 5},
 {4, 12}, {4, 15}, {4, 16}, {4, 19}, {4, 20}, {5, 12}, {5, 15},
 {5, 16}, {5, 19}, {5, 20}, {12, 15}, {12, 16}, {12, 19}, {12, 20},
 {15, 16}, {15, 19}, {15, 20}, {16, 19}, {16, 20}, {19, 20}}
```

なので，これらが連結している（すなわち要素が１）なのか，連結していない（すなわち要素が０）なのかを確かめればよい．これは各要素を抽出するだけの話なので，

```
Extract[adRg2004, KSubsets[findOne, 2]]
```

```
{0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0,
 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1}
```

これらの和を k(k-1)/2 で割った値が求めるクラスター係数である．

ここまでの話を整理して，すべての点について一度にクラスター係数を求められるような関数を作成してみることにする．

まず，各点における位数を求めるには，前述のように，隣接行列の各行について和をとればよいので，

```
Apply[Plus, adRg2004]
```

```
{8, 6, 6, 8, 6, 6, 10, 4, 8, 7, 6, 8, 11, 4, 8, 7, 7, 6, 5, 9}
```

で良い．これらを k(k-1)/2 倍するには，

```
degrees = Map[# (# - 1) / 2 &, Apply[Plus, adRg2004]]
```

```
{28, 15, 15, 28, 15, 15, 45, 6, 28, 21, 15, 28, 55, 6, 28, 21, 21, 15, 10, 36}
```

でよい．

次に各点についてクラスター係数の分子を求める方法を考える．上に述べた方法を順次あてはめていくには，

```
counts = Apply[Plus, Map[Extract[adRg2004, #] &, Map[KSubsets[#, 2] &,
    Map[Flatten, Map[Position[#, 1] &, adRg2004], {1}], {1}], {1}], {1}]
```

```
{9, 3, 6, 11, 5, 7, 15, 1, 9, 7, 6, 11, 12, 3, 13, 6, 9, 6, 3, 14}
```

このようにして作られたふたつのリストを互いに割ればいいので，

```
MapThread[Divide, {counts, degrees}]
```

$$\left\{\frac{9}{28}, \frac{1}{5}, \frac{2}{5}, \frac{11}{28}, \frac{1}{3}, \frac{7}{15}, \frac{1}{3}, \frac{1}{6}, \frac{9}{28}, \frac{1}{3}, \frac{2}{5}, \frac{11}{28}, \frac{12}{55}, \frac{1}{2}, \frac{13}{28}, \frac{2}{7}, \frac{3}{7}, \frac{2}{5}, \frac{3}{10}, \frac{7}{18}\right\}$$

と求められる.その平均は,

```
Mean[MapThread[Divide, {counts, degrees}]] // N
```

```
0.352377
```

以上の結果もひとつの関数にまとめられる．ただし位数がゼロの場合を考慮してないことに注意をする．

```
calcClusteringCoeff[g_] := Mean[MapThread[Divide,
    {Apply[Plus, Map[Extract[ToAdjacencyMatrix[g], #] &, Map[KSubsets[#, 2] &,
        Map[Flatten, Map[Position[#, 1] &, ToAdjacencyMatrix[g]], {1}], {1}], {1}],
      {1}], Map[# (# - 1) / 2 &, Apply[Plus, ToAdjacencyMatrix[g]]]}]];
```

```
calcClusteringCoeff[rg2004] // N
```

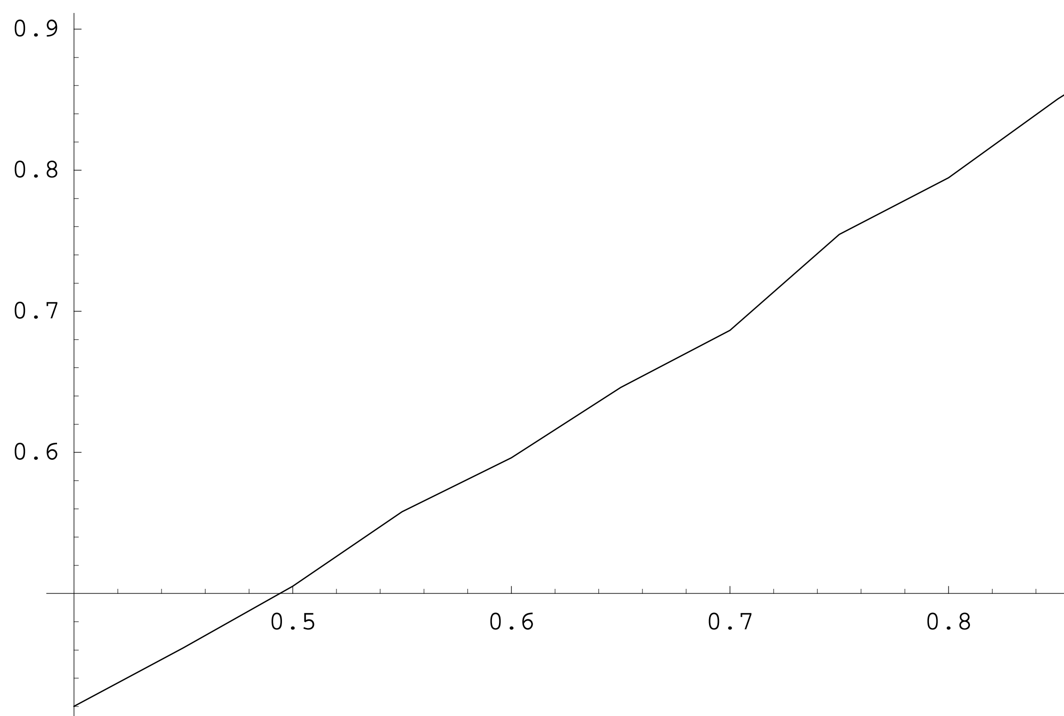
```
0.352377
```

```
(cflist =
   Table[{p, Mean[Map[calcClusteringCoeff, Table[RandomGraph[30, p], {10}]]]},
    {p, 0.4, 0.9, 0.05}]) // Timing
```

$$\left\{21.609\ \text{Second}, \left\{\left\{0.4, \frac{1222377613}{2909907000}\right\}, \left\{0.45, \frac{16115071121}{34918884000}\right\}, \left\{0.5, \frac{1764123899}{3491888400}\right\}, \left\{0.55, \frac{1566688877}{2808162000}\right\}, \left\{0.6, \frac{217667109497}{365061060000}\right\}, \left\{0.65, \frac{2593951019747}{4015671660000}\right\}, \left\{0.7, \frac{250624347061}{365061060000}\right\}, \left\{0.75, \frac{413136963107}{547591590000}\right\}, \left\{0.8, \frac{3191340703301}{4015671660000}\right\}, \left\{0.85, \frac{242790822257}{285427642500}\right\}, \left\{0.9, \frac{97258215979}{108162054000}\right\}\right\}\right\}$$

こいつも，ランダムグラフのリストを作って，平均値の確率依存性を調べてみる．

```
ListPlot[cflist, PlotJoined → True];
```

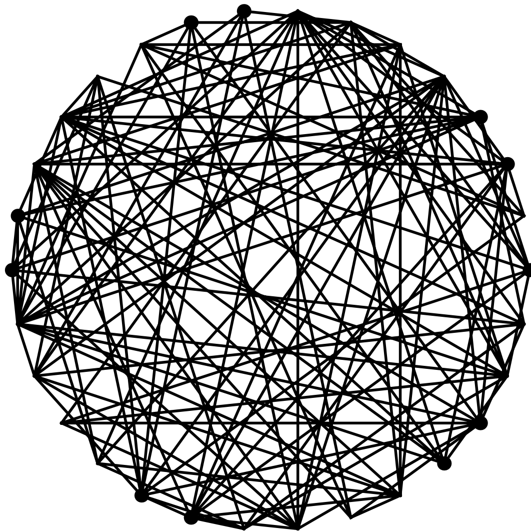


ランダムグラフ

ここではランダムグラフの性質について更に詳しく書く予定．続きを読みたいと思ったらリクエストメールを送ろう（笑）

```
rgh = RandomGraph[30, 0.3];
```

```
ShowGraph[rgh]
```



```
- Graphics -
```

ランダムグラフの集合を作って,その統計的な性質を見てみる.

```
lstGraph = Table[RandomGraph[30, 0.3], {1000}];
```

辺の本数を数えて平均を取ってみる.

```
Map[M, lstGraph] // Mean // N
```

```
130.456
```

解析解は以下のとおり.

```
Binomial[30, 2] * 0.3
```

```
130.5
```

このグラフの集合を用いて幾つかの性質を確認してみよう.例えば，ひとつのグラフについてその直径を求めるためには，Path の中で最も大きい値を求めればよい.

```
AllPairsShortestPath[rgh] // Flatten // Max
```

```
3
```

```
Map[Max[Flatten[AllPairsShortestPath[#]]] &, lstGraph] // Short
```

```
{3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, <<954>>,
 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3}
```

Small World Network

最近流行りの Small World Network **の性質を** Mathematica **を使って調べてみよう**.**以下ではグラフと同じ意味でネットワークという言葉を用いる．**

Small world の構成方法

Small World **と呼ばれるグラフには何種類かある．ここでは** Watts **と** Strongatz **の論文に書かれた方法とほぼ同じ方法で** Small World Network **を構築することを試みる．**

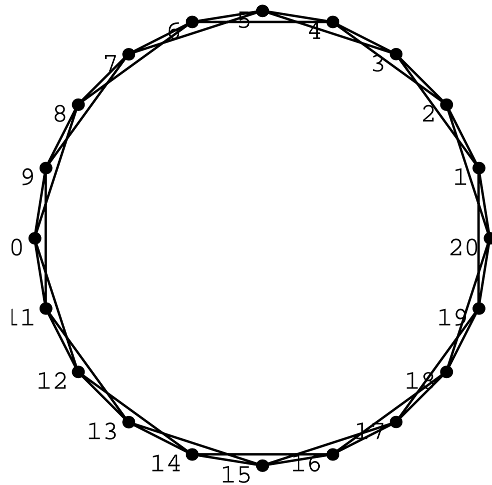
はじめに規則的なネットワークを作成しよう．前述の正則グラフを用いる．

```
makeSymmetricalRegularGraph[n_, k_] :=
 Apply[GraphSum, Map[CirculantGraph[n, #] &, Range[1, k / 2]]]
```

```
Edges[reg2004]
```

```
Edges[reg2004]
```

```
orig = makeSymmetricalRegularGraph[20, 4]; ShowGraph[orig, VertexNumber → On]
```

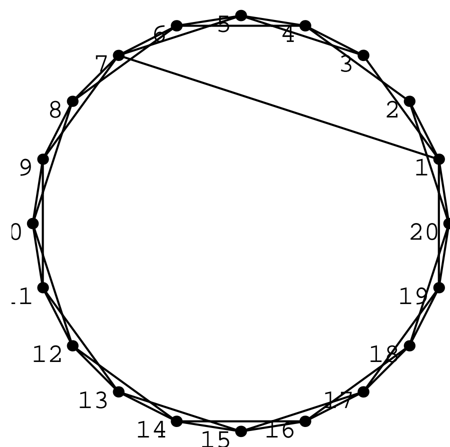


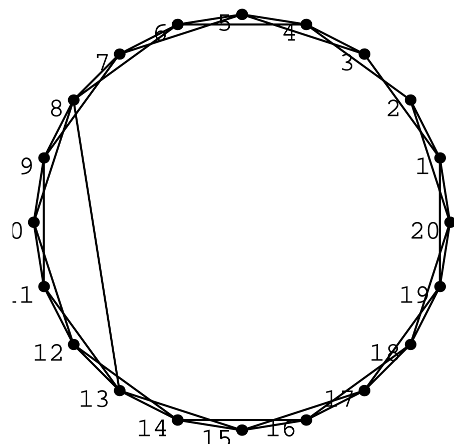

```
- Graphics -
```

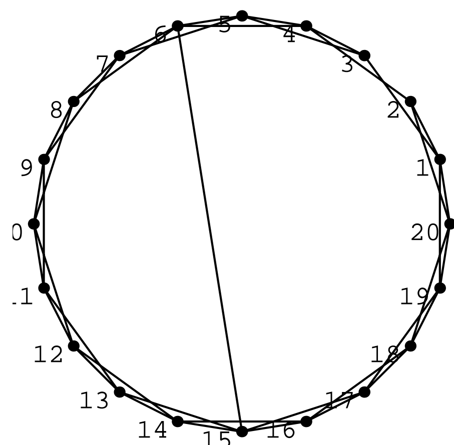
ひとつの辺を指定して，それを現在連結されてない二点を結ぶ辺に置き換える関数を作成する．

```
func[g_, olditem_] :=
 Module[{newg, newitem, elist, n},
  newg = g;
  n = V[g];
  elist = Edges[newg];
  While[MemberQ[elist,
      (newitem = {Random[Integer, {1, n - 1}], Random[Integer, {2, n}]})] ||
    (newitem[[1]] >= newitem[[2]])];
  (* ふたつの乱数の大小関係を確認しないといけない *)
  AddEdges[DeleteEdges[newg, olditem], newitem]
 ]
```

```
Table[ShowGraph[func[orig, {2, 3}], VertexNumber → On], {3}];
```

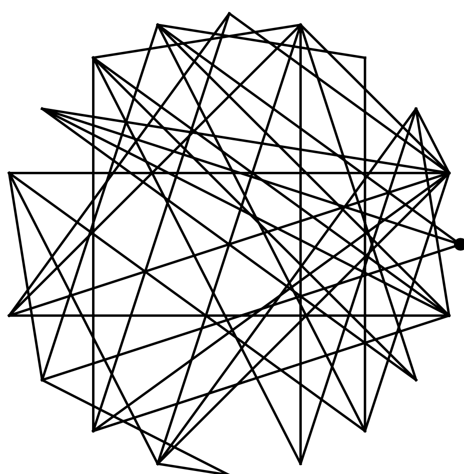






更にこれを全ての辺について適用するために Fold **を用いる.**

```
Fold[func, makeSymmetricalRegularGraph[20, 4],
  Edges[makeSymmetricalRegularGraph[20, 4]]] // ShowGraph
```



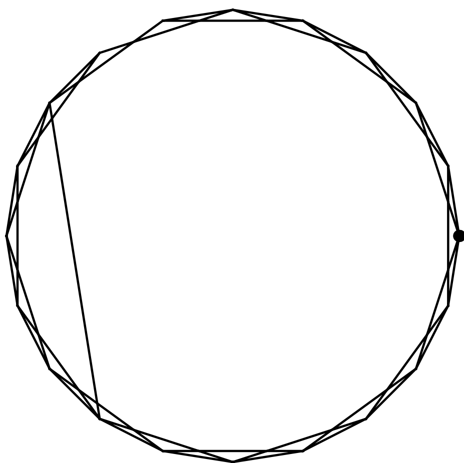
```
- Graphics -
```

Fold **する前に変更の有無を選択する関数を作る.**

```
answerG[g_, item_] := If[Random[] < 0.05, func[g, item], g];
```

これを用いれば Small World の完成．

```
Fold[answerG, makeSymmetricalRegularGraph[20, 4],
  Edges[makeSymmetricalRegularGraph[20, 4]]] // ShowGraph
```

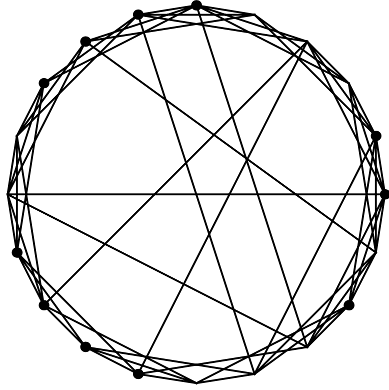


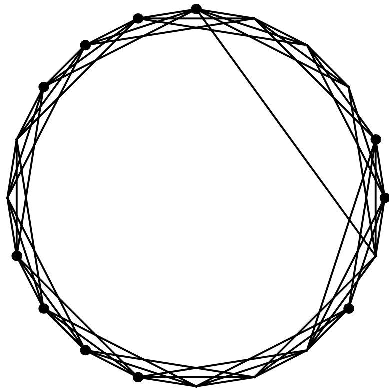
```
- Graphics -
```

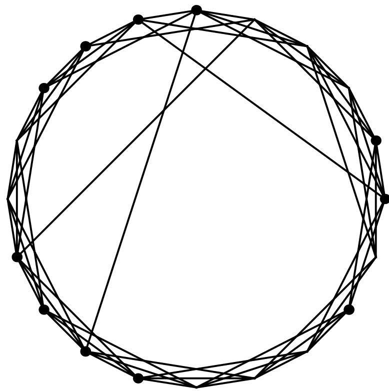
ひとつの関数にまとめておく.

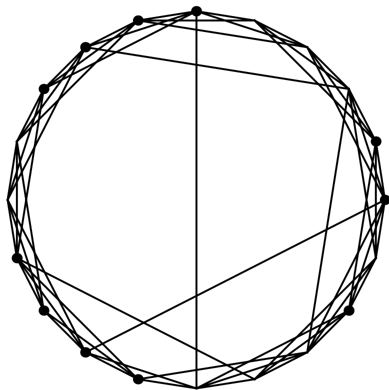
```
makeSmallWorld[n_, k_, p_] :=
 Module[{delandadd, ansG, makesymmetricalregulargraph},
  makesymmetricalregulargraph[ln_, lk_] :=
   Apply[GraphSum, Map[CirculantGraph[ln, #] &, Range[1, lk / 2]]];
  delandadd[g_, olditem_] := Module[{newg, newitem, elist},
    newg = g; elist = Edges[newg]; While[MemberQ[elist, (
        newitem = If[Random[] ≥ 0.5, {olditem[[1]], Random[Integer, {1, n}]},
          {Random[Integer, {1, n - 1}], olditem[[2]]}]
       )] || (newitem[[1]] >= newitem[[2]])]; AddEdges[
     DeleteEdges[newg, olditem], newitem]
   ];
  ansG[g_, item_] := If[Random[] < p, delandadd[g, item], g];
  Fold[ansG, makesymmetricalregulargraph[n, k],
   Edges[makesymmetricalregulargraph[n, k]]]
 ]
```

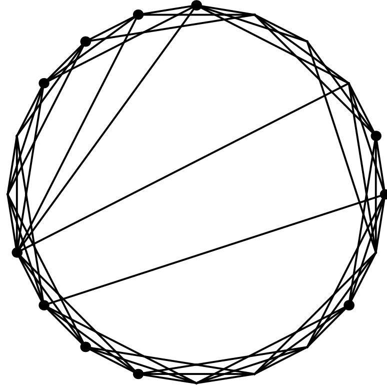
```
Table[makeSmallWorld[20, 6, 0.1] // ShowGraph, {5}];
```











```
Timing[(swglist =
    Table[Table[makeSmallWorld[1000, 8, p], {10}], {p, 0.0, 0.3, 0.05}])] // Short
```

```
{3639.5 Second,
 {{-Graph:<4000, 1000, Undirected>-, <<8>>, <<1>>}, <<6>>}}
```

```
dialst = Map[Diameter, swglist, {2}]; // Timing
```

```
{13399.7 Second, Null}
```

```
Map[Mean, dialst]
```

$\left\{125, \frac{56}{5}, \frac{89}{10}, \frac{15}{2}, 7, 7, \frac{31}{5}\right\}$

p が大きくなるとともに半径（の平均値）が小さくなることがわかる．

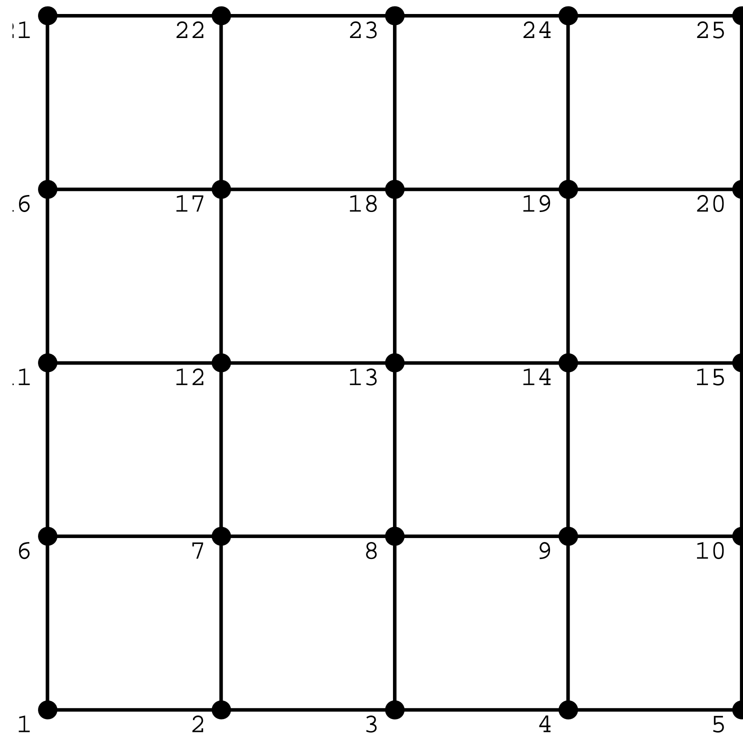
以下， Small World の性質についても調べたいが，これも次の更新に譲ることにしよう．リクエスト待ってます（来ないってw）

■ グリッドグラフに small world を適用した場合

周期境界条件を課した二次元正方格子上での small world を作成する．

はじめにグリッドグラフを作る．

```
ShowGraph[GridGraph[5, 5], VertexNumber → True]
```




```
- Graphics -
```

周期境界条件を用いたいので， $n \times n$ **のグリッドグラフに対して，** $i\,n$ **番目と** (in − 4) **番目，および** 1 **から** n **番目と** n(n-1) +1 **から** n(n-1) + n **までをそれぞれ繋げる．**

```
Join[Table[ {5 i , 5 i - 4}, {i, 1, 5}] , Table[{i, 5 (5 - 1) + i}, {i, 1, 5}]]
```

```
{{5, 1}, {10, 6}, {15, 11}, {20, 16},
 {25, 21}, {1, 21}, {2, 22}, {3, 23}, {4, 24}, {5, 25}}
```

```
AddEdges[GridGraph[5, 5],
 Join[Table[ {5 i , 5 i - 4}, {i, 1, 5}] , Table[{i, 5 (5 - 1) + i}, {i, 1, 5}]]]
```

```
-Graph:<50, 25, Undirected>-
```

```
ToAdjacencyMatrix[GridGraph[5, 5]] // MatrixForm
```

$$\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0
\end{pmatrix}$$

```
ToAdjacencyMatrix[AddEdges[GridGraph[5, 5], Join[Table[ {5 i , 5 i - 4}, {i, 1, 5}] ,
    Table[{i, 5 (5 - 1) + i}, {i, 1, 5}]]]] // MatrixForm
```

```
0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0
```

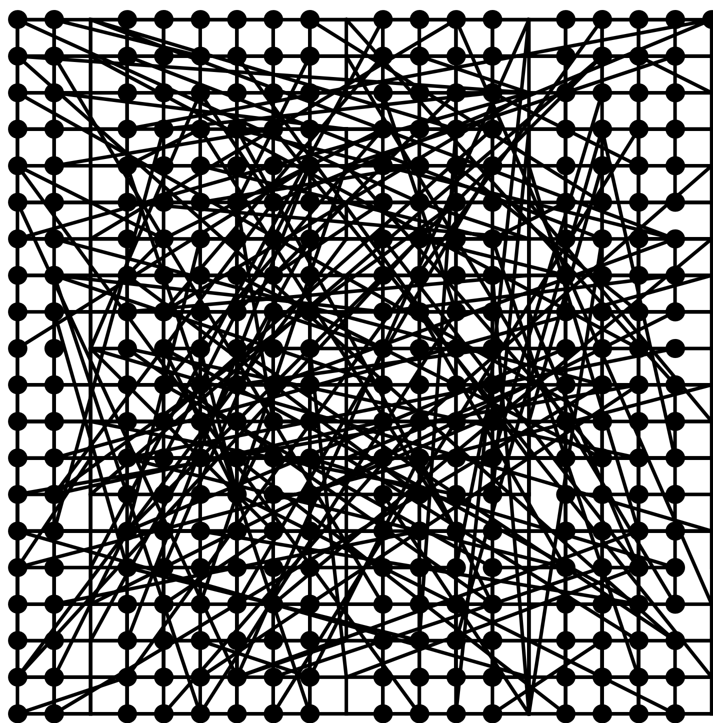
一般化しておこう．

```
makeRBCGraph[n_Integer] := AddEdges[GridGraph[n, n],
  Join[Table[ {n i , n i - (n - 1)}, {i, 1, n}] , Table[{i, n (n - 1) + i}, {i, 1, n}]]]
```

グラフができてしまえば，それに対して small world **を作ることはたやすい．上で作成したモジュールを一部変更すればよい．**

```
makeSmallWorldRBC[n_, p_] :=
 Module[{delandadd, ansG, targetG, makeRBCGraph, nVert, elist},
  makeRBCGraph[ln_Integer] :=
   AddEdges[GridGraph[ln, ln], Join[Table[ {ln i , ln i - (ln - 1)}, {i, 1, ln}] ,
     Table[{i, ln (ln - 1) + i}, {i, 1, ln}]]];
  targetG = makeRBCGraph[n];
  elist = Edges[targetG];
  nVert = V[targetG];
  delandadd[g_, olditem_] := Module[{newg, newitem, elist},
    newg = g; elist = Edges[newg]; While[MemberQ[elist, (
        newitem = If[Random[] ≥ 0.5, {olditem[[1]], Random[Integer, {1, nVert}]},
          {Random[Integer, {1, nVert - 1}], olditem[[2]]}]
       )] || (newitem[[1]] >= newitem[[2]])]; AddEdges[
     DeleteEdges[newg, olditem], newitem]
   ];
  ansG[g_, item_] := If[Random[] < p, delandadd[g, item], g];
  Fold[ansG, targetG, elist]
 ]
```

```
ShowGraph[makeSmallWorldRBC[20, 0.2]]
```



```
- Graphics -
```

接続を書くときに周期境界条件を使わないので中央に入れ替えが集中した印象があるが，見た目の問題だけなので気にしないことにする．

次に，このグラフの統計的な性質について調べてみる．

```
Table[Diameter[makeSmallWorldRBC[20, 0.6]], {10}]
```

```
{9, 9, 9, ∞, 10, ∞, ∞, ∞, ∞, ∞}
```

と，たっぷりと入れ替えてしまうと孤立する頂点が出現してしまう．少しに押さえておくと，どうなるかというと，

```
Table[Diameter[makeSmallWorldRBC[20, 0.2]], {10}] // Timing
```

```
{196.5 Second, {9, 9, 9, ∞, 9, 9, 9, 9, ∞, 9}}
```

直径はあまり変化がおきない．入れ替えがもう少し進まないと一番遠い部分がどうしても残ってしまうのかもしれない．そこで平均距離に注目してみることにする．幾つか確率を変えてみると，

```
graphList =
  Table[Table[calcMeanPath[makeSmallWorldRBC[20, p]], {20}], {p, 0.0, 0.3, 0.02}];
```

```
Map[Mean, graphList] // N
```

```
{10.0251, 7.61392, 6.84804, 6.34852, 6.06952, 5.79447, 5.61984,
 5.4705, 5.36007, 5.23972, ∞, 5.1062, 5.04825, 4.99082, ∞, ∞}
```

と，p が大きくなると計算ができないものの，小さくなっていくようである．

Scale Free Network

次に Barabási and Albert が提案した Scale Free Network について，その作り方と統計的な性質を見てみることにしよう．Networkの形成は２つのステップからなる．

１．成長（Growth）：小さな数
m_0 の頂点から始めて, 時間ステップが1だけ過ぎるごとに m ($\leq m_0$) 本の辺を持つ1個の頂点を追加してゆく．辺は既存の点と繋がるようにする．

２．選択結合（Preferential　Attachment）：その点の次数に比例した確率で繋がる．つまり，個々の点の次数を k_i とするとi 番目の頂点に連結する確率は $\Pi_i = k_i / \sum k_i$ である.

このアルゴリズムによって作られたネットワークが Scale Free Network と呼ばれるのは，次数 k を持つ点の存在確率 P(k) がべき乗則に従うことがわかっているためである．

```
scaleFreeNetwork[loop_Integer, m0_Integer: 3, m_Integer: 3] :=
  Module[{g, addlst, addNewVertex},
   (* ローカルな関数の定義 *)
   addNewVertex[grph_Graph] := Module[{selectPosition, n, problst, gr},
     n = V[grph];
     problst = Map[Total, ToAdjacencyMatrix[grph]] // # / Total[#] & //
       FoldList[Plus, 0, #] &;
     selectPosition[prlst_List] := Module[{rv},
       (* 連結の数に比例した確率分布を作る *)
       rv = Random[];
       Position[prlst,
          First[Select[prlst, GreaterEqual[#, rv] &, 1]], 1][[1, 1]] - 1
      ];
     (*Weight のリストを作る *)
     addlst = {};
     While[Length[addlst] < m,
      addlst = (selectPosition[problst] //
         If[! MemberQ[addlst, #], Append[addlst, #], addlst] &)];
     (* Print["Before"];
      ShowGraph[grph]; *)
     gr = AddVertex[grph, (n - m0) * {Random[], Random[]}];
     AddEdges[gr, (Map[List[#, V[gr]] &, addlst])]
    ];
   (* グラフの初期化 *)
   (* m0 個の頂点のみからなるグラフを作る *)
   g = RandomVertices[MakeGraph[Range[m0], , Type → Undirected]];
   (* 空のリストから連結相手の頂点番号のリストを作る *)
   addlst = {};
   While[Length[addlst] < m,
    addlst = (Random[Integer, {1, m0}] //
       If[! MemberQ[addlst, #], Append[addlst, #], addlst] &)];
   g = AddVertex[g, {0, 0}];
   g = AddEdges[g, Map[List[#, m0 + 1] &, addlst]];
   (* ここまでで初期化が終了 *)
   (* Table[g=addNewVertex[g],{loop-1}]  *)
   NestList[Evaluate[addNewVertex[#]] &, g, loop - 1]
   (* 最後の結果のみを欲しい場合には NestList を Nest に修正する *)
  ];
```

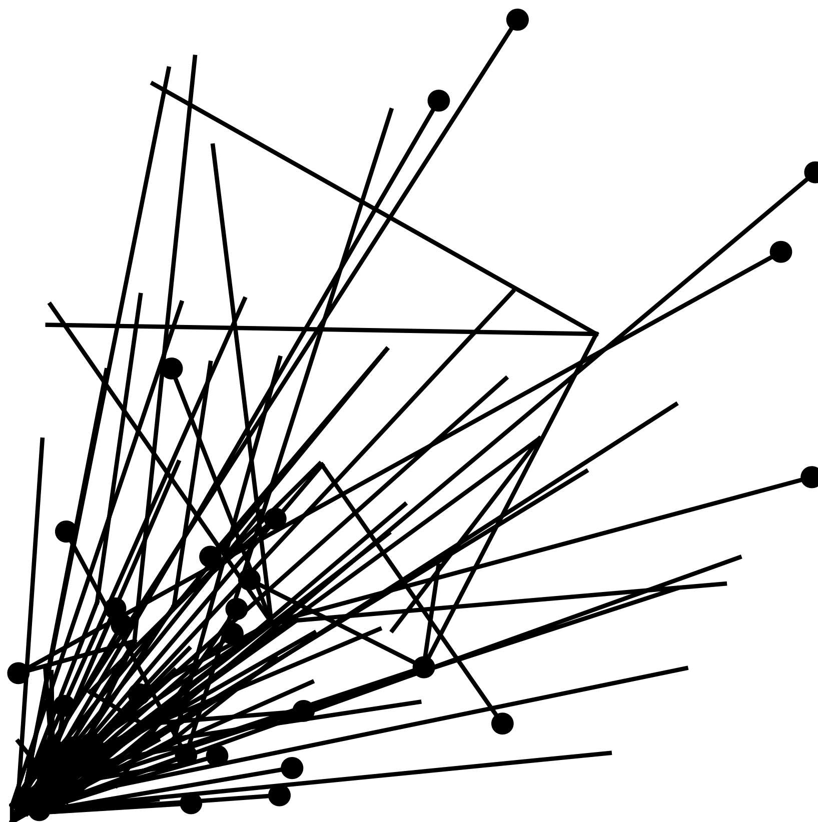
上記のプログラムが初期ヴァージョンだったのだが，$m < m_0$ のときに，最初に作る頂点が孤立してしまい，その後孤立したままになってしまう．そこで，最初の m_0 個については完全グラフを作るように変更した．それが以下のヴァージョンである．

```
scaleFreeNetwork[loop_Integer, m0_Integer:3, m_Integer:3] :=
  Module[{g, addlst, addNewVertex},
   (* ローカルな関数の定義 *)
   addNewVertex[grph_Graph] := Module[{selectPosition, n, problst, gr},
     n = V[grph];
     problst = Map[Total, ToAdjacencyMatrix[grph]] // # / Total[#] & //
       FoldList[Plus, 0, #] &;
     selectPosition[prlst_List] := Module[{rv},
       (* 連結の数に比例した確率分布を作る *)
       rv = Random[];
       Position[prlst,
          First[Select[prlst, GreaterEqual[#, rv] &, 1]], 1][[1, 1]] - 1
      ];
     (*Weight のリストを作る *)
     addlst = {};
     While[Length[addlst] < m,
      addlst = (selectPosition[problst] //
         If[! MemberQ[addlst, #], Append[addlst, #], addlst] &)];
     (* Print["Before"];
      ShowGraph[grph]; *)
     gr = AddVertex[grph, (n - m0) * {Random[], Random[]}];
     AddEdges[gr, (Map[List[#, V[gr]] &, addlst])]
    ];
   (* グラフの初期化 *)
   (* m0 個の頂点からなる完全グラフを作る *)
   g = CompleteGraph[m0];
   (* 空のリストから連結相手の頂点番号のリストを作る m0=
     1 の 場合にエラ-が出ないようにするために必要 *)
   addlst = {};
   While[Length[addlst] < m,
    addlst = (Random[Integer, {1, m0}] //
       If[! MemberQ[addlst, #], Append[addlst, #], addlst] &)];
   g = AddVertex[g, {0, 0}];
   g = AddEdges[g, Map[List[#, m0 + 1] &, addlst]];
   (* ここまでで初期化が終了 *)
   (* Table[g=addNewVertex[g],{loop-1}]  *)
   NestList[Evaluate[addNewVertex[#]] &, g, loop - 1]
   (* 最後の結果のみを欲しい場合には NestList を Nest に修正する *)
  ];
```

たとえば $m_0 = 3,\ m = 1$ を100世代まで作ってみると以下のようになる．

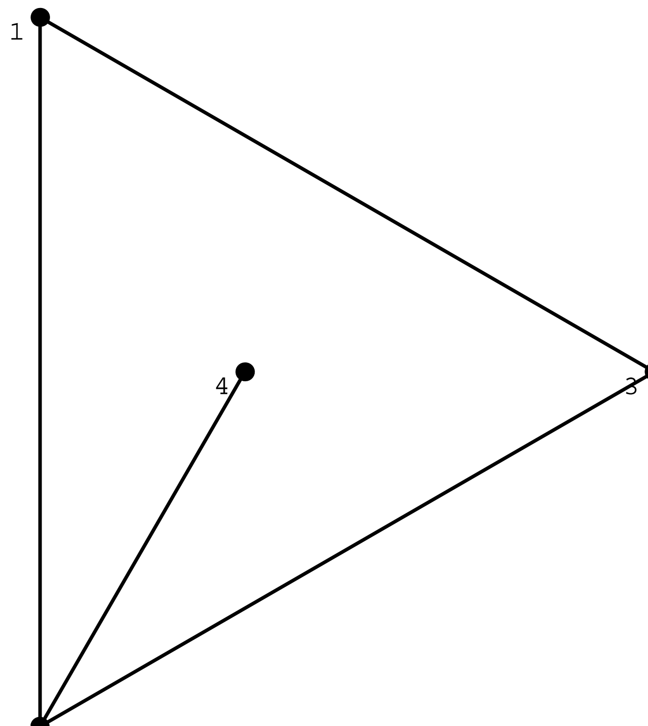
```
reslst = scaleFreeNetwork[100, 3, 1];
```

```
ShowGraph[Last[reslst]];
```



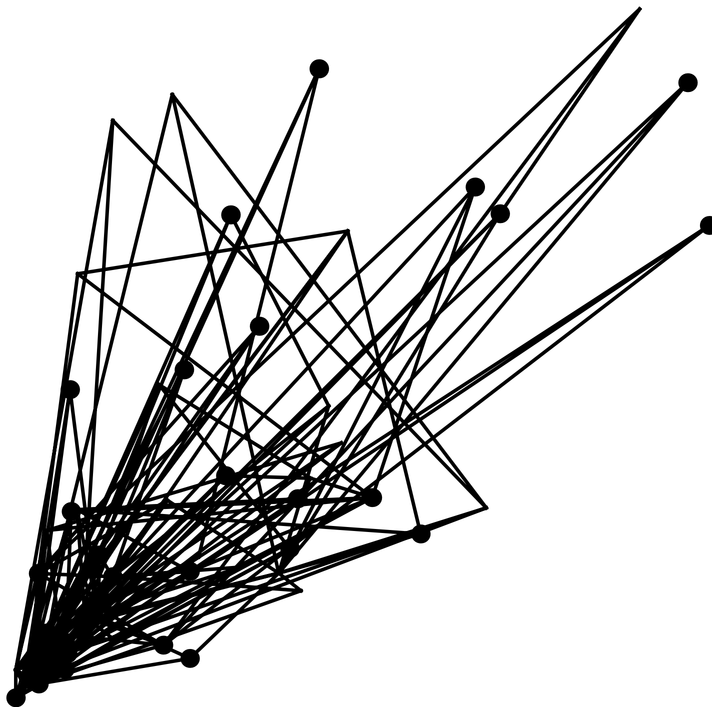
シーケンシャルに追加の様子を見る場合には以下のような感じ．

```
Map[ShowGraph[#, VertexNumber → True, PlotRange → {{0, 1}, {0, 1}}] &,
  reslst[[Range[1, 100, 10]]]];
```



$m_0 = m = 3$ を50世代の場合には以下のようになる．

```
scaleFreeNetwork[50, 3, 3] // Last // ShowGraph
```



```
- Graphics -
```

プログラムの説明や解析結果については後日．簡単にメモをつけておくと，最終結果の平均経路は，

```
calcMeanPath[Last[reslst]]
```

$$\frac{7230}{1751}$$

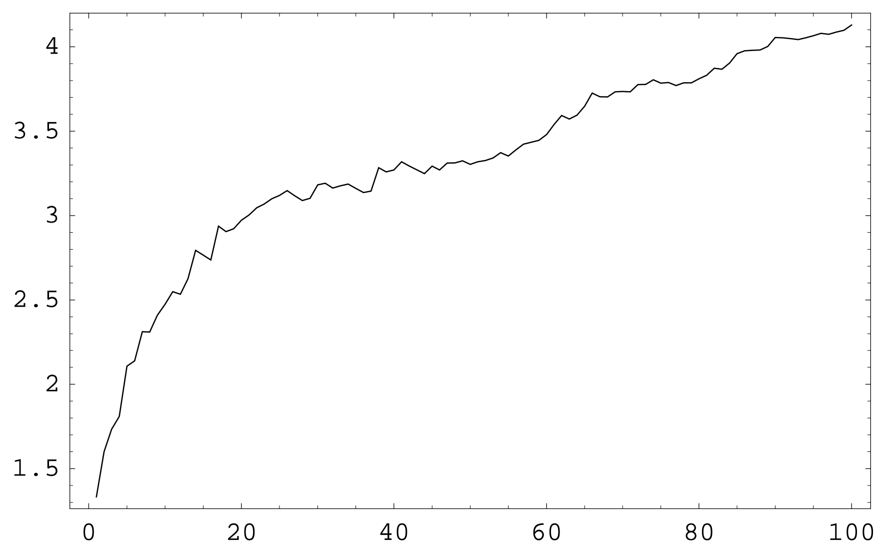
```
%% // N
```

```
- Graphics -
```

で，計算できる．もしも，グラフの形成の過程でどのように値が変化してゆくのかを知りたいのならば，以下の手順で行えばよい．

```
pathlst = Map[calcMeanPath, reslst];
```

```
ListPlot[pathlst, PlotJoined → True, Frame → True];
```



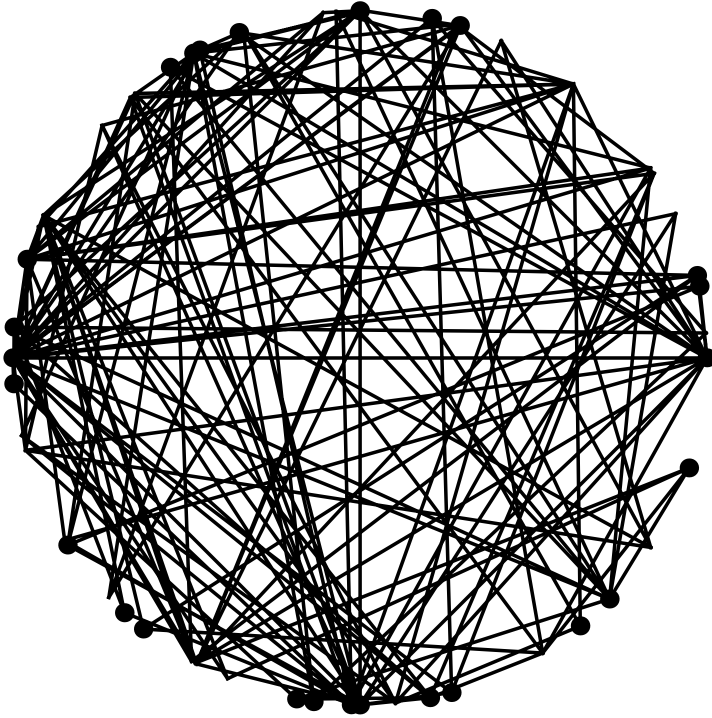
100 ぐらいで飽和しているようなので，それを確かめておく．

■ グラフの表示

Scale Free Network を円周上の点を用いて表示したいというリクエストが来たので，ランダムに点を置くものと，等間隔に点を置くものを作ってみた．もうひとつパラメータを置いてオーバーロードさせようかとも思ったけれど，紹介するだけにとどめる．

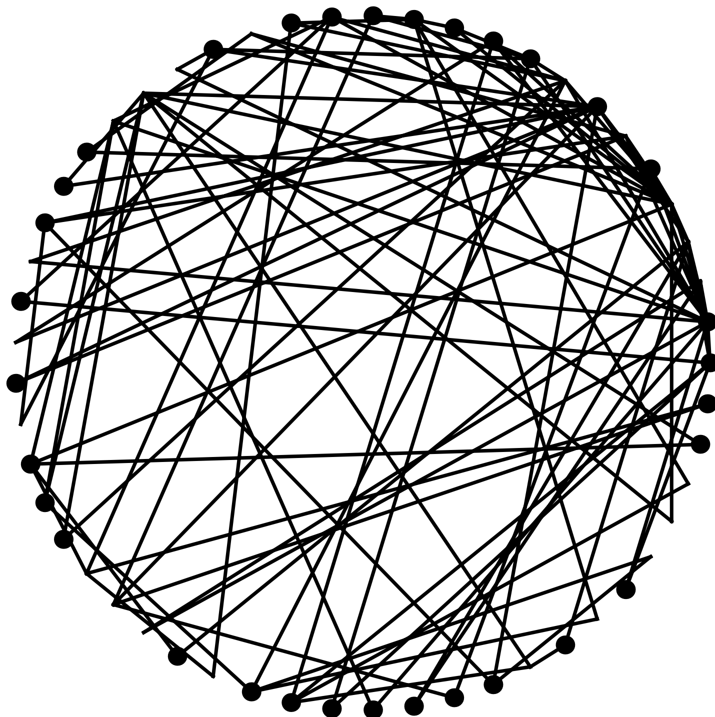
```
scaleFreeNetwork2[loop_Integer, m0_Integer: 3, m_Integer: 3] :=
  Module[{g, addlst, addNewVertex},
   (* ローカルな関数の定義 *)
   addNewVertex[grph_Graph] := Module[{selectPosition, n, problst, gr},
     n = V[grph];
     problst = Map[Total, ToAdjacencyMatrix[grph]] // # / Total[#] & //
       FoldList[Plus, 0, #] &;
     selectPosition[prlst_List] := Module[{rv, theta},
       (* 連結の数に比例した確率分布を作る *)
       rv = Random[];
       Position[prlst,
          First[Select[prlst, GreaterEqual[#, rv] &, 1]], 1][[1, 1]] - 1
      ];
     (*Weight のリストを作る *)
     addlst = {};
     While[Length[addlst] < m,
      addlst = (selectPosition[problst] //
         If[! MemberQ[addlst, #], Append[addlst, #], addlst] &)];
     (* Print["Before"];
      ShowGraph[grph]; *)
     theta = 2 Pi Random[]; (* !!! でたらめな角度を決めて *)
     gr = AddVertex[grph, {Cos[theta], Sin[theta]}];
     (* その角度で円周上の点を決める *)
     AddEdges[gr, (Map[List[#, V[gr]] &, addlst])]
    ];
   (* グラフの初期化 *)
   (* m0 個の頂点からなる完全グラフを作る *)
   g = CompleteGraph[m0];
   (* 空のリストから連結相手の頂点番号のリストを作る m0=
     1 の 場合にエラ-が出ないようにするために必要 *)
   addlst = {};
   While[Length[addlst] < m,
    addlst = (Random[Integer, {1, m0}] //
       If[! MemberQ[addlst, #], Append[addlst, #], addlst] &)];
   g = AddVertex[g, {Cos[E], Sin[E]}]; (* 最初の点は対称性が関係ない点にした *)
   g = AddEdges[g, Map[List[#, m0 + 1] &, addlst]];
   (* ここまでで初期化が終了 *)
   (* Table[g=addNewVertex[g],{loop-1}]  *)
   NestList[Evaluate[addNewVertex[#]] &, g, loop - 1]
   (* 最後の結果のみを欲しい場合には NestList を Nest に修正する *)
  ];
```

```
ShowGraph[Last[scaleFreeNetwork2[50, 4, 3]]];
```



```
scaleFreeNetwork3[loop_Integer, m0_Integer: 3, m_Integer: 3] :=
  Module[{cnt, g, addlst, addNewVertex},
   (* ローカルな関数の定義 *)
   cnt = m0 + 1; (* 追加する点の番号は m0 +1 からスタ-ト *)
   addNewVertex[grph_Graph] := Module[{selectPosition, n, problst, gr},
     n = V[grph];
     problst = Map[Total, ToAdjacencyMatrix[grph]] // # / Total[#] & //
       FoldList[Plus, 0, #] &;
     selectPosition[prlst_List] := Module[{rv, theta},
       (* 連結の数に比例した確率分布を作る *)
       rv = Random[];
       Position[prlst,
          First[Select[prlst, GreaterEqual[#, rv] &, 1]], 1][[1, 1]] - 1
      ];
     (*Weight のリストを作る *)
     addlst = {};
     While[Length[addlst] < m,
      addlst = (selectPosition[problst] //
         If[! MemberQ[addlst, #], Append[addlst, #], addlst] &)];
     (* Print["Before"];
      ShowGraph[grph]; *)
     (* theta=2 Pi Random[]; *)(* !!! でたらめな角度を決めて *)
     theta = 2.0 Pi cnt / (m0 + loop);
     gr = AddVertex[grph, {Cos[theta], Sin[theta]}];
     (* その角度で円周上の点を決める *)
     cnt++; (* 追加する点の番号をひとつ増やす *)
     AddEdges[gr, (Map[List[#, V[gr]] &, addlst])]
    ];
   (* グラフの初期化 *)
   (* m0 個の頂点からなる完全グラフを作る *)
   g = CompleteGraph[m0];
   g = ChangeVertices[g,
     Table[{Cos[2 Pi (i - 1) / (m0 + loop)], Sin[2 Pi (i - 1) / (m0 + loop)]}, {i, m0}]];
   (* 空のリストから連結相手の頂点番号のリストを作る m0=
     1 の 場合にエラ-が出ないようにするために必要 *)
   addlst = {};
   While[Length[addlst] < m,
    addlst = (Random[Integer, {1, m0}] //
       If[! MemberQ[addlst, #], Append[addlst, #], addlst] &)];
   g = AddVertex[g, {Cos[2.0 Pi (cnt - 1) / (m0 + loop)],
      Sin[2.0 Pi (cnt - 1) / (m0 + loop)]}];
   g = AddEdges[g, Map[List[#, m0 + 1] &, addlst]];
   (* ここまでで初期化が終了 *)
   (* Table[g=addNewVertex[g],{loop-1}]  *)
   NestList[Evaluate[addNewVertex[#]] &, g, loop - 1]
   (* 最後の結果のみを欲しい場合には NestList を Nest に修正する *)
  ];
```

```
ShowGraph[Last[scaleFreeNetwork3[50, 3, 2]]];
```



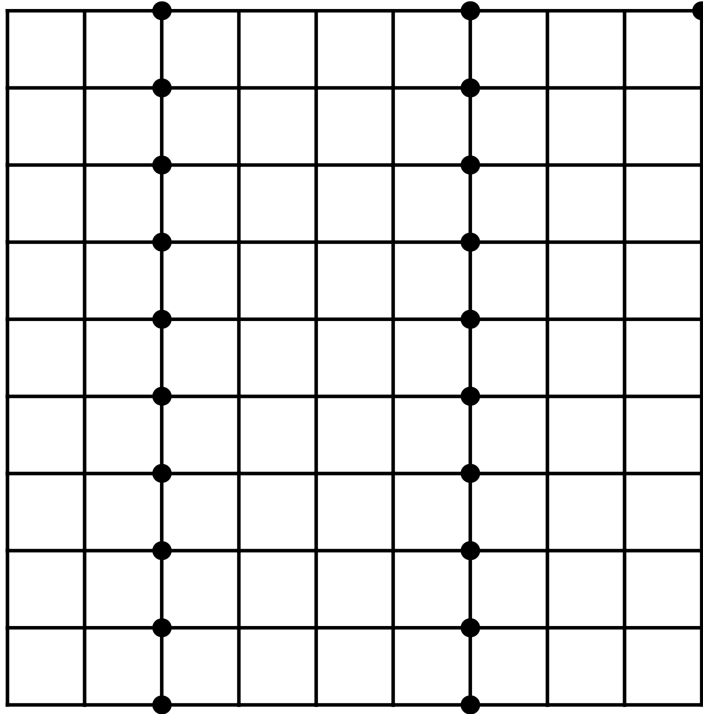
Combinatorica でパーコレーション

うまく書けるかどうかわかからないけど，できそうなことは書いてみよう．とりあえず，combinatorica を読み込むところは同じである．

```
<< DiscreteMath`Combinatorica`
```

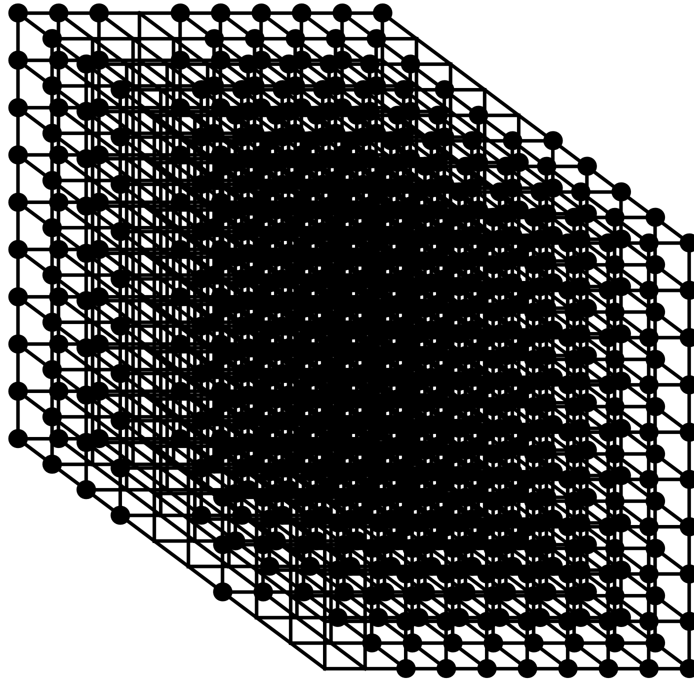
おそらく使うのは Grid グラフぐらいなものだろう．作り方をもう一度確認しておくと，

```
(grph2d = GridGraph[10, 10]) // ShowGraph
```



```
- Graphics -
```

```
(grph3d = GridGraph[10, 10, 10]) // ShowGraph
```

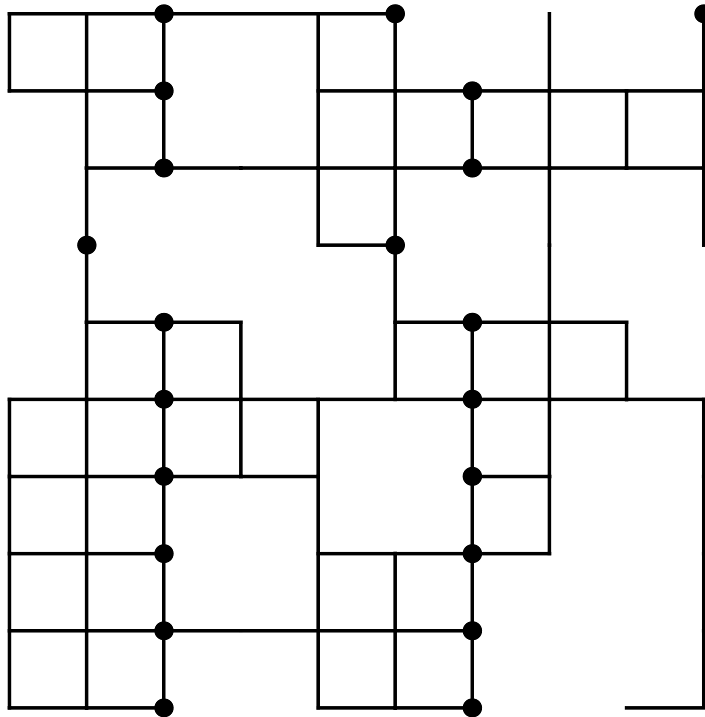


```
- Graphics -
```

Conbinatorica 的に言えば，パーコレーションとは辺の生き残りゲームを行うことだ．

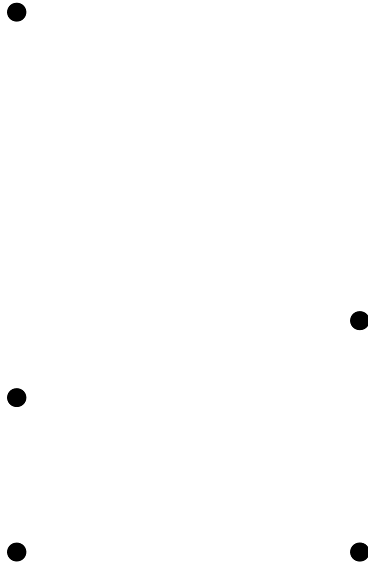
```
s = InduceSubgraph[grph2d, RandomKSubset[Range[100], 80]];
```

```
ShowGraph[s];
```



上のようにひとつの方法として，生き残る点の数を与えてランダムに点を選ぶことで，新しいグラフを作ることができる．生き残る点の数を変えると，徐々に連結されていく様子がわかる．いわゆるサイトパーコレーションである．下の図はダブルクリックすると徐々に接続する辺の数が増えてゆく．しかし，RandomKSubset を使っているので辺は固定ではない．

```
Table[ShowGraph[InduceSubgraph[grph2d, RandomKSubset[Range[100], i]]],
  {i, 10, 90, 5}];
```



繋がった辺を消すことなく増やしてゆくためには，点の数を追加してゆくような関数を作ればよいはずである．少し格好悪いけど，こんな関数を作ってみよう．

```
randomSwap[sourcelst_List] :=
  Module[{newlst = {}, restlst = sourcelst, randomlst},
   randomlst = Map[Random[Integer, {1, #}] &, Reverse[Range[Length[sourcelst]]]];
   addAnddel[{nl_List, rl_List}, loc_Integer] :=
    {Append[nl, rl[[loc]]], Delete[rl, loc]};
   Fold[addAnddel, {newlst, restlst}, randomlst] // First
  ];
```

```
randomSwap[Range[20]]
```

```
{8, 13, 18, 12, 9, 6, 11, 15, 17, 16, 5, 14, 1, 4, 20, 2, 7, 3, 19, 10}
```

```
randomSwap[{a, b, c, d, e}]
```

```
{c, e, a, d, b}
```

```
(randomSwap[Range[200]] // Sort) == Range[200]
```

```
True
```

この関数を作ると以下のようにして，関数をでたらめに番号を増やしていくことができる．

```
randomSwap[Range[10]] // Table[#[[Range[i]]], {i, 10}] &
```

```
{{6}, {6, 5}, {6, 5, 1}, {6, 5, 1, 10}, {6, 5, 1, 10, 9},
 {6, 5, 1, 10, 9, 2}, {6, 5, 1, 10, 9, 2, 8}, {6, 5, 1, 10, 9, 2, 8, 3},
 {6, 5, 1, 10, 9, 2, 8, 3, 4}, {6, 5, 1, 10, 9, 2, 8, 3, 4, 7}}
```

下のグラフをクリックして，頂点が増えることで浸透が進んでゆく様子を確認できる．

```
sublst = randomSwap[Range[V[grph2d]]] // Table[#[[Range[i]]], {i, V[grph2d]}] & ;
```

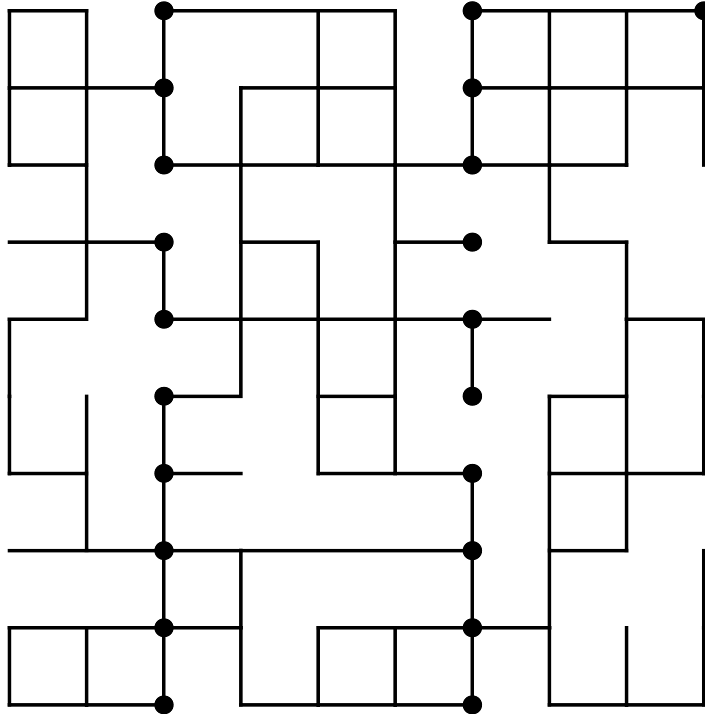
```
glst = Map[InduceSubgraph[grph2d, #] &, sublst];
```

```
Map[ShowGraph, glst[[Range[1, V[grph2d], 5]]]];
```

上の図はクリックすればわかれば，１回の描画で頂点が５点ずつ増えていく．何をやっているのかは明白だろう．

同じようにしてボンドパーコレーションを考えられないだろうか？確率を与えればよいのならこんな方法でいいのかな．

```
(g = DeleteEdges[grph2d, RandomKSubset[Edges[grph2d], 54]]) // ShowGraph;
```



```
Length[Edges[grph2d]]
```

```
180
```

とりあえずパーコレーションと同じ状態のグラフを作ることはできそうである．では，これを解析するにはどうしたらよいのだろうか？サイトパーコレーションの場合には，明らかにConnectedQ では解析が十分ではない．なぜなら，ひとつでも孤立した点があると False を返すこの関数は，端から端まで繋がったグラフがあるのかを判断するパーコレーションの枠組みに合わないからだ．逆にボンドパーコレーションの場合には ConnectedQ でもいいような気がしていたり．微妙だ…

文献を漁ってみると，ゴールド＆トボニチク「計算物理学入門」にこんなことが書いてあった．「格子の大きさが有限だと，格子の端から端まで連結したことの定義には任意性がある．たとえば，連結した経路の定義として，(i)格子全体を横方向か縦方向のどちらかに連結した経路，(ii)ある定められた方向，例えば縦方向に連結した経路，あるいは，(iii)横方向と縦方向の両方に連結した経路，いずれを用いてもよい」だそうだ．じゃぁ，どれを使おうかという話になるんだが，結局システムサイズ無限大に外装してしまえば，どれも変わらないということのようである．

ということで ConnectedQ を用いてグラフの成長を調べてみると…

```
Map[ConnectedQ, glst]
```

```
{True, False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False, False, False, False,
 False, False, False, False, False, False, False, False, False,
 True, True, True, True, True, True, True, True, True, True, True,
 True, True, True, True, True, True, True, True, True, True, True}
```

なんとなくいい感じなんだけど，最初の True **が**… ConnectedQ **の定義を修正しようかとも思ったのだが，いちいち修正してから使うのも大変なので最初が邪魔なときは** Rest **を使って最初だけ切り落としてみることにする．**

```
((Cases[Map[ConnectedQ, glst], False] // Length) + 1) / (glst // Length) // N
```

```
0.78
```

という感じで，全体が繋がるまでの確率が計算できた．サイトパーコレーションの確率としては大きすぎ．幾つか試してみることにする．

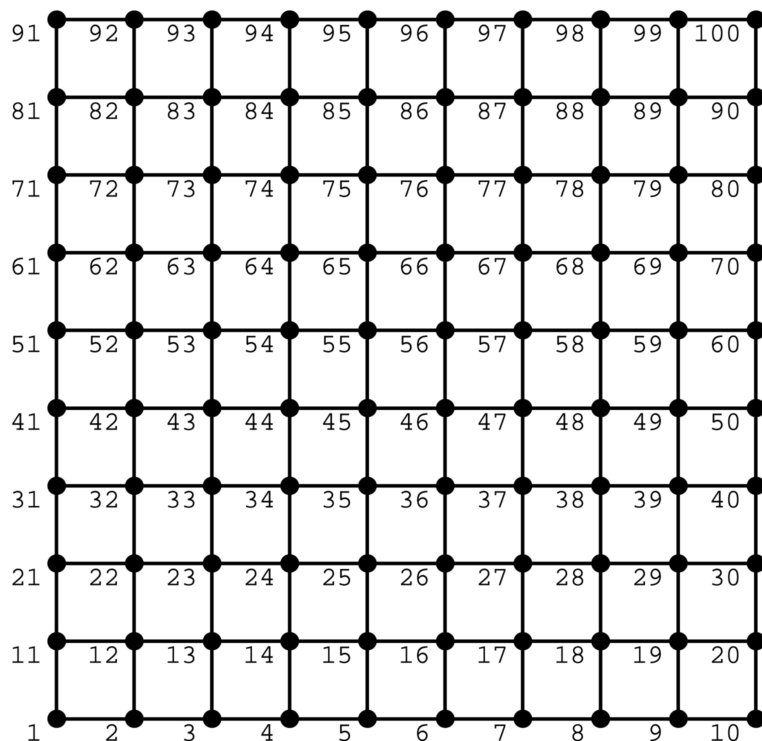
```
makeConnectionSequence[grph_Graph] := Module[{randomSwap, sublst},
  randomSwap[sourcelst_List] :=
   Module[{newlst = {}, restlst = sourcelst, randomlst},
    randomlst = Map[Random[Integer, {1, #}] &,
      Reverse[Range[Length[sourcelst]]]];
    addAnddel[{nl_List, rl_List}, loc_Integer] :=
     {Append[nl, rl[[loc]]], Delete[rl, loc]};
    Fold[addAnddel, {newlst, restlst}, randomlst] // First
   ];
  sublst = randomSwap[Range[V[grph]]] // Table[#[[Range[i]]], {i, V[grph]}] & ;
  Map[InduceSubgraph[grph, #] &, sublst]
 ]
```

```
Table[
 ((Cases[Map[ConnectedQ, makeConnectionSequence[grph2d]], False] // Length) + 1) /
   (glst // Length) // N, {10}]
```

```
{0.78, 0.81, 0.84, 0.74, 0.94, 0.75, 0.61, 0.8, 0.81, 0.77}
```

境界の影響かもしれないということで，周期境界条件の場合を試してみる．

```
ShowGraph[GridGraph[10, 10], VertexNumber → True, PlotRange → All];
```



10×10 のシステムならば，横方向は 10 (i -1) +1 と 10i を結び，縦方向には i と 10×(10-1)+i を結べば良さそうだ．

```
Join[Table[{10 (i - 1) + 1, 10 i}, {i, 10}], Table[{i, 10 (10 - 1) + i}, {i, 10}]]
```

```
{{1, 10}, {11, 20}, {21, 30}, {31, 40}, {41, 50}, {51, 60},
 {61, 70}, {71, 80}, {81, 90}, {91, 100}, {1, 91}, {2, 92}, {3, 93},
 {4, 94}, {5, 95}, {6, 96}, {7, 97}, {8, 98}, {9, 99}, {10, 100}}
```

```
g = AddEdges[GridGraph[10, 10], %23]
```

```
AddEdges[-Graph:<180, 100, Undirected>-, - Graphics -]
```

```
Map[ConnectedQ, makeConnectionSequence[g]]
```

```
makeConnectionSequence[
 ConnectedQ[AddEdges[-Graph:<180, 100, Undirected>-, - Graphics -]]]
```

全体の数に対する False の数の比が閾値になるんじゃないかと思われる．

```
Table[((Cases[Map[ConnectedQ, makeConnectionSequence[g]], False] // Length) + 1) /
   (glst // Length) // N, {20}]
```

```
{0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01}
```

少し小さくなったけど，まだ大きめ．計算時間がかかるのを覚悟してシステムサイズを大きくしてみると…

```
makeRepeatedBoundaryGrid[n_Integer] := AddEdges[GridGraph[n, n],
  Join[Table[{n (i - 1) + 1, n i}, {i, n}], Table[{i, n (n - 1) + i}, {i, n}]]]
```

```
Table[
  ((Cases[Map[ConnectedQ, makeConnectionSequence[makeRepeatedBoundaryGrid[20]]],
        False] // Length) + 1) / (20^2) // N, {10}] // Timing
```

```
{142.797 Second, {0.7575, 0.7825, 0.84,
  0.7525, 0.805, 0.6925, 0.7775, 0.9025, 0.7625, 0.8025}}
```

やはり大きめに出てしまう…もっとシステムサイズを大きくして外挿を試みるかとも思ったのだけど，仕事で使っている

コンピュータがずっと占有されてしまったので３日で中止した．後日，どこかで走らせてみることにする．

```
(res = Table[Table[
      ((Cases[Map[ConnectedQ, makeConnectionSequence[makeRepeatedBoundaryGrid[
                i]]], False] // Length) + 1) /
        (i^2) // N, {100}], {i, 5, 50, 5}]); // Timing
```

```
$Aborted
```

さまざまな network 上での囚人のジレンマゲーム

「囚人のジレンマ」はゲーム理論ではもっとも重要なゲームである．この囚人のジレンマをある戦略に基づいて繰り返し実施したときに結果がどのように変わっていくのかを以下の要領でしらべてみることにする．戦略の選択の仕方などの詳細は Hauert and Doebeli **を参照のこと．**

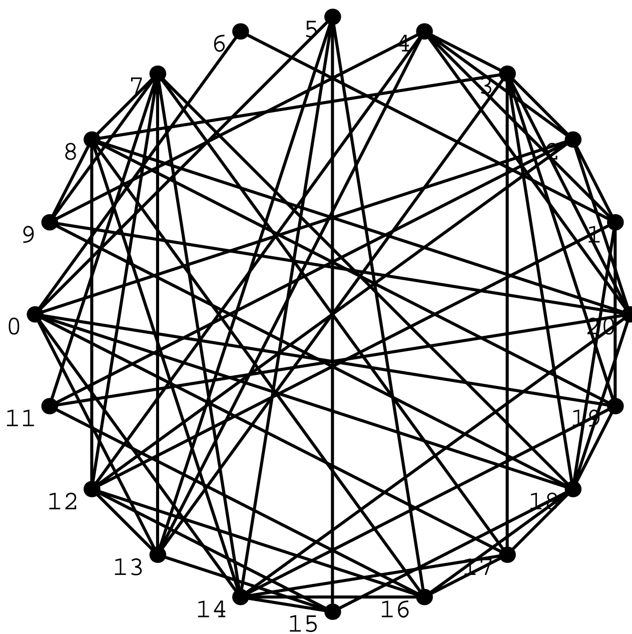
プログラム作成のための簡単な覚書

Combinatorica のパッケージを読む．

```
<< DiscreteMath`Combinatorica`
```

いい加減なグラフを作ってそのグラフで動作確認してゆく．

```
rg2004 = RandomGraph[20, 0.4]; ShowGraph[rg2004, VertexNumber → True];
```




```
adjacencies = ToAdjacencyLists[rg2004]
```

```
{{2, 4, 6, 12, 18, 19}, {1, 3, 4, 10, 11, 12, 20},
 {2, 4, 8, 13, 17, 18, 19, 20}, {1, 2, 3, 9, 12, 13, 20},
 {10, 13, 14, 15, 16}, {1, 10}, {8, 9, 11, 12, 13, 14, 17, 18},
 {3, 7, 9, 12, 14, 16, 19, 20}, {4, 7, 8, 18, 20},
 {2, 5, 6, 13, 14, 17, 18, 19}, {2, 7, 16, 20}, {1, 2, 4, 7, 8, 13, 15, 16},
 {3, 4, 5, 7, 10, 12, 15}, {5, 7, 8, 10, 15, 16, 17, 19, 20},
 {5, 12, 13, 14, 18}, {5, 8, 11, 12, 14, 17, 18},
 {3, 7, 10, 14, 16, 18}, {1, 3, 7, 9, 10, 15, 16, 17, 19, 20},
 {1, 3, 8, 10, 14, 18}, {2, 3, 4, 8, 9, 11, 14, 18}}
```

例えば５番目の vartex と連結していある相手を知りたい場合には５番目の要素を抽出すればよい．

```
adjacencies[[5]]
```

```
{10, 13, 14, 15, 16}
```

Thread **を使ってふたつの連結している点を関数** f **に送ろう．**

```
Thread[f[5, adjacencies[[5]] ]]
```

```
{f[5, 10], f[5, 13], f[5, 14], f[5, 15], f[5, 16]}
```

各プレーヤーの現在の strategy **を表すリストと** payoff **を計算する関数を作ると計算が可能になりそう．**

はじめに strategy **を表すリストを作る．**0 **ならば** Cooperation **，** 1 **ならば** Betray **としよう．**

```
strategy = Table[Random[Integer], {V[rg2004]}]
```

```
{1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1}
```

payoff **を計算する関数は** 0 **と** 1 **の組み合わせが４つ必要なだけである．例えば** cost-benefit ratio **を** r **とすると，こんな感じになるだろう．**

```
payoff[a_, b_, r_: 1.0] := If[a == 0, If[b == 0, 1, -r], If[b == 0, 1 + r, 0]]
```

```
Table[Table[payoff[i, j, r], {i, 0, 1}, {j, 0, 1}] // MatrixForm, {r, 0.1, 0.4, 0.1}]
```

$$\left\{\begin{pmatrix}1 & -0.1\\ 1.1 & 0\end{pmatrix}, \begin{pmatrix}1 & -0.2\\ 1.2 & 0\end{pmatrix}, \begin{pmatrix}1 & -0.3\\ 1.3 & 0\end{pmatrix}, \begin{pmatrix}1 & -0.4\\ 1.4 & 0\end{pmatrix}\right\}$$

自分と相手の戦略から自分の利益を計算する関数を定義する．

```
calcBenefit[x_Integer, y_Integer] := payoff[strategy[[x]], strategy[[y]], 0.4]
```

連結するすべての相手とゲームをしたときの利益の計算例．

```
Apply[Plus, Thread[calcBenefit[5, adjacencies[[5]] ]] ] / Length[adjacencies[[5]] ]
```

```
0.56
```

すべての点についてそれを行ってみる．

```
benefitList = Table[Apply[Plus, Thread[calcBenefit[i, adjacencies[[i]] ]] ] /
   Length[adjacencies[[i]]], {i, V[rg2004]}]
```

```
{0.7, -0.2, -0.05, 0.4, 0.56, -0.4, -0.225, 0.525, 0.56, 0.525, 1.05,
 0.7, 0.6, 0.466667, -0.12, -0.2, 0.933333, 0.16, 0.466667, 0.525}
```

となって，めでたく利益のリストを得ることに成功した．このリストを元にして戦略を変更することになる．

自分と連結している中からランダムに選んだ相手の利益を比べてみて，相手の方が利益が大きい場合，自分の戦略を変更することにしよう．自分の位置が与えられたときに，新しい戦略を返す関数を定義する．

```
degs = Degrees[rg2004]
```

```
{6, 7, 8, 7, 5, 2, 8, 8, 5, 8, 4, 8, 7, 9, 5, 7, 6, 10, 6, 8}
```

```
Table[Random[Integer, {1, degs[[5]]}], {50}]
```

```
{3, 3, 1, 1, 1, 4, 1, 1, 4, 2, 1, 2, 4, 4, 1, 5, 4, 4, 2, 3, 5, 5, 2, 1, 5, 4,
 1, 3, 3, 4, 5, 4, 5, 2, 3, 5, 5, 3, 2, 2, 3, 4, 1, 4, 3, 5, 2, 5, 4, 4}
```

```
newStrategy[x_] := Module[{diff, cbr = 0.4},
  diff = benefitList[[Random[Integer, {1, degs[[x]]}]]] - benefitList[[x]];
  If[diff ≥ 0,
   If[diff / (1 + 2 cbr) ≥ Random[], 1 - strategy[[x]], strategy[[x]]], strategy[[x]]]
 ]
```

```
Table[newStrategy[i], {i, V[rg2004]}]
```

```
{1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1}
```

これを繰り返し行うようなプログラムを作れば，与えられたグラフについて人口の変動をみることができる．すばらしい．

プログラムを作ってみよう

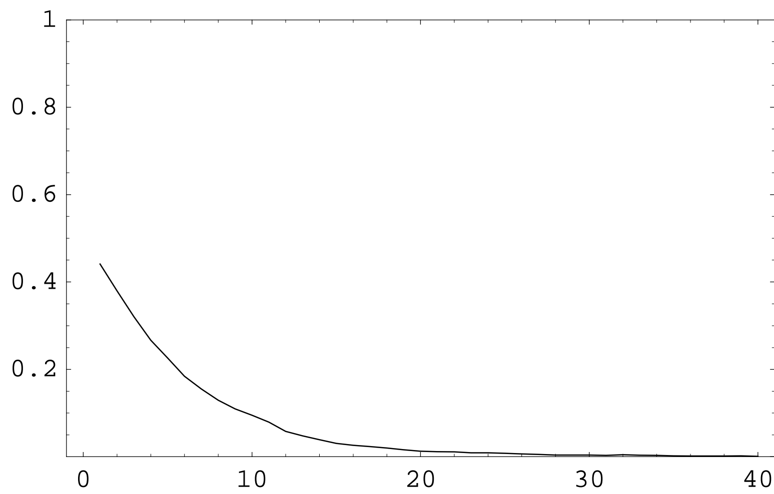
上の手順を振り返りながらプログラムを作ってゆく．グラフの形を引数に与えることにした．また，cost-benefit ratio **は変数に昇格して，グラフと同様に引数として与えることにした．最初に** Combinatorica **を読み込まないとエラーがでる．**

```
finalform[g_Graph, n_Integer, cbr_: 0.4] :=
 Module[{adjacencies, degs, strategies, benefitList, newStrategy},
  (* 関数の定義と初期化の始まり *)
  Needs["DiscreteMath`Combinatorica`"]; (* Combinatorica を読み込む *)
  adjacencies = ToAdjacencyLists[g]; (* 連結相手ののリストを作る *)
  degs = Degrees[g]; (* 連結相手の数を保存する *)
  strategies = Table[Random[Integer], {V[g]}]; (* 戦略の初期化 *)
  payoff[a_, b_] := If[a == 0, If[b == 0, 1, -cbr], If[b == 0, 1 + cbr, 0]];
  (* ペイオフマトリクスの定義 *)
  calcBenefit[x_Integer, y_Integer] := payoff[strategies[[x]], strategies[[y]]];
  (* 自分 x からみた相手 y との取引結果を計算する *)
  newStrategy[x_] := Module[{diff},
    diff = benefitList[[Random[Integer, {1, degs[[x]]}]]] - benefitList[[x]];
    If[diff ≥ 0, If[diff / (1 + 2 cbr) ≥ Random[],
      1 - strategies[[x]], strategies[[x]]], strategies[[x]]]
   ]; (* 戦略の変更を決定するルール*)
  strategiesUpdate[dummy_] := Module[{},
    benefitList = Table[Apply[Plus, Thread[calcBenefit[i, adjacencies[[i]] ]] ] /
       degs[[i]], {i, V[g]}]; (* 得られた利得の計算 *)
    strategies = Table[newStrategy[i], {i, V[g]}]; (* 新しい戦略の決定 *)
    1 - Apply[Plus, strategies] / V[g]
   ]; (* dummy 変数があるのは FixedPointList も使えるように配慮したため *)
  (* 定義と初期化の終了 *)
  Table[strategiesUpdate[0], {n}]
  (* FixedPointList[strategiesUpdate,0,n] *)
 ]
```

```
grph = RandomGraph[2000, 0.3]
```

```
-Graph:<598681, 2000, Undirected>-
```

```
ListPlot[finalform[grph, 40, 0.5], PlotJoined → True,
 PlotRange → {Automatic, {0, 1}}, Frame → True]
```

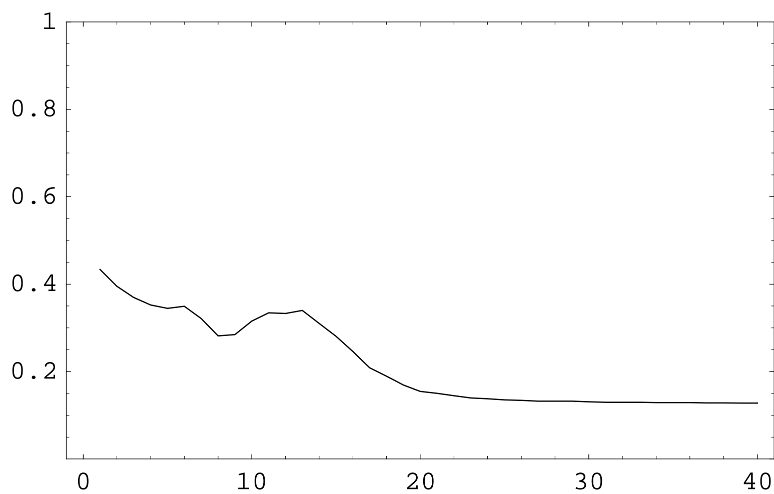



```
- Graphics -
```

```
grph = Wheel[2000]
```

```
-Graph:<3998, 2000, Undirected>-
```

```
ListPlot[finalform[grph, 40, 0.5], PlotJoined → True,
 PlotRange → {Automatic, {0, 1}}, Frame → True]
```




```
- Graphics -
```

参考文献

Albert,R. and Barábasi, A.-L., Rev. Mod. Phys., 74, 47-97, 2002

Barábasi, A.-L. and Albert, R., Emergence of Scaling in Random Networks, Science, 286, 509-512, 1999

Hauert,C.and Doebeli,M., Spatial structure often inhibits the evolution of cooperation in the snowdrift game,Nature Vol.428,643-646,2004

Watts, D. and Strongatz., S. H., Nature, 393, 440-442, 1998